VITASCOPE
Extensible and Scalable 3D Visualization of Simulated Construction Operations

by
Vineet Rajendra Kamat

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Civil Engineering

Doctoral Committee:
Dr. Julio C. Martinez, Chair
Dr. Douglas A. Bowman
Dr. Jesus M. de la Garza
Dr. Anthony D. Songer
Dr. Antonio A. Trani
Dr. Michael C. Vorster
Dr. Ronald R. Wakefield

April 14, 2003
Blacksburg, Virginia

Keywords: 3D Visualization, Animation, Construction Operations, Credibility, Discrete-Event Simulation, Validation, Verification, Vitascope

Copyright © 2003, Vineet Rajendra Kamat
VITASCOPE
Extensible and Scalable 3D Visualization of Simulated Construction Operations
by
Vineet Rajendra Kamat

Abstract
In the domain of operations design and analysis, the ability to see a 3D animation of processes that have been simulated allows for three very important things: 1) The developer of a simulation model can ascertain that there are no errors in the coding (Verification); 2) The experts, field personnel, and decision makers can discover differences between the way they understand the operation and the way the model developer understands it (Validation); and 3) A model can be communicated effectively which, coupled with verification and validation, makes it “credible” and thus used in making decisions. In the case of simulated construction operations, the existent body of knowledge and understanding did not generally permit modeled processes to be accurately visualized in 3D. The purpose of this research was to remedy this situation and find methods of describing animated 3D worlds that show how construction operations modeled using Discrete-Event Simulation were/can be carried out, using simple text statements and references to 3D CAD drawings. The fundamental question the work addressed was how to achieve accurate, dynamic, smooth, and continuous 3D animation of arbitrarily-complex simulated construction processes, based on meager pieces of operational information that can only be communicated when discrete events occur in simulation runs. The end result of this effort is VITASCOPE, an acronym for VIsualizaTion of Simulated Construction OPErations. VITASCOPE is a simple, parametric-text animation description language that is meant to be written out by end-user programmable software such as discrete-event simulation tools. Sequential instructions written in this language allow a computer to create a 3D virtual world that is accurate in time, space, and appearance; and that shows people, machines, and materials interacting as they build constructed facilities.
vi.ta.scope (va"ta*-skop’)

Noun
1. A machine for projecting on a screen a series of pictures, moved rapidly and intermittently before an objective lens, and producing by persistence of vision the illusion of continuous motion.
2. A form of machine for exhibiting animated pictures.
3. Any of several other machines or devices producing moving pictorial effects.
4. Acronym for VIsualizaTion of Simulated Construction OPERations

Etymology: Latin vita, life + -scope, to see.
Acknowledgments

I am grateful to my advisor, Professor Julio C. Martinez for his inspiration, wisdom, guidance, encouragement, and friendship. I am also indebted to my dissertation committee members, Professors Douglas A. Bowman, Jesus M. de la Garza, Anthony D. Songer, Antonio A. Trani, Michael C. Vorster, and Ronald R. Wakefield, each of whom significantly contributed to my graduate education in many ways.

I would like to express my gratitude to the Vecellio Fellowship Program in the Construction Engineering and Management division at Virginia Tech, and to the National Science Foundation for their financial support.

I am also grateful to Professors Photios G. Ioannou, Robert I. Carr, John G. Everett, and Nikolaos D. Katopodes at the University of Michigan. Your early confidence in this research provided the necessary boost that helped accomplish the work in good time.

Thanks to my cousin, Manoj Dharwadkar for your inspiration and guidance, and for encouraging me to pursue graduate education in the United States. Thanks to Niti and Leena Dharwadkar for your emotional support, understanding, and affectionate company.

Thanks to all my friends, especially Poorna and Sandeep Kalelkar, Nita and Prashant Pai Angle, Jose and Arliz Perdomo, Gunnar Lucko, Kyunghwan Kim, Harry Diaz, Juan Pinero, Narsinha Bodke, Vipul Dessai, Ketan Nadkarni, Gurpreet Khanuja, Niresh Naik, Saiprasad Naik, Ganapati Idoorkar, and Ramnath Raiturkar for your warmhearted friendship and company.

Most of all, thanks to my wife Sonia V. Kamat, my parents Rajendra A. Kamat and Geeta R. Kamat, my maternal grandparents Bhaskar H. Dalvi and Durga B. Dalvi, my parents-in-law Ratnakar D. Kossambe and Sunita R. Kossambe, my brother-in-law Salil R. Kossambe, and my entire family for your emotional support, encouragement, understanding, and affection.
Chapter 1

Introduction ................................................................. 1

1. Importance of the Research Activity ........................................ 1
2. Visualizing Simulated Construction Operations........................... 4
   2.1 Schematic Animation Tools.................................................. 6
   2.2 Generic 2D Process Animation Tool ..................................... 6
   2.3 3D Animation Tools Integrated with Manufacturing Simulation Systems ...... 7
   2.4 Generic 3D Process Animation Tool ..................................... 8
3. Research Objectives.................................................................. 9
4. Methodology ........................................................................ 11
5. Dissertation Outline .......................................................... 14
6. References............................................................................ 21

Chapter 2

Automated Generation of Dynamic, Operations Level
Virtual Construction Scenarios ............................................. 25

1. Introduction........................................................................ 25
   1.1 Project Level Visualization.................................................. 25
   1.2 Operations Level Visualization........................................... 26
2. Challenges in Operations Level Visualization.......................... 27
Chapter 3
Extensible and Scalable 3D Visualization of Construction Processes and Products

1. Introduction ........................................................................................................... 52
   1.1 Main Contribution ......................................................................................... 52
2. Research Motivation ............................................................................................. 54
3. Extensible and Scalable Framework ..................................................................... 56
   3.1 Challenges ..................................................................................................... 57
      3.1.1 Extensibility ......................................................................................... 57
Chapter 4
Dynamic 3D Visualization of Fluid Construction Materials ......................................................... 84
1. Introduction ......................................................................................................................... 84
2. Challenges ......................................................................................................................... 86
3. Initiative ............................................................................................................................. 87
4. Particle Systems ................................................................................................................. 88
4.1 Particle Attributes ......................................................................................................... 89
4.2 Recursive Model ............................................................................................................ 89
Chapter 5
Practical 3D Animation of Multiply Articulated Construction Equipment

1. Background........................................................................................................... 117
   1.1 Problem Description ................................................................................... 118
       1.1.1 Detailed Modeling for Animation ....................................................... 119
       1.1.2 Geometric Operational Details ............................................................. 122
   1.2 Main Contribution ....................................................................................... 124
2. Challenges ......................................................................................................... 125
3. Initiative ............................................................................................................. 126
   3.1 Kinematics .................................................................................................. 127
       3.1.1 Forward Kinematics ............................................................................. 127
       3.1.2 Inverse Kinematics ............................................................................. 128
Chapter 6
Large-Scale, Dynamic Terrain in Simulated Construction Process Visualizations ........................................ 149

1. Introduction .................................................................................................................. 149
   1.1 Research Motivation .......................................................................................... 150
   1.2 Main Contribution ............................................................................................ 151

2. Challenges .................................................................................................................... 152

3. Digital Representation of Terrains .............................................................................. 154

4. Visualizing Dynamic Terrains ................................................................................... 156
   4.1 Generating 3D Terrains from DEMs ................................................................... 156
       4.1.1 Specifying the Appearance of Generated Terrains ................................... 157
   4.2 Maintaining 3D Terrain Databases ..................................................................... 159
       4.2.1 Choosing the Appropriate Storage Model ................................................ 160
   4.3 Rendering 3D Terrain Models ............................................................................. 162
       4.3.1 View-Dependent Terrain Model Triangulation ......................................... 163
Chapter 7
Path Generation for Variable-Speed Resource Motion in Animations of Discrete-Event Process Models

1. Introduction................................................................. 184
1.1 Research Motivation ................................................... 184
1.1.1 DES Enforced Temporo-Spatial Inconsistencies .......... 185
1.1.2 Animation Method-Caused Spatial Discrepancies ...... 187
1.2 Main Contributions .................................................... 190
2. Challenges................................................................. 191
3. Representing Accurate Motion Trajectories ...................... 193
3.1 Choosing an Appropriate Interpolating Spline ............... 196
3.1.1 Suitability of Kochanek-Bartels Splines .................. 199
3.1.2 Parameterization by Arc Length ............................ 201
3.2 Describing 3D Paths in 2D ......................................... 203
4. Describing Variable Speed Motion .................................. 205
4.1 Hypothetical Hybrid Animation Approach .................... 206
4.2 Time-Based Scaling of Velocity Profile Shapes .............. 207
Chapter 8
Efficient Interference Detection in Automated 3D
Construction Process Visualizations

1. Introduction
   1.1 Main Contribution
2. Challenges
   2.1 Desiderata
3. Collision Detection Algorithms
   3.1 Problem Domain
   3.2 Relevant Taxonomy
   3.3 Algorithmic Optimizations
4. Technical Approach
   4.1 Pruning Scene Object Pairs
   4.2 Exact Pair-wise Collision Tests
      4.2.1 Construction of Bounding Volume Hierarchy
      4.2.2 Hierarchical Exact Overlap Test
5. C-COLLIDE
   5.1 Object Definition and Initialization
   5.2 Interference Detection Control
   5.3 Collision Feedback and Response
6. Advantages of Add-On Approach
7. Conclusion and Future Work
8. References
Chapter 9
Validating Complex Construction Simulation Models
Using 3D Visualization

1. Introduction
2. Verification and Validation
3. Earthmoving Operation Case Study
4. Discrete Event Simulation Model
5. VITASCOPE
6. Visualization of the Modeled Earthmoving Operation
   6.1 Verification
   6.2 Validation
   6.3 Insightful Visual Details
7. Conclusion
8. References

Chapter 10
Conclusion

1. Fundamental Capabilities Provided by VITASCOPE to Visualize Simulated
   Construction Processes
   1.1 Concatenation of Elementary Geometric Transformations
   1.2 Accurate Resource Motion Trajectories
   1.3 Variable Speed Resource Motion
   1.4 Automated Landscape Generation
   1.5 Dynamic Construction Jobsite Terrains
   1.6 Fluid Construction Materials
   1.7 Multiply Articulated Construction Equipment
   1.8 Interference Detection and Reporting
   1.9 Quantitative Run-Time Evaluation
2. Contributions........................................................................................................... 275

3. Implementation and Validation............................................................................... 277
   3.1 Validating Simplicity ...................................................................................... 280
   3.2 Validating Sufficiency .................................................................................... 280
   3.3 Validating Portability ...................................................................................... 281
   3.4 Validating Extensibility and Scalability ......................................................... 282

4. Directions for Future Research ............................................................................... 283
   4.1 Integration of Simulation and CAD Modeling Tools ..................................... 284
   4.2 Special-Purpose Animated Construction Simulators ...................................... 284
   4.3 Concurrent Animation of Field Construction Processes ................................. 285
   4.4 Interactive, Simulation-Driven Virtual Construction Environments .......... 286

5. References............................................................................................................... 287

Appendices ...................................................................................................................... 289
Vita.................................................................................................................................. 443
# List of Figures

## Chapter 2

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VITASCOPE Animation Schema</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>Snapshots of Visualized Operations</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Designing Higher-Level Language Constructs</td>
<td>47</td>
</tr>
</tbody>
</table>

## Chapter 3

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open, Loosely-Coupled Visualization Methodology</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>Extensible Visualization Scheme</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td>VITASCOPE Animation Event Loop</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>Augmented Dynamic Object Updating Process</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>VITASCOPE Add-On Entry Point Function Prototype</td>
<td>68</td>
</tr>
<tr>
<td>6</td>
<td>Function Prototypes for Registering Add-On Facilities</td>
<td>69</td>
</tr>
<tr>
<td>7</td>
<td>Add-On Facility-Registering Scheme</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>Function Prototypes for Executing Foundational Statements</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>Dump Truck Assembly and Manipulation with Foundational Statements</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>Higher-Order Statements for Dump Truck Assembly and Manipulation</td>
<td>74</td>
</tr>
<tr>
<td>11</td>
<td>Dump Truck Assembly and Manipulation with Higher-Order Statements</td>
<td>75</td>
</tr>
</tbody>
</table>

## Chapter 4

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basic Particle System Model</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>Defining a Water Distribution Truck</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>Animation Snapshot of a Water Distribution Process</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>Describing Concrete Placement Processes</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td>Animation Snapshot of Concrete Placement Process</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>Source Domains</td>
<td>104</td>
</tr>
<tr>
<td>7</td>
<td>Velocity Domains</td>
<td>104</td>
</tr>
<tr>
<td>8</td>
<td>Detecting Particle-Plane Collision</td>
<td>106</td>
</tr>
</tbody>
</table>
Figure 2: Piecewise-Linear 3D Motion Path Trajectory ......................................................... 188
Figure 3: Axial Orientations of 3D Objects ........................................................................ 188
Figure 4: Simulation Object Depicted with Incorrect Side Roll ........................................ 190
Figure 5: Abrupt Directional Changes on Linear Path Junctions ...................................... 194
Figure 6: Motion-Smoothering Effect of the Rear Guide Point ....................................... 195
Figure 7: Smooth Object Motion on Curved Trajectories................................................... 196
Figure 8: Piecewise Cubic Spline Curve ........................................................................ 197
Figure 9: Deriving Tangent Components at Control Points ............................................. 200
Figure 10: Effect of Changing the Tension, Continuity, and Bias Parameters .................. 201
Figure 11: Relation between Natural Spline Parameterization and Arc Length ................ 202
Figure 12: Problems in 3D Path Definition ..................................................................... 204
Figure 13: Superimposition of 2D Trajectories on 3D Terrain Models ............................ 205
Figure 14: Definition of Velocity Profile Shape ............................................................... 209
Figure 15: Derivation of Time-Scaled Velocity Profiles.................................................... 210
Figure 16: Moving Simulation Objects on Defined Motion Paths .................................... 214
Figure 17: Animation Snapshots of Terrain-Followed Object Motion ............................ 215

Chapter 8

Figure 1: Interference Detection Algorithm Architecture .............................................. 231
Figure 2: Axis Aligned Bounding Boxes ........................................................................ 233
Figure 3: Bounding Box Projections ............................................................................. 234
Figure 4: Oriented Bounding Boxes ............................................................................. 235
Figure 5: Building the OBBTree .................................................................................. 235
Figure 6: Traversing Bounding Volume Hierarchies to Detect Collisions ...................... 237
Figure 7: Searching for a Separating Axis ..................................................................... 238
Figure 8: Original and Triangulated CAD Model Representations ................................ 242
Figure 9: Animation Snapshot of a Bridge Construction Jobsite ................................... 244
Figure 10: Specification of Contextual Rules and Assumptions ................................... 245
Chapter 9

Figure 1: Process of Verification and Validation ......................................................... 252
Figure 2: Plan View of the Earthwork Operation Jobsite .............................................. 254
Figure 3: Modeling Flaw - Bi-directional Traffic in One-way Segment ......................... 259
Figure 4: Modeling Flaw - Near Collision at the Y-Intersection .................................... 260
Figure 5: Modeling Flaw - Rear-end Collision at the Main Loading Area .................... 261
Figure 6: Portion of the Initial STROBOSCOPE Model ............................................. 261
Figure 7: Bunched up Trucks Waiting to Enter the Narrow Segment .......................... 264
Figure 8: Bunched up Trucks at the Main Loading Area ............................................. 264
Figure 9: Bunched up Trucks at the Alternate Loading Area ...................................... 265

Chapter 10

Figure 1: Animated Segmental Concrete Bridge Construction Processes .................... 279
List of Tables

Chapter 2
Table 1: Geometric Transformation Level Decomposition .................................................. 41

Chapter 4
Table 1: Usage of Selected ParticleWorks Statements ....................................................... 98
Table 2: Usage of Higher-Order ParticleWorks Statements ........................................... 110

Chapter 5
Table 1: KineMach Implemented Backhoe Statements ....................................................... 140
Table 2: KineMach Implemented Lattice Boom Crane Statements ................................ 141

Chapter 6
Table 1: Usage of ViTerra Statements ............................................................................ 172

Chapter 7
Table 1: Usage of PathFinder-Implemented Animation Statements .............................. 212

Chapter 8
Table 1: Usage of Selected C-COLLIDE Statements ...................................................... 241
List of Appendices

A: List of Core VITASCOPE Language Statements ................................................................. 290
B: VITASCOPE Language Reference ..................................................................................... 293
C: VITASCOPE Virtual Environment Application Reference ............................................... 316
D: Segment of a VITASCOPE Animation Trace File ............................................................... 320
E: VITASCOPE Add-On Interface Reference ......................................................................... 323
F: KineMach Add-On for VITASCOPE Source Code ............................................................ 328
G: ParticleWorks Add-On for VITASCOPE Reference ........................................................... 400
H: KineMach Add-On for VITASCOPE Reference ................................................................. 408
I: ViTerra Supported Elevation Data Input Formats ............................................................... 428
J: ViTerra Add-On for VITASCOPE Reference ..................................................................... 429
K: PathFinder Add-On for VITASCOPE Reference ................................................................. 433
L: C-Collide Add-On for VITASCOPE Reference ................................................................. 438
M: ExcelWorks Add-On for VITASCOPE Reference ............................................................. 441
Chapter 1
Introduction

The purpose of this research was to find methods of describing animated 3D worlds that show how construction operations modeled using Discrete-Event Simulation were/can be carried out, using simple text statements and references to CAD models. The fundamental question this research addressed was how to achieve accurate, dynamic, smooth, and continuous animation of simulated processes, based on meager pieces of operational information that can be communicated only when discrete events occur in simulation runs.

The end result of this effort is VITASCOPE, an acronym for VIsualizaTion of Simulated Construction OPErations. VITASCOPE is a simple, parametric-text animation description language that is meant to be written out by end-user programmable software such as discrete-event simulation tools. Sequential instructions written in this language allow a computer to create a 3D virtual world that is accurate in time, space, and appearance; and that shows people, machines, and materials interacting as they build constructed facilities.

1. Importance of the Research Activity

Discrete-Event Simulation (DES) is a powerful objective function evaluator that is well suited for the design of construction operations. DES as applied to construction operations planning and analysis entails the creation of schematic process models that represent how construction operations will be performed. These models consider the different resources that are required to carry out the construction operations, the rules under which the different tasks that compose the operations are performed, the managerial decisions made during the operations, and the stochastic nature of events. Once the models are created, the modeled operations can be simulated in the computer and the statistical measures of performance for the operations can be studied. The results typically include the cost and time of construction as well as resource utilization rates,
waiting times and length at queues, etc. The results usually point out important parts of the operations with potential for improvements that may result in cost and/or time savings.

Considering these observations, the operations analyst may modify the models to reflect changes in operating procedures, resource allocations, space for temporary storage of materials, etc. The modified models can then be simulated and analyzed, with the results used to further improve the operations. The procedure continues until the operations analyst is convinced that no further improvements are possible. State-of-the-art construction simulation tools permit the modeling of complex construction operations at any level of detail (Martinez and Ioannou 1999). These tools can provide detailed insights on modeled construction operations such as expected resource utilization rates, resource idleness, operation bottlenecks, production rates, and the resulting expected cost, before commencing actual work in the field.

Notwithstanding these facts, there has been limited use of DES in planning and designing construction operations (Halpin and Martinez 1999, Tucker et al. 1998, Huang and Halpin 1994). A key reason for this is that simulation models and their results cannot be communicated effectively enough for them to be understood by typical decision makers in construction. Discrete-event modeling is an inherently complex activity that is both a science and an art. The modeling of a construction operation requires the description (in the language of the simulation tool) of mental plans that are often complex and elaborate. Differences between the mental plan and the operation actually modeled in a first attempt are ubiquitous. “Verification” is the process by which the model creator looks at what has been actually modeled, compares it to what was intended, and updates the model to accurately reflect the intention.

The developer of the computer simulation model, however, may have misconceptions about how the actual operation will take place in the field. Thus, a model may not be an accurate representation of reality despite proper verification by its developer. Such errors cannot be discovered by verification because the model indeed reflects what the model
creator intended. The aim of “validation” therefore is to determine whether simulation models accurately represent the real-world system under study. This is typically carried out by consulting persons who are intimately familiar with the operations of the actual system, but who are not necessarily proficient in simulation itself. Simulation models are termed as “credible” when the models and their results are accepted as being valid, and are used as an aid in making decisions (Law and Kelton 2000).

In the case of both verification and validation, the inner workings of a model and its output must be communicated to others for discussion and input in a way that is both comprehensive and lucid (Law and Kelton 2000, Oloufa and Ikeda 1997). Construction simulation tools typically provide results in the form of numerical or statistical data. However, they do not illustrate the modeled processes graphically. This poses significant difficulty in communicating the results and the logic of simulation models, especially to domain experts who are not trained in simulation itself. Decision makers often do not have the means, the training and/or the time to verify and validate simulation models based solely on numerical output (Ioannou and Martinez 1996). Potential practitioners are therefore always skeptic about simulation analyses and have little confidence in their results. This inability to elicit credibility is a major deterrent to the widespread use of simulation as an operations planning tool in construction.

The design of construction operations using simulation is justified only if the insights gleaned from the analyses are used in making decisions and in increasing understanding (i.e., the models are credible). A simulation model’s credibility directly depends on how effectively its results and internal workings (i.e. logic) can be communicated. There is thus a clear need to communicate simulated operations in a way that can conspicuously help verify, validate, and accredit simulation models. Visualizing simulated operations is an effective means of achieving this (Carson 2002, Law and Kelton 2000, Rohrer 2000, Jain 1999, Henriksen 1998, Tucker et al. 1998, Robinson 1997, Ioannou and Martinez 1996, Cox 1988, Biles and Wilson 1987).
It is a generally accepted fact that visually presented information is understood and grasped more easily than any other form of communication. By presenting information visually, the underlying data and/or logic can be understood quickly and easily. In the case of simulated construction operations, visualization can transfer the numerical results and underlying model logic into coherent visual images that allow the information to be quickly absorbed in a more natural and lucid manner. Volumes of data that take hours to review can be effectively communicated in a few seconds.

For instance, many techniques are available to simulation analysts to perform verification (e.g., looking at simulation logs). However, a 3D animation of what occurred in the simulation model can reveal such errors very quickly. Similarly, visually communicating the working of simulation models to field experts can allow errors in logic to be easily identified and corrected, and can allow planning groups to participate in discussions aimed at improving the modeled plan. Visualizing simulated construction operations can also provide several valuable insights about the modeled processes that are otherwise non-quantifiable and non-presentable, but are nevertheless critical for making decisions.

Advances in computer graphics allow the creation of very realistic animations in 3D. This is evident in computer games and digitally animated motion pictures. Advances in discrete-event simulation, on the other hand, permit the detailed modeling of very complex construction operations. The existing body of knowledge and understanding, however, did not generally permit a simulated construction operation to be accurately animated in a 3D virtual world.

2. **Visualizing Simulated Construction Operations**

In construction, different people understand different things by the term “visualization”. In particular, the term has been used in the literature to refer to any kind of series of sequential computer frames without taking into account their origin or their contents (Op den Bosch 1994). In effect, numerous computer-based visual activities that can be directly or indirectly used in construction planning may be appropriately termed visualization. These activities include, but are not limited to, the following:
• Manual coding and/or interactive creation of specific virtual construction scenarios using graphics libraries (e.g. World Tool Kit) or interactive tools (e.g. Bentley Dynamic Animator). Such works are described in Tseng et al. (2000), Fukuchi et al. (1999), Adjei-Kumi and Retik (1997), and Tsay et al. (1996).

• CAD model-based animation of CPM construction schedules (i.e. 4D CAD) (Koo and Fischer 2000, Cleveland 1989)

• Visualization of assembly sequences and real-time virtual interactive modeling of construction equipment (e.g. Interactive Visualizer ++) (Op den Bosch 1994)


• CAD model-based project information access over the internet using VRML (e.g. Lipman and Reed 2000, Campbell 2000)

This research differs from those listed above and is about enabling the smooth, continuous, dynamic 3D visualization of construction processes modeled using Discrete-Event Simulation tools. Modeled construction operations themselves have been previously animated in several ways. The choice influences the level of visually presented detail, and the information/insights that can be gleaned from the visualization process. These in turn determine the nature of modeling issues that can be verified and validated, and hence the amount of credibility that visualization can elicit for a simulation analysis. Previous techniques for “visualizing” simulated construction operations include:

1. Schematic process models overlaid with dynamic simulation statistics such as EZStrobe (Martinez 2001) and DISCO (Huang and Halpin 1994); and schematic process models overlaid with iconic animation such as COOPS-R (Liu and Ioannou 1993) and ABC (Shi 1999).

2. General, simulation system-independent, 2D post-processor animation tool called PROOF (Henriksen 1998).

3. 3D animation tools integrated with manufacturing simulation systems such as AutoMod (Rohrer 2000) and Quest (Donald 1998).

4. General, simulation system-independent, 3D post-processor animation tool called the Dynamic Construction Visualizer (Kamat and Martinez 2001).
2.1 Schematic Animation Tools

Schematic process models overlaid with run-time statistics have been previously used to illustrate the dynamics of simulated construction process models. This technique uses the models’ schematic networks as the background and overlays them (during simulation runs) with dynamic statistical data such as the simulation time, resource utilization rates, productivity etc. Iconic animation-integrated tools, on the other hand, animate the flow of resources during a simulation run by moving icons on the schematic process model background. The icons that represent model resources change position on the model’s network during a simulation run to reflect their flow in the simulated system.

Schematic visualization can be of help to simulation model developers in debugging models during the development (i.e. coding) stage. Debugging is a major step in model verification that helps determine whether a model contains any modeling flaws. Schematic visualization, however, is of little use in model validation. This is so because schematic visualization is rarely effective at communicating simulation models to persons who do not understand that particular simulation system. The process of validation, however, exclusively involves consultation with field experts who are typically untrained in any simulation tool and/or in simulation itself. In addition, several operational issues cannot be validated using schematic visualization altogether. For instance, validation of issues that pertain to the shortage of and conflicts in working space is difficult to achieve through schematic visualization even if space is explicitly modeled as a simulation resource.

2.2 Generic 2D Process Animation Tool

PROOF is a generic, post-processor visualization tool that supports 2D drawing and animation (Henriksen 1998). Changes in the state of a discrete-event process model are illustrated graphically in one plane of motion by changing the position, shape, and/or color of icons that represent resources. PROOF is independent of any particular discrete-event simulation tool. To create an animation, model developers use the software’s built-in drawing capabilities to create layouts and specify resource movement paths. The
model developer then instruments a simulation model to generate the required sequence of animation commands during a simulation run. PROOF then post-processes the recorded animation commands to visually depict the dynamic state of the modeled processes with chronological accuracy. PROOF has been effectively used in the past to visually communicate modeled construction (Ioannou and Martinez 1996, Martinez 1998) and mining (Sturgul and Seibt 1999) operations, as well as processes in other disciplines.

Construction operations are, however, rarely performed on one, flat, planar surface. Although effectively used in communicating some modeled construction processes, 2D animations inherently lack the real-world, 3D features that are indispensable to convincingly animate most construction operations. In particular, due to the degrees of freedom exercised in performing most construction processes, it is impossible to accurately portray construction in two dimensions. Examples of commonly encountered construction details that cannot be portrayed in 2D include the gradients of haul routes and other topographical characteristics in earthmoving operations, the vertical movement of resources in elevators and lifts in addition to the activities performed on the ground during building construction, and the accurate motion of complex construction equipment having multiple degrees of freedom.

2.3 3D Animation Tools Integrated with Manufacturing Simulation Systems

Several proprietary simulation tools developed for the manufacturing industry include integrated 3D animation capabilities. These tools include modeling constructs and built-in 3D templates of common manufacturing environments and equipment. These features allow users to effectively simulate and animate several common systems encountered in the manufacturing environment. Typical manufacturing operations are characterized by sequential process and assembly lines that are fixed in location and geometry. In contrast, construction operations are carried out in a more complex manner. They involve the transformation of space and the evolution of a product. In addition, they can be laterally distributed over a vast area (e.g. earthmoving, tunneling, paving etc.). Manufacturing
simulation tools are thus unable to effectively handle the additional complications introduced by the dramatic changes in the geometry of a construction site as work progresses (Tucker et al. 1998).

In addition, in all 3D-enabled manufacturing simulation systems, the simulation engines are tightly coupled with their built-in visualizers. Tight coupling between the simulation engine and the animation methods requires that the simulation model be created only using the native system’s simulation engine, which in most cases means that their suitability for modeling construction operations is limited (Martinez and Ioannou 1999). Furthermore, this also compels model developers desirous of animating their models in 3D to learn and use a different simulation tool than the one they are proficient with. For instance, a GPSS (Schriber 1995) user intending to visualize simulated processes in 3D would have to entirely recreate (i.e. recode) the GPSS simulation models in the language of a different, 3D-enabled simulation tool. This is highly unacceptable since the time and effort invested by typical modelers in achieving proficiency in a particular simulation tool of choice is phenomenal.

2.4 Generic 3D Process Animation Tool

The Dynamic Construction Visualizer (DCV) is a generic, simulation system-independent, post-processor visualization tool that supports 3D animation of discrete-event process models. The DCV is a prototype tool that was designed and implemented as part of the preliminary work that inspired this research. The tool implements a very limited, straight-line 3D animation language for animating modeled processes. The DCV is capable of animating a limited number of simulated construction operations in 3D, but at a level of detail and accuracy that is insufficient to extend complete credibility to most simulation models.

In particular, the DCV is only capable of portraying 1) processes in which simulation objects (i.e. resources) move at constant speeds, 2) processes in which involved resources do not undergo any shape deformation, 3) processes that do not involve unstructured or fluid materials (e.g. concrete), and 4) processes in which resources (equipment and crew)
always move on fixed, well-defined motion paths. The prototype DCV tool essentially provides visualization capabilities equivalent to that available in the 3D-enabled manufacturing simulation tools described in the earlier section, but decoupled from any specific simulation tool. In terms of expressiveness, it shares the limitations of the manufacturing simulation tools described earlier.

3. Research Objectives

Visualizing simulated construction operations accurately in 3D is a complex proposition. In order to visualize a construction process, it is necessary to see, in addition to the physical components of a facility, the equipment, personnel, materials and temporary structures required to build it. It is necessary to accurately portray the movements, transformations and interactions between the virtual simulation objects (i.e. construction resources). The movements and transformations of these objects must be spatially and temporally accurate. In order to depict smooth motion, visual elements must be shown at the right position and orientation several times per second. Issues such as arbitrary trajectories in 3D space, speed and acceleration, unstructured fluid materials, and deformations of visual entities must be carefully considered.

Accurate 3D visualization of modeled construction processes involves being able to see graphically on the computer, the operations being carried out in the same way as they would be in the real world. In the case of a simulated world, this includes the same logical and physical relationships that are embedded in the underlying discrete-event process models. In discrete-event simulation, the state of models change only at discrete, but possibly random, sets of simulated time points (Schriber and Brunner 1999). These time points are typically the start or end of activities, and it is only then that a discrete-event simulation can communicate with other processes, or perform other actions such as input/output. This is in contrast to continuous simulation, where the state of a model is monitored continuously at every time instant using differential equations of motion (Law and Kelton 2000).
The fundamental question that this research addressed, then, was how to achieve accurate, dynamic, smooth and continuous motion, based on meager pieces of information that can be communicated only when discrete events occur in a simulation. The specific nature of the communication cannot be determined a-priori by a general purpose simulation tool, as it is dependent on the specific operation being modeled. Thus, this communication has to be achieved by end-user programming of the simulation system. Such communication therefore has to be based on a notation that is both expressive (to achieve realistic visualization) and simple (so that it can be generated by end-user programming).

In addition, the notation itself has to be open and loosely coupled so that it is independent of any particular simulation tool. This is necessary to allow 3D animation capabilities to simulation modelers without compelling them to recreate their models in a different, unfamiliar, and perhaps unsuitable 3D animation-enabled simulation tool. Furthermore, an open and loosely coupled methodology can enable sources, other than simulation models, to communicate dynamic visualizations.

To address these issues, it was necessary to design a simple 3D animation authoring language to allow a discrete event process model (or other operations analysis tools) to generally communicate a dynamic visualization. The animation language needed no flow control (i.e., “while” or “for” loops) because that is achieved by the generating process (e.g., the simulation model while it runs). In essence, what was needed was a straight-line language that is capable of succinctly describing construction in all its complexity by properly illustrating all common construction tasks.

The goal of this research was thus to find methods of describing accurate animated 3D worlds that show how modeled construction operations were carried out, using simple parametric-text statements and references to 3D CAD models. The following were identified as the work’s specific objectives:
• Determine the information required to accurately and unambiguously describe, both spatially and temporally, a complex construction operation so that it can be recreated in a virtual world. This includes the movements and transformations of the people, machines, materials and the evolving constructed facility.

• Design a straight-line, 3D animation description language that is simple enough to be generated by end-user programmable software such as a discrete-event simulation tool. By necessity, programs in this language would be arbitrarily long.

• Implement the designed 3D animation language on multiple computing platforms in a scalable and user-extensible framework that will allow others to seamlessly extend the language.

4. Methodology

The steps and results enumerated below outline the research methodology:

• Analyzed very carefully how discrete-event simulation tools work in terms of being able to communicate information to describe a dynamic visualization while a model runs. The result of this analysis is summarized in the previous section “Research Objectives”.
  o Identified that any animation-describing communication from simulation models should be based on a simple yet expressive parametric, animation description language.

• Studied and analyzed several hierarchical taxonomies of construction operations in order to identify an appropriate level of classifying construction work from the perspective of the problem being addressed.
  o Identified elementary geometric transformations as the required building blocks of the notation for generally describing a smooth, continuous construction operation in 3D.

• Designed a set of simple, expressive, geometric transformation-based, parametric animation statements that could collectively describe the rigid elementary motions resources undergo as they perform construction work.
  o Animation statements in this set constitute the core VITASCOPE animation description language.
• Implemented the core language in a portable, virtual environment application that parses, interprets, and processes sequential animation instructions; and recreates a dynamic, smooth, continuous visualization of the communicated operations using references to 3D CAD models of the involved construction resources.

• Tested the efficacy of the core animation language in being able to faithfully represent simulated (and communicated) construction operations in 3D.
  o Instrumented several discrete-event process models to automatically generate animation trace files in the VITASCOPE language as they run.
  o Post-processed the generated trace files in the virtual environment application and confirmed that the communicated construction operations could indeed be depicted in dynamic 3D virtual worlds.
  o Performed a detailed simulation and visualization case study to evaluate 3D animation’s capabilities in verifying, validating, and accrediting simulated construction operations.

• Investigated techniques to allow the core 3D animation description language to be seamlessly extended by others.
  o Implemented the designed methods in a scalable and extensible framework that defines an extension (add-on) interface to the core VITASCOPE animation language.
  o Validated the implemented add-on interface by subsequently utilizing it to design several non-trivial extensions to the core animation language.

• Investigated techniques to textually communicate and accurately portray volumes of ubiquitous fluid construction materials such as concrete, mortar, water, and slurry in animated virtual construction worlds.
  o Implemented the designed animation statements as extensions to the core VITASCOPE animation language in an add-on named ParticleWorks.

• Designed and implemented “smart” pieces of virtual construction equipment that can be instantiated and manipulated using simple text statements in a higher-level, contextual, construction work-like terminology.
  o Implemented the designed higher-level animation statements in an add-on called KineMach using VITASCOPE’s flexible add-on interface.
• Developed automated techniques to define and portray dynamic, deformable terrain in animations of simulated construction processes
  o Designed mechanisms to automatically generate photorealistic, digital, 3D terrain CAD databases (i.e. models) to represent construction jobsites by combining readily available digital topographical (e.g. USGS Digital Elevation Maps) and aerial imagery (e.g. National Aerial Photography Program) data.
  o Investigated techniques to compute and visually depict the evolution of virtual construction jobsites by describing terrain surface deformations in response to animated construction processes such as digging and dumping dirt.
  o Implemented the automatic terrain generation mechanisms and the techniques that accurately compute and portray deformations to their surface as a VITASCOPE add-on called ViTerra.

• Explored techniques to portray the accurate, variable-speed motion of simulation objects (i.e. construction resources) on realistically shaped motion trajectories in animations of simulated processes.
  o Designed simple, parametric-text, simulation model-authorable statements to define and manipulate curved trajectories of arbitrary shape and length to represent 3D resource motion paths.
  o Devised geometric techniques to accurately describe the three-dimensional spatial configuration of virtual construction resources as they travel on defined motion trajectories during animation.
  o Formulated a unique computation scheme to portray variable-speed motion of simulation objects, using only a one-time, simulation model-authorable, parametric-text definition of the desired velocity profile shape and per-instance activity timing information (i.e. start time and duration).
  o Implemented the designed statements and computation techniques in an add-on named PathFinder using VITASCOPE’s add-on interface.
• Designed and implemented techniques to identify and report undesirable conflicts (i.e. interferences and/or collisions) that can occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources in animations of simulated construction operations.
  o Implemented the designed techniques in another add-on called C-Collide by capitalizing on VITASCOPE’s extensible framework.
• Developed a text statement-controlled dynamic charting tool to graphically animate simulation statistics alongside 3D view ports during animation of simulated construction processes.
  o Designed animation methods (i.e. statements) to control MS Excel chart sheets using Excel’s OLE automation capabilities.
  o Implemented the dynamic charting statements in an add-on called ExcelWorks using VITASCOPE’s add-on interface.
• Developed a capstone simulation-driven animation featuring processes involved in the construction of a cast-in-place concrete segmental bridge to demonstrate the combined efficacy of the core and extended animation language statements in being able to accurately portray simulated construction operations in 3D. The exercise also helped validate VITASCOPE’s simplicity, sufficiency, scalability, and extensibility.

5. Dissertation Outline

This dissertation is a compilation of scientific manuscripts that document the research involved in designing VITASCOPE and its add-ons. In particular, chapters 2 through 8 are stand-alone papers that describe details of individual scientific questions answered, challenges encountered, alternatives considered, and methods adopted in addressing a specific research issue or a set of related research issues. The paper in chapter 9 presents a detailed case study documenting the efficacy of 3D animation in verifying, validating, and accrediting complex construction simulation models. The dissertation concludes with chapter 10, which summarizes the contributions and achievements of this research, and
suggests directions for future work. Since the chapters are written as stand-alone manuscripts, certain information may be repeated across chapters.

All chapters have been written so that they can be read and understood by a wide audience, including people without any prior exposure to 3D computer graphics and/or computer programming. Working knowledge of discrete-event simulation is helpful, but not required. Preliminary work that inspired this research is described in detail in (Kamat 2000), which is available from the internet at the address indicated in the “References” section of this chapter. Perusal of that work will enhance the reader’s understanding of this dissertation and is therefore recommended, but not requisite.

The manuscripts in this dissertation discuss all aspects of VITASCOPE’s core and extended 3D animation description language. Several annotated examples of describing animations using the language are presented throughout the papers. However, due to the scientific nature of the manuscripts, each animation statement and detailed instructions on their usage are not included, i.e. the manuscripts are not user-manuals. Information of that nature is presented in detail in the dissertation’s appendices. To provide a clearer picture of the remaining dissertation chapters, their titles and brief contents are presented below:

- **Chapter 2 – Automated Generation of Dynamic, Operations Level Virtual Construction Scenarios**

This paper describes research that led to the design of the core VITASCOPE 3D animation description language. The questions addressed in the work are: 1) What minimum information is required to accurately describe a smooth, continuous construction operation in 3D?; 2) To what extent can that information be extracted from running discrete-event simulation models?; and 3) How can any animation-describing information be communicated from discrete-event simulation tools to 3D computer graphics facilities in a general, open, and loosely-coupled manner?
First, general techniques to enable operations analysis tools (e.g. simulation models) to communicate dynamic 3D animations of construction processes are investigated. Second, an appropriate level (based on elementary geometric transformations) in the hierarchical taxonomy of field construction processes is identified for the purpose of designing a specific 3D animation description notation. Finally, a set of core animation language statements are designed, implemented, and tested.

### Chapter 3 – Extensible and Scalable 3D Visualization of Construction Processes and Products

This paper describes the research that led to the design of mechanisms that allow the core VITASCOPE 3D animation description language to be seamlessly extended by others. The questions addressed in the work were: 1) How can the designed generic 3D animation description language be implemented in an extensible framework that is open to contributions from others?; 2) How can the complexity of that framework’s implementation be concealed from others so that extensions to the language can be designed without having to modify or understand the existing implementation?; and 3) To what extent can a balance between competing groups of technical requirements (i.e. completeness, efficiency, and simplicity versus adaptability and flexibility) be achieved in designing the extensible framework?.

First, a set of core animation statements of the most general use from the extension viewpoint is identified. Second, methods to design an add-on interface to the identified core animation methods are investigated by capitalizing on documented principles of application framework design. Finally, the designed add-on interface and its scalability are validated by implementing the extensible framework on multiple computing platforms and then extending the core animation language with several non-trivial extensions using the designed add-on interface.
Chapter 4 – Dynamic 3D Visualization of Fluid Construction Materials

This paper describes work that led to the design of automatic, simulation-driven methods to accurately visualize volumes of ubiquitous fluid construction materials (e.g. concrete, mortar, dirt, slurry) in animated 3D construction worlds. The questions addressed in the research were: 1) How can fluid construction material volumes of arbitrary shape, size, and appearance be described in simple, succinct, parametric-text statements?; 2) How can the dynamics of fluid construction material volumes be mathematically described in a way that facilitates efficient computing?; and 3) How can irregular, complex, and ill-defined fluid material surfaces be generally represented inside 3D virtual worlds?.

First, feasible procedures to accurately describe the motion and dynamics of arbitrary volumes of fluid construction materials are investigated using principles of Newtonian mechanics where appropriate. Second, simple methods to encapsulate such physical descriptions in a finite number of parametric-text statements are designed. Finally, real-time techniques to represent fluid construction material volumes in a visually convincing and realistic manner are explored. The work capitalizes on a classical technology – called Particle Systems - of describing fluid objects in interactive virtual environments.

Chapter 5 – Practical 3D Animation of Multiply Articulated Construction Equipment

This paper describes research that led to the design of “smart” pieces of complex, articulated, virtual construction equipment that can be instructed (by simulation models) to perform basic construction tasks instead of elementary geometric motions. The task instructions can be communicated using a high-level, construction work-like terminology. The research questions this work addressed were; 1) How can elaborate construction tasks performed by multiply-articulated pieces of equipment be encapsulated in terse, parametric statements of text?; and 2) How can the amplitudes and directions of intermediate elementary motions of components of equipment pieces that perform the communicated construction tasks be generated automatically?.
First, simple, high-level, parametric-text methods (i.e. statements) that simulation models can use to communicate basic construction tasks to virtual equipment pieces are designed. Second, techniques to compensate for simulation models’ lack of information about the sub-tasks that comprise each communicated basic task are investigated. Finally, techniques to alleviate simulation models from encapsulating several spatial details about modeled processes that in the absence of animation would otherwise be ignored are explored. The work capitalizes on robust principles of forward and inverse kinematics to design and implement the generic pieces of multiply-articulated virtual construction equipment.

**Chapter 6 – Large-Scale, Dynamic Terrain in Simulated Construction Process Visualizations**

This paper describes work that led to the design of techniques to 1) automatically generate 3D terrain databases (i.e. models) to represent virtual construction jobsites; and 2) visualize the deformation and evolution of the generated jobsite terrains in response to common animated construction processes (e.g. digging and dumping dirt). The main questions that the research addressed were: 1) How can digital elevation and aerial imagery data sources be efficiently synthesized such that the resulting 3D terrain databases (i.e. models) are an accurate visual representation of construction jobsites?; and 2) How can the interaction between virtual pieces of terrain-deforming construction equipment and the automatically generated terrain models be described accurately and efficiently?.

First, techniques to automatically generate terrain CAD databases from readily available digital elevation and aerial imagery data sources are identified and implemented. Second, feasible data structures to internally maintain and locally manipulate (on demand) the generated terrain databases are investigated. Third, methods to compute the amount, location, and the resultant shape of deformations that virtual terrains undergo in response to animated construction processes; and techniques to apply those deformations to the terrain databases in real-time are designed. Finally, computationally efficient techniques
of maintaining and rendering the dynamic 3D terrain CAD databases at interactive animation frame rates are identified and adopted.

**Chapter 7 – Path Generation for Variable-Speed Resource Motion in Animations of Discrete-Event Process Models**

This paper documents research that addresses the problem of describing the accurate, variable-speed motion of simulation objects (i.e. construction resources) on realistically-shaped trajectories (i.e. paths) in animations of simulated construction processes. The research questions successfully addressed were: 1) How can complex, curved, 3D motion path trajectories of arbitrary shape and length traversed by typical construction equipment be accurately defined and manipulated in succinct, parametric statements of text?; 2) How can the correct orientation of virtual construction resources (e.g. equipment pieces) be calculated accurately along all three axes (i.e. yaw, pitch, and roll) when they travel on paths or stand still inside animated 3D worlds?; and 3) How can simulation objects (i.e. virtual construction resources) move on defined paths with arbitrary velocity profiles when the motion start and end time instants are the only available pieces of operational information?.

First, techniques of mathematically representing flexible, arbitrarily-complex curves to describe realistic resource motion trajectories are examined. Second, geometric computation techniques for describing the interaction of moving simulation objects with virtual 3D terrain databases (i.e. models) are investigated in order to accurately compute and depict an object’s spatial configuration during motion. Finally, techniques of compensating for the lack of operational information (about activity instances) available from simulation models are explored. This leads to the design of animation methods that move simulation objects with uneven, arbitrarily-shaped velocity profiles that comply with an animation-authoring simulation model’s temporal integrity.
Chapter 8 – Efficient Interference Detection in Automated 3D Construction Process Visualizations

This paper describes the research that led to the design and implementation of effective techniques to identify and report all undesirable conflicts (i.e. collisions) that can occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources in 3D animations of simulated construction processes. Efficient and accurate methods for automated construction process level interference detection and conflict analyses at interactive animation rates are examined. The designed computation scheme puts in place an integrated framework for performing combined design-level, activity-level, and process-level temporo-spatial interference analyses that is of significant help in the verification and validation exercises of construction process models.

Chapter 9 – Validating Complex Construction Simulation Models using 3D Visualization

This paper presents a detailed case study of a complex simulated earthmoving operation. The efficacy of 3D animation in verifying and validating the complex simulation model is presented. In addition, examples of subtle, critical operational details made apparent by 3D visualization are described. Such details, although critical in making decisions, would otherwise be non-quantifiable and non-presentable.

Chapter 10 – Conclusion

This chapter summarizes the contributions and achievements of this research. The exercise conducted for validating the work’s tangible product (i.e. the VITASCOPE language) is presented in detail. Information on the capstone simulation-driven animation featuring processes involved in the construction of a cast-in-place concrete segmental bridge is also included. The chapter concludes the dissertation by suggesting specific directions for interesting future research.
6. References


Chapter 2
Automated Generation of Dynamic, Operations Level Virtual Construction Scenarios

1. Introduction
Planning and control of construction can take place at the project and/or the operation level (Halpin and Riggs 1992). At the project level, a facility is broken down into activities each of which maps to a physical project component (e.g., second floor columns) or to a major time-consuming process (e.g., order and delivery of kitchen cabinets). The planner uses techniques such as the Critical Path Method (CPM) to estimate the time frame during which activities can take place and the times at which important project milestones can be reached.

At the operation level, planning and control are concerned with the technological methods, number and type of resources, and logical strategies required to accomplish an activity or a group of related activities (e.g., erect second floor columns). The effort focuses on work at the field level. The interactions between equipment, labor, materials and space are considered explicitly in the performance of tasks (e.g., lower hook, attach and lift column). The same tasks may be repeated many times, using non-deterministic durations described by probability distributions. Although the planning and control techniques used at the project and operation levels are different, both can benefit substantially from dynamic 3D visualization.

1.1 Project Level Visualization
Numerous research studies have explored and exploited dynamic 3D visualization at the project level. The efforts are motivated by the shortcomings of traditional scheduling and control techniques such as Bar Charts and CPM in being able to represent all aspects of construction necessary for project level planning (Skolnick 1993). Visualization is achieved by linking a 3D CAD model representing the design of the infrastructure and a
construction schedule (Cleveland 1989). This form of visualization has popularly become known as 4D CAD.

4D CAD focuses on the visualization of the construction product over the period of its construction. As time advances, individual components (CAD elements) of the facility are added to the visual model in their final position and form as dictated by the schedule. 4D CAD has been demonstrated to provide construction planners with tangible visual insights that can be of help in planning and controlling construction at the project level (Koo and Fischer 2000). This type of visualization is being exploited because the appropriate technology is straightforward and available. Commensurate with the importance given to project level planning (relative to operations level planning), significant research effort has been invested in improving 4D visualization, both by academia and the industry (e.g. McKinney et. al. 1996, Bentley Systems 2000).

1.2 Operations Level Visualization

The value of visualizing construction at the operations level is obvious. Being able to visualize construction sequences and operations can result in tremendous savings in money and time and help keep projects on schedule (Alciatore et al. 1991). Visualizing construction operations in 3D can permit the complete subjective analysis of construction processes. Subtle details such as maneuverability problems at loading and dumping areas in earthmoving operations, the restricted visibility of crane operators in steel erection and lifting, overcrowding in work zones due to simultaneous execution of different trades in building construction, and a host of other safety problems such as potential collision between two machines can easily be deciphered by visualizing the actual construction operations that lead to the completion of the constructed product.

In addition, visualizing construction processes in 3D can allow the validation and verification of operational concepts; enable checking for design interferences; and facilitate overall constructability review and the sharing of project information. It can also enable the testing and validation of construction sequences, checking for physical
clashes of moving pieces and enable communication/coordination among multiple project participants.

Researchers and industrial proponents of 4D CAD have always been aware of the importance of operations level planning in general and operations level visualization in particular. The awareness of the importance of operations planning and visualization is evidenced by recent 4D CAD research works that aim to convey operations planning information about construction space requirements through 4D visualizations (Riley 1998, Akinci and Fischer 2000). In addition, the need to integrate product and process visualization to encompass both project and operations level planning has also been acknowledged in the literature (Griffis and Sturts 2000).

2. Challenges in Operations Level Visualization

Visualizing construction at the operations level is however a complex proposition that involves being able to view the interaction of the various resources as they build the product or perform a support service. These resources include, but are not limited to, temporary structures, materials, equipment, and labor as they create the product. At this level of detail, visualization of the evolving product can be naturally achieved as a by-product of visualizing the operations that build it.

Visualizing construction operations also encompasses construction procedures that do not necessarily involve the assembly of a tangible product such as a building or a bridge. For instance, construction operations such as paving, tunneling, quarrying, and earthmoving can obviously be visualized at the operations level. However, at the project level, construction of this nature can only be planned in terms of the desired production rate and has no corresponding visualization (i.e. 4D CAD) context due to the absence of a tangible, laterally limited product that requires assembly.

The planning information that 4D visualization synthesizes is derived from project level planning tools (i.e. CPM schedule and CAD model of the infrastructure). It is not possible to visualize the actual construction operations that lead to the construction of the
end product using the sources of 4D CAD (Adjei-Kumi and Retik 1997, Fukai 2000). In
other words, 4D visualization can depict the evolution of the construction product but not
the interaction of the resources that build it. “True” visualization of construction at the
operations level involves being able to “see” graphically on the computer, the operations
being carried out in the same way as they would be in the real world. The practical and
educational benefits of being able to visualize construction at this level of detail are
phenomenal.

In order to visualize an operation it is necessary to see, in addition to the physical
components of the facility, the equipment, personnel, materials and temporary structures
required to build it. Moreover, it is necessary to depict the movements, transformations
and interactions between these visualization elements. The movements and
transformations must be spatially and temporally accurate. In order to depict smooth
motion, visual elements must be shown at the right position and orientation several times
per second. Issues such as trajectories in 3D space, speed and acceleration need to be
considered. Due to the amount of detail and precision involved, visualizing construction
at this level is a challenging prospect.

3. Previous Research and its Limitations

Many researchers have discussed the potential of 3D operations visualization and virtual
interactive environments; some with futuristic perspectives that approach science fiction
and that assume that the enabling technology will in time be developed by others. A
limited number of researchers have actually experimented with specific scenarios of very
short duration (relatively speaking) resulting from long term efforts dedicated to the
creation of specific cases (Wakefield and O’Brien 1994, Tsay et al. 1996, Fukuchi et al.
1999, Tseng et al. 2000). Software toolkits (e.g. World Tool Kit) and specific higher-
level tools (e.g. 3D Studio, Bentley Dynamic Animator) are available for doing this
interactively with the same (significantly large) ratio of ‘time invested to develop the
specific case’ to ‘actual duration of the visualization experience’.
Individual developed scenarios and cases can, to an extent, be applied in teaching and training. Such cases however have little applicability in planning and designing construction operations. This is because large scale efforts, a significant amount of time, and sophisticated computing skills, all of which are typically at a premium in a construction setting, are required in developing each individual scenario. In order to effectively apply visualization and virtual reality technologies to construction operations design, we must be able to rapidly and easily generate alternate virtual construction scenarios for comparison and evaluation. This is an absolutely essential feature without which no methodology or tool can be practically used for operations analysis and design.

Conceptually, construction operations can be visualized by linking together discrete-event simulation models and CAD models of the infrastructure as well as of the construction equipment (i.e. machines), temporary structures, and other resources. The result would be 3D animations of simulated (modeled) construction operations. Discrete-event construction process simulation tools facilitate rapid analyses of alternate construction scenarios and provide quantitative guidelines to compare them. However, in order to graphically depict a modeled scenario in a virtual world, such tools must collaborate with appropriate computer graphics facilities (data structures, algorithms and routines). In addition, to cause objects under the control of such simulations to be aware of, and react to, humans and human controlled machines (via hardware controls), the virtual environment (VE) implementation must communicate bi-directionally and at high speed with the simulations, and the controls, each of which could be running in another process and perhaps in another machine.

Discrete-event simulation tools and other processes capable of rapidly generating construction scenarios cannot communicate directly with 3D computer graphics facilities. This is why visualizations depicting construction scenarios have to be either “hard-coded” using high-level languages and graphics libraries or developed interactively using user-interfaces (typically form-based) provided by specific higher-level tools. The large, long-term efforts required in developing such individual cases (some involving interactive hardware controls) renders their use in construction operations design
impractical. Additional discussion on previous research and the state-of-the-art in construction operations visualization can be found in (Kamat and Martinez 2001) and (Kamat and Martinez 2002).

4. Potential of Construction Operations Visualization

Operations level visualization can be potentially utilized in many ways in construction practice and education. Using available CAD models of the infrastructure and the resources, it can be possible to re-create in a virtual world what happened in the past, what is currently happening (from real-time field data), or what may happen in the future (by showing what was simulated by a discrete-event process simulation model). These visualizations can be very realistic, with accurate depictions of construction sites, infrastructure, equipment, and atmospheric conditions (visibility, fog, rain). Historical (from past data) and predicted (from simulations) animations can be in compressed or expanded time. A 20 second incident could be studied in very slow motion. General operations, in contrast, could be animated in fast motion so that several hours of operations are viewed in a few minutes. By rapidly generating and visualizing construction scenarios in virtual worlds, constructors can study the difference between alternative construction methods, materials, labor levels and management strategies with speed and accuracy at very low cost.

Beyond dynamic 3D animations limited to navigation (i.e., walkthrough), the possibility of interacting and controlling an operation in real time have the potential to add significant value beyond passive visualization as far as learning, operator training, and operations design is concerned. To realize such a vision, and depending on the application, visualization has to be possible on all hardware platforms (Immersive and Non-Immersive). In addition, the visualization application needs to communicate bi-directionally and at high speed with simulation models running in another process and perhaps in another machine. The technology needed to animate operations passively (i.e. walkthrough only) must form the basis for advances that will enable immersive, interactive construction environments to be commonplace. In these virtual environments, objects under the control of simulation models will be aware of, and react to, humans and
human controlled machines. The impacts to learning, operator training, and operations design are phenomenal.

As evidenced by many references to work throughout this proposal, construction researchers see tremendous potential in the uses of 3D operations visualization and have significant related questions to investigate. The unavailability of proper information technology has limited the number and value of discoveries that may dramatically reduce the life-cycle costs of constructed facilities.

5. Main Contribution

Several stages of research are necessary to facilitate rapid automatic generation of alternate construction scenarios (of any duration and complexity) in immersive and interactive 3D virtual worlds. The first of those research stages is the specification and design of an interface that can facilitate communication between computer graphics facilities and external processes capable of automatically describing dynamic construction operations. This interface must define a necessary layer of abstraction that effectively separates visualizations from the processes that generate them. This is essential to facilitate communication between diverse mathematical construction operations analyses tools (e.g. discrete-event simulation systems) and virtual worlds. Such a common interface is also critical if multiple processes and controls are to simultaneously interact with visualizations in real time.

The presented research investigates methods to define such an interface and exploit it to automatically and rapidly generate operations level virtual construction scenarios by facilitating inter-process communication. The work has the following specific objectives:

- Design a specific description that external software (e.g. simulation models) and hardware (e.g. controls) processes can use to communicate a dynamic construction operation.
- Implement a virtual environment application that interprets and processes instructions in that description to recreate a virtual world representation of the communicated operation.
• Validate the effectiveness of the description in facilitating automatic, rapid generation of operations-level virtual construction scenarios using data communicated by external processes.

6. Technical Approach

External processes communicating a dynamic operation in real-time implies that the specific nature of the communication cannot be determined a-priori, as it is dependent on the specific instance of the operation being simulated and/or communicated. Thus, this communication has to be achieved by end-user programming of the driving process (e.g. a discrete-event process simulation model). The transfer of information from external processes to the computer graphics facilities therefore needs to be based on methods that are both expressive (to achieve realistic visualization) and simple (so that they can be generated by end-user programming). The methods have to be open and loosely coupled so that 1) They are independent of any particular driving process (e.g. a specific simulation system) and 2) Processes, other than simulation models, can simultaneously interact with a dynamic visualization.

To achieve the research objectives while addressing these issues, we designed a set of parametric statements that allow external processes to communicate a dynamic visualization. The designed statements together define the specification of a 3D animation language named VITASCOPE (acronym for VIsualizaTion of Simulated Construction OPErations). The VITASCOPE language is capable of describing dynamic construction scenarios by properly illustrating all common construction tasks. VITASCOPE is a straight-line language (Appel 1997). In other words, VITASCOPE language statements that describe a dynamic visualization are only processed sequentially. The language needs no flow control (i.e., while or for loops) because that is achieved by the intended generating processes (e.g., a simulation model while it runs).

We implemented the designed 3D animation language in a corresponding VE application. The VITASCOPE application sequentially 1) Parses and interprets individual animation language statements, 2) Executes appropriate computer graphics algorithms in response
to the interpreted statements, and 3) Manages the dynamically evolving CAD database to represent the communicated operation dynamically in a 3D virtual world. To communicate a dynamic visualization, external processes use VITASCOPE’s parametric language statements to 1) Control the simulation time and speed, 2) Define multiple motion trajectories that constitute resource movement paths, 3) Create scene objects (terrain, equipment, materials, machines etc.) by instantiating pre-created CAD models, 4) Define behavioral properties for instantiated scene objects, and 5) Manipulate scene objects to describe their dynamic behavior.

Figure 1 presents a schema that describes the relationships between the VITASCOPE language, the corresponding VE application, computer graphics facilities (e.g. algorithms, data structures etc.), and external software and hardware authoring processes. External authoring processes (e.g. simulation models, hardware controls, and real-time positional data streams) are individually and/or collectively capable of generating information that is required for and sufficient to describe dynamic operations level construction scenarios.

The conceptual mechanics of how VITASCOPE automatically converts quantitative information communicated by external processes into dynamic smooth continuous motion are relatively straightforward. As presented in figure 1, The VITASCOPE VE application reads and interprets animation statements (in the syntax of the VITASCOPE language) that describe, chronologically (and sequentially), the things that happen in an operation. In the current VITASCOPE implementation, the animation statements representing static and dynamic events must be pre-recorded sequentially in an ASCII text file (hereinafter referred to as the trace file).

The events contained in the trace file can be communicated (in the syntax of the VITASCOPE language) by any external process capable of generating formatted text output as it runs. Such an external process (e.g. a discrete-event process simulation model) can be instrumented (i.e. programmed) to dynamically write the animation trace file while it runs. The communicated VITASCOPE animation statements can refer to 3D CAD objects created with other tools for this purpose.
VITASCOPE can read CAD files in many different file formats including VRML. By using the existing CAD models and interpreting the animation instructions communicated by external processes, VITASCOPE recreates a virtual world representation of the described operations. This is achieved by invoking appropriate computer graphics algorithms and routines and applying them to manipulate appropriate CAD models after instantiating them in a 3D virtual world.
VITASCOPE specifies an open description to describe dynamic operations and implements an interpreter that invokes appropriate computer graphics facilities to represent the described operations in a smooth, continuous 3D virtual world. By doing so, VITASCOPE introduces a tangible interface or layer of abstraction that effectively separates dynamic operations visualizations from the processes that can generate them. This has several distinct advantages and opens many opportunities.

For example, loose coupling between process simulation systems and the virtual world environment (i.e. the VITASCOPE application) allows 3D visualizations of operations to be generated from any end-user programmable simulation tool. On the other hand, tight coupling between the VITASCOPE visualization engine and a particular simulation system would compel model developers who desire to visualize their models (created using their system of choice) to learn and use a different simulation tool than the one they are proficient with. In addition, an open and loosely coupled methodology suggests that visualizations need not necessarily be generated by mathematical simulation models. They could also be generated by a wide variety of other external processes.

For instance, as figure 1 also presents, real-time data streams (e.g. GPS positional data) could be translated into the syntax of the VITASCOPE language in real-time. This translated stream can then be fed to the VITASCOPE visualization engine to visualize ongoing operations in real-time. In addition, we postulate that the enabling and large scale deployment of immersive, interactive construction virtual environments (discussed in section 4) is directly dependent on, or is greatly facilitated by, enabling a loosely coupled, general-purpose visualization methodology such as that adopted by VITASCOPE.

7. The VITASCOPE Language

The VITASCOPE language is a set of 45 parametric statements that together allow external processes to communicate sequential, time-stamped static and dynamic events. These events can describe a smooth and continuous operation of any length and complexity. The communicated events are then interpreted by VITASCOPE’s
visualization engine. The engine then invokes appropriate data structures, algorithms, and routines to manipulate CAD models and other 3D geometric primitives to present a smooth accurate virtual world representation of the operations communicated by the external processes.

Appendix A lists the core VITASCOPE language statements. The statements are grouped according to their functionality as described below. VITASCOPE’s parametric language statements are divided into the following functional groups:

### 7.1 System Statements

The major statement in this group (TIME) allows external processes to control the simulation time and specify the instants at which specific events take place (if static) or start (if dynamic). The VIEWRATIO statement allows the animator to interactively control the speed (i.e. viewing ratio) of the visualization, while TIMEJUMP permits instantaneous rewind/fast forward to specific points of interest in a pre-recorded visualization (i.e. a trace file). LOADADDON allows the VITASCOPE VE application to interactively load computer modules that implement extensions to the core VITASCOPE language (See Chapter 3 for details). The final statement, SCHEDULE, in this central group complements the TIME statement and allows the execution of other VITASCOPE animation statements to be scheduled at future animation time instants in an animation sequence.

### 7.2 Scene Construction Statements

Statements in this group allow external processes to describe the entities (terrain, equipment, materials, machines etc.) that constitute the virtual construction site. This is done by referencing CAD models of relevant resources (e.g. equipment), instantiating (or destroying) multiple specific CAD objects, assembling (or disassembling) CAD objects into logical geometric hierarchies, and placing and orienting objects in the desired state on the virtual construction site. This group also contains statements that are used to define multiple motion paths (i.e. 3D trajectories) that entities will travel on while performing operations.
7.3 Property-Setting Statements

This group of VITASCOPE statements allows external processes to specify the dynamic behavioral properties of virtual entities that have already been instantiated. An example of such a property is the fore clearance of an object. This property specifies the minimum distance that a trailing object must maintain from another leading object when both are traveling on the same motion trajectory. The rear guide point (RGP) is similarly another property that defines an object’s behavior as it travels along curves in a motion path. Statements in this group also allow dynamic strings of text and/or numeric data (e.g. resource properties, operation state variables etc.) to be displayed and dynamically updated in visualizations. To do this, VITASCOPE provides statements that can be used to instantiate text strings, attach them to specific scene objects if necessary, and dynamically update the strings as visualization progresses.

7.4 Dynamic Statements

This group of statements constitutes the core of the VITASCOPE language. The group consists of several statements that external processes can use to dynamically manipulate instantiated scene objects to depict the performance of a smooth and continuous operation. Statements in this group describe dynamic geometric transformations of scene objects. These transformations change the positions, orientations, and scales (i.e. sizes) of instantiated objects in the 3D virtual environment to depict the accurate motions objects undergo while performing the communicated operations.

VITASCOPE’s parametric animation statements allow an external process to communicate the elemental motions involved in performing construction in a geometric transformation level parlance. VITASCOPE’s primary motion-describing statements (e.g. MOVE, ROTATE, SCALE) each describe a single elemental motion that a construction resource undergoes as it performs work (e.g. A truck MOVEs along a haul road, the cab of a backhoe ROTATEs as it swings, a crane’s cable SCALEs as the hook is dropped or raised). A time-stamped sequence of an arbitrary number of such elemental motions communicated by an external process is then post-processed to describe a smooth, continuous 3D rendition of the pertinent construction operation.
7.5 View Manipulation Statements

The last group of VITASCOPE statements allows external processes to programmatically manipulate the position and the orientation of the viewer while in the virtual environment. Statements in this group also allow the user’s viewpoint to be attached to (and detached from) a dynamic entity in the visualization. This, for example allows the viewer to ride in the cab of a moving truck or an operating crane. These statements are complemented by the VITASCOPE application’s user interface that allows users to navigate to any position and orientation on the virtual jobsite using keyboard keys and/or a mouse to steer.

In appendix A, statement parameters enclosed by < and > signs (e.g. <parameter>) indicate that a relevant numerical value or a text argument is required when feeding the particular statement to the visualization engine. In general, VITASCOPE language statements are quite readable. In addition, describing each statement and its functionality individually is beyond the scope of this paper. That information is presented in detail in appendix B.

VITASCOPE’s parametric language statements are intended to be simple as well as sufficient. Simplicity, in this context, means that the statements of the language and the information sought by the statement’s parameters are both within the authoring capabilities of intended external authoring interfaces such as process simulation systems. The statements are also designed to be sufficient or semantically rich so that the succinct parametric language constructs can encapsulate all information required to describe the performance of construction operations in a smooth and continuous manner. Balancing simplicity and sufficiency (which are conflicting objectives) was a major challenge that was carefully addressed in designing the VITASCOPE language.
8. Designing VITASCOPE

8.1 The Appropriate Taxonomy

The taxonomy of the construction industry covers a wide range of interests, each appropriate to different people looking at different problems. Any problem in construction requires examination at the appropriate level of detail. In order to identify the appropriate level, the taxonomy of construction must be carefully studied and analyzed from the perspective of the problem that is addressed.

Conceptually, each entity (equipment and human craftsmen) on a virtual construction site can be considered to be equivalent to a robot performing a certain construction task. Just as robots perform construction operations on actual construction sites using real resources, virtual equipment and craftsmen perform construction on the computer screen using virtual resources. The analogy between the two arises from the fact that both robots and virtual entities do not have any intrinsic knowledge about performing construction and need to be instructed (i.e. programmed) to carry out particular construction sequences. In terms of information needs, therefore, virtual construction equipment and human craftsmen are quite similar to construction robots.

We investigated previous works aimed at classifying construction by studying different hierarchical taxonomies developed by various researchers. Many of these taxonomies were developed from interests rooted in automation and robotics (Bernold et. al 1990, Everett and Slocum 1994). Our objective, however, in studying these works was to identify an appropriate level of classifying construction in order to address the problem of designing a description to depict construction operations in 3D virtual worlds.

Both works cited in the paragraph above suggest that classifying construction operations based on basic work tasks (e.g. cut, place, position, connect etc.) is most suitable as far as concentrating automation efforts in construction is concerned. Everett and Slocum (1994) also present a set of 12 basic tasks that are mutually exclusive and collectively encompass all on-site construction operations. The literature also suggests that depending on the
nature and the level-of detail of the problem being analyzed, each basic work task can be further broken down into a set of elemental motions (e.g. reach, grasp, put etc.).

### 8.2 The Geometric Transformation Level

Communication with virtual construction entities (e.g. virtual pieces of equipment) can only be achieved using a computer interpretable vocabulary. A virtual piece of equipment cannot be directly told to perform a basic construction task. In fact, such an entity cannot directly be told to perform even elemental motions. In order to communicate instructions in computer interpretable vocabulary, the elemental motion level must be further broken down into geometric transformations such as rotations and translations.

Bernold et al. (1990) implicitly suggest that the geometric transformation level is the next logical level of decomposition (after elemental motions) in the hierarchy of construction field operations. Just as each basic task is comprised of a set of elemental motions, every elemental motion can be broken down into a set of geometric transformations. We concluded that in the context of generally programming virtual construction performers, geometric transformations must be the basic building blocks.

Consider the example of an excavator scooping dirt. Digging can be defined as a basic task (Everett and Slocum 1994). Table 1 breaks down the basic task into a set of elemental motions, which are in turn broken down into geometric transformations. The geometric transformations in the final column are described in the exact syntax of the VITASCOPE language. For instance, the first statement in the third column of table 1 instructs the boom of a virtual excavator to rotate in the vertical plane (indicated by VERT) by an amount of 27 degrees in the clockwise direction (indicated by the negative sign) in 12 time units (seconds in this case). The statement in the fourth row similarly instructs the cab of the virtual excavator to rotate in the horizontal plane (indicated by HOR) by an amount of 90 degrees in the anticlockwise direction in 16 seconds. We can communicate construction operations to virtual pieces of equipment only at this geometric transformation level.
The motions of most construction equipment and craftsmen can ultimately be broken down into rotations, translations, and other geometric transformations. This is the only level at which instructions to perform construction sequences can be directly communicated to virtual pieces of equipment and craftsmen. The basic dynamic constructs of the VITASCOPE language are therefore designed at the geometric transformation level.

### 8.3 Parameterization

Many parts of construction work have been identified as repetitive and cyclic. However, while each construction work cycle might be similar to its predecessor and successor, the cycles are not identical (Everett and Slocum 1994). The basic tasks comprising work cycles, although structurally similar, are different in amplitude and direction from their predecessors and successors.

<table>
<thead>
<tr>
<th>Basic Task</th>
<th>Elemental Motions</th>
<th>Geometric Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dig Dirt</td>
<td>Lower Empty</td>
<td>ROTATE Boom VERT -27 12; ROTATE Stick VERT -10 9;</td>
</tr>
<tr>
<td></td>
<td>Scoop Dirt</td>
<td>ROTATE Bucket VERT -90 12;</td>
</tr>
<tr>
<td></td>
<td>Lift Loaded</td>
<td>ROTATE Boom VERT 27 16; ROTATE Stick VERT 10 12;</td>
</tr>
<tr>
<td></td>
<td>Swing Loaded</td>
<td>ROTATE Cab HOR 90 16;</td>
</tr>
<tr>
<td></td>
<td>Lower Loaded</td>
<td>ROTATE Boom VERT -35 10; ROTATE Stick VERT 20 8;</td>
</tr>
<tr>
<td></td>
<td>Dump</td>
<td>ROTATE Bucket VERT 90 6;</td>
</tr>
<tr>
<td></td>
<td>Lift Empty</td>
<td>ROTATE Stick VERT -20 12; ROTATE Boom VERT 35 9;</td>
</tr>
<tr>
<td></td>
<td>Swing Empty</td>
<td>ROTATE Cab HOR -90 14;</td>
</tr>
<tr>
<td></td>
<td>Lower Empty</td>
<td>ROTATE Boom VERT -27 12; ROTATE Stick VERT -10 9;</td>
</tr>
</tbody>
</table>
For example, the excavator scooping dirt goes through the same set of motions in loading a truck i.e. dig, lift, swing loaded, dump, swing empty, and dig again. However, the amplitude of these motions may vary for each pass of the excavator. This could either be due to the increasing depth of the hole that is being dug, or the repositioning of the excavator as it performs work, or the fact that each successive empty truck does not stop at exactly the same spot.

Therefore, although the performance of construction can ultimately be broken down into geometric transformations, their amplitudes are not constant and cannot be predetermined. Complexity further increases with the increase in the degrees of freedom each piece of equipment or a human craftsman has and exercises while performing construction.

VITASCOPE addresses this problem by specifying parameterized instructions. In table 1, the elemental motions involved in performing the basic task of digging have been decomposed into basic geometric transformations (i.e. rotations). However, the direction of rotation (i.e. plane of rotation) is different for each elemental motion. For example, the cab of an excavator rotates on a plane that is parallel to the surface on which the excavator rests. The boom, stick, and the bucket, however, rotate along a plane that is perpendicular to the resting surface and parallel to the forward direction of the cab. In addition, the amplitude of rotations during each elemental motion differs as the excavator adjusts to the deforming terrain and due to other factors.

In VITASCOPE, this issue is addressed by designing an ability to specify parametric instructions to virtual pieces of equipment and craftsmen. For example, in the third column of table 1, the third and fourth arguments in the rotation instructions specify the plane and the amplitude of rotation for each of the geometric transformations that comprise the elemental motions. The fifth argument specifies the time required for the particular instant of the transformation and will obviously be different each time. In general, all numerical parameters are typically described by sampling values from probability distributions.
9. The VITASCOPE Virtual Environment Application

The VITASCOPE application is the VE implementation that interprets and processes sequential instructions (in the syntax of the VITASCOPE language) to recreate a 3D virtual world representation of operations communicated by external processes. In order to facilitate communication between external authoring processes and computer graphics facilities, the application implements three key technologies: 1) The conversion of discrete animation information (i.e. parametric VITASCOPE language statements) into smooth motion; 2) The spatial organization and rendering of multiple dynamic 3D CAD objects; and 3) The efficient sequencing and timing of frames such that the ratio of viewing to simulated time is constant.

The VITASCOPE VE application requires a graphical database of 3D scene objects that must be created, manipulated, and maintained in order to depict animation. Scene graph architectures are effective for organizing such databases and are well supported by several industrial-strength commercial libraries (Kamat and Martinez 2002). The dynamic maintenance of scene graphs needed to represent virtual construction worlds that are constantly evolving, however, requires the development of algorithms specific to the application. The VITASCOPE VE application creates, manipulates, and manages such scene graph based databases to animate communicated construction operations. The scene graph based data structures and algorithms designed in implementing the VITASCOPE VE application are interesting research results that are discussed in detail in (Kamat and Martinez 2002).

10. Validation

We validated the effectiveness of the designed specification (i.e. the VITASCOPE language) and its implementation (i.e. the VITASCOPE application) in facilitating automatic, rapid generation of operations-level virtual construction scenarios communicated by external processes. To do this, we instrumented discrete-event process simulation models to automatically generate animation trace files (in the VITASCOPE language) as they run. The instrumented models were then executed to produce the
required animation trace files. A simulation model typically executes in only a few seconds even though the animation trace file it produces is typically several thousand lines long and may describe a VITASCOPE visualization spanning minutes, hours, or even years.

We then post-processed the trace files in VITASCOPE’s VE application to investigate the extent to which the simulated (and communicated) operations can be recreated in 3D virtual worlds. The VITASCOPE application was able to accurately depict graphical 3D representations of the communicated operations. The degree of accuracy was faithful to the amount of detail communicated by the driving process (i.e. the discrete event simulation models in this case). In order to generate alternate virtual scenarios of each operation, we then simply manipulate the quantitative and/or logical simulation parameters of the driving process model.

These parameters include the number and type of resources (e.g. number of trucks in an earthmoving operation, type of crane to use in steel erection, space for temporary storage of materials etc.); the rules under which the different tasks that compose the operations are performed; and different random variates to describe the durations of individual tasks. The modified model is then promptly rerun to produce an alternate virtual world scenario that can then be processed and visualized by VITASCOPE for evaluation. This procedure of instrumenting a discrete-event process model and manipulating it’s parameters to generate a VITASCOPE virtual world scenario is described in detail with the help of a worked example in (Kamat and Martinez 2001).

Figure 2 presents snapshots of some construction operations that were modeled, communicated, and visualized as part of VITASCOPE’s validation exercise. The types of construction operations that have been visualized include earthmoving, block masonry, delivery and placement of concrete, and steel erection.
Appendix D presents a short portion of the VITASCOPE animation trace file that describes the concrete delivery and pouring operation presented in figure 2(c). The entire VITASCOPE animation trace files and the driving discrete-event process models are both significantly large to include in this dissertation even as appendices. However, complete animation trace files and the driving simulation process models are available for download and inspection from this paper’s accompanying website (http://strobos.cee.vt.edu/vitascope/Dissertation). Files of both types can be inspected by opening them in any formatted text editor. The accompanying website also hosts demonstration movies of each of the VITASCOPE visualizations whose snapshot is presented in figure 2.
11. Extensibility

As the animation trace files and the corresponding videos prove, most elemental motions involved in performing common construction operations can be ultimately broken down into parametric geometric transformations. However, the number of such transformations required for even a simple set of elemental motions can be very large. For example, as presented in Table 1, communicating just one pass of an excavator that has only two possible planes and four axes of rotation, requires the specification of 14 geometric transformations. It is left to the reader to imagine how many geometric transformations might be required to realistically depict a human craftsman having several joints (axes of rotation) and infinite degrees of freedom (planes of rotation and translation).

We envision that elemental motions and even basic tasks (in some cases) can be communicated to virtual construction resources using higher-order language constructs. These higher-order constructs can concatenate basic geometric transformations such as rotations and translations to describe elemental motions or even basic tasks. The parametric components will still remain, although their number and the values they represent may indicate factors other than those required in specifying basic geometric transformations. The VITASCOPE language and the corresponding VE application are therefore designed to be extensible and scalable. Researchers and engineers can design extensions to the VITASCOPE language and implement them without having to understand or modify the existing implementation (See Chapter 3 for details).

Revisiting the digging excavator example discussed in section 8.2, it can be possible to describe the elemental motions involved in a pass using just one parametric statement as described in Figure 3. The language construct EXCAVATORPASS is described as being comprised of various geometric transformations (i.e. rotations). The parameters required in communicating the pass to a virtual excavator are now the location of the dirt and the position of the empty truck and not the values and planes of individual rotations. Figure 3 is an oversimplified version provided to clearly present the concept of scalability and extensibility. A true higher-order definition for such a pass will obviously also include a
representation of time both inside the function body as well as in the list of parameters (See Chapter 5 for details).

EXCAVATORPASS (DirtLocation, TruckLocation)
{
    ROTATE Boom VERT -27;
    ROTATE Stick VERT -10;
    ROTATE Bucket VERT -90;
    ROTATE Boom VERT 27;
    ROTATE Stick VERT 10;
    ROTATE Cab HOR 90;
    ROTATE Boom VERT -35;
    ROTATE Stick VERT 20;
    ROTATE Bucket VERT 90;
    ROTATE Stick VERT -20;
    ROTATE Boom VERT 35;
    ROTATE Cab HOR -90;
    ROTATE Boom VERT -27;
    ROTATE Stick VERT -10;
}

Figure 3: Designing Higher-Level Language Constructs

Combinations of elementary geometric transformations could similarly be used to describe other elemental motions or even basic tasks if they can be encapsulated by a finite number of parameters. Describing the performance of all common construction operations, especially the accurate motions of equipment and human craftsmen, using only basic geometric transformations is extremely cumbersome. On the other hand, designing higher-level constructs to communicate performance of all elemental motions and basic tasks is challenging, arduous, time-consuming, and beyond the capabilities of a single researcher. Extensibility and scalability of the described nature is therefore essential if the VITASCOPE language is to evolve substantially over time and through the collective efforts of many research scientists.

12. Future Work

Statements in the VITASCOPE language describe the performance of a construction process as a concatenation of elemental motions in geometric transformation level dialect. This parlance is advantageous for its generality and flexibility. However, it is
tedious and often impossible to realistically describe the motions of certain, highly articulated construction resources using this low-level vocabulary. For instance, describing the realistic motions construction workers undergo as they perform work is a challenging proposition using VITASCOPE’s elemental motion based language. Additional research is necessary to design and implement higher level animation methods that can succinctly describe the complex motions of articulated resources in a finite number of parametric text statements.

VITASCOPE’s efficacy has been validated using discrete-event simulation models as external authoring interfaces in post-processed mode. While this exercise confirms the simplicity and expressiveness of the designed animation language, the ability to generate a smooth, continuous, 3D animation using other software and hardware authoring interfaces needs to be investigated. For instance, exploring the possibility of using real-time data streams (e.g. from GPS receivers mounted on equipment pieces) to describe a virtual world representing the current status of a jobsite presents an interesting research initiative. In addition, integration of simulation models with hardware controls to describe interactive virtual construction worlds requires the investigation of several research issues and calls for some interesting work.

13. Conclusion

In order to effectively apply virtual reality technologies to construction operations design, we must be able to rapidly generate alternate operations-level virtual world scenarios for comparison, evaluation, and “what-if” analyses. To do this, different external processes capable of describing construction operations must be able to interact (often simultaneously) with 3D virtual environments through a common description. The interface that such a description defines must effectively separate visualizations from the processes that generate them.

A general description to communicate dynamic construction operations of any length and complexity must be carefully designed at the correct level of abstraction. Previous works identify the basic task as the fundamental building block of all construction field
operations. This study observed that the basic task level, and indeed the next elemental motion level, are both too general as far as communicating construction sequences to virtual pieces of equipment and human craftsmen is concerned. A taxonomical level based on basic geometric transformations is therefore identified as appropriate for the design of such a description and subsequently used in designing the VITASCOPE 3D animation language.

Validation of the designed VITASCOPE language effectively proves that the geometric transformation level is not only practical in allowing external processes to communicate dynamic construction operations, but also quite effective. In addition, by post-processing simulation model-generated trace files to recreate dynamic construction operations, we are able to conclude that the designed interface effectively separates and entirely decouples the virtual world (i.e. the visualization engine) from the processes that generate dynamic scenarios. The validation exercise also confirms that VITASCOPE facilitates rapid generation and visualization of alternate operations level construction scenarios. This is achieved by changing decision variables in discrete-event process models and communicating the operations to the VE implementation in the language’s syntax.

14. References


Chapter 3
Extensible and Scalable 3D Visualization of Construction Processes and Products

1. Introduction

Visualization of simulated operations can be of significant help in the verification and validation of discrete-event process models (Law and Kelton 2000). This is especially true in construction where typical decision makers are experts in their domain but are not generally proficient in simulation itself. Visualization can also provide decision makers with valuable insight into subtleties of planned construction operations that are otherwise non-quantifiable and non-presentable. The necessity to effectively communicate modeled construction processes and the resulting evolving products is the motivation behind ongoing visualization research efforts at Virginia Tech. These efforts focus on designing automatic, process simulation-driven methods to visualize construction processes and evolving products in dynamic 3D virtual worlds.

The tangible outcome of this ongoing research is the VITASCOPE visualization system. VITASCOPE is an acronym for VIsualizaTion of Simulated Construction OPErations. VITASCOPE is a 3D animation language designed specifically for visualizing modeled construction operations in smooth, continuous, dynamic 3D virtual worlds. A limited subset of the VITASCOPE language and the corresponding prototype implementation were referred to as the Dynamic Construction Visualizer (DCV) in some prior publications (Kamat and Martinez 2001, 2002).

1.1 Main Contribution

In this paper, we present an extensible and scalable framework for the 3D visualization of construction processes and evolving products. We capitalize on documented principles of application framework design to devise and implement an extensible framework for 1) extending engineers’ ability to visualize simulated construction processes, and for 2)
developing higher-level construction process visualization tools. This framework is implemented as an extension (add-on) interface to the VITASCOPE visualization system and its core 3D animation facilities.

Engineers and scientists can use the framework’s interface and VITASCOPE’s core facilities to design and implement custom 3D animation statements and functions that extend the VITASCOPE language and engineers’ ability to visualize simulated construction processes. Such extensions reside in dynamically shared runtime libraries and can be designed and implemented without having to modify or understand VITASCOPE’s core design and implementation. Extensions that contain one or more animation statements and/or functions are considered as add-ons to the VITASCOPE visualization system. Researchers and end-users can design add-ons for use with specific process visualizations (e.g. statements that describe the motion of a unique piece of equipment), or they can design general extensions that become part of the VITASCOPE animation language and can subsequently be used in any visualization (e.g. statements that facilitate the description of a new construction method).

The work thus contributes to the computing infrastructure for construction research by allowing researchers to conceive and implement solutions to higher-level construction process visualization problems. This allows attention to be concentrated and valuable research resources to be spent on advancing the state-of-the–art rather than on repeatedly designing and implementing foundation-level visualization support as would be the case otherwise. Visualization of constructions operations is a very broad and complex domain. Similar to other aspects of knowledge, the advancement of simulation-driven construction process visualization technology requires the collective efforts and skills of many different researchers such that it incrementally evolves per the scientific method. The presented work recognizes these facts and attempts to put in place the infrastructure that will facilitate synergy among various researchers to rapidly advance the field.
2. Research Motivation

The current state-of-the-art in scientific, simulation-driven construction process visualization allows engineers to visualize several common construction processes in 3D. Engineers can use the VITASCOPE language and instrument simulation models to instantiate CAD objects in virtual environments and then to apply basic geometric transformations (translation, rotation, scaling) to those objects to describe their dynamic behavior and collectively depict a construction operation (Kamat and Martinez 2001).

Using these elementary geometric transformation-based semantics, engineers can visualize construction operations in which resources can be assumed to be moving uniformly at constant speeds, operations that do not require any sort of CAD model deformation (i.e. operations involving rigid body dynamics only), operations that do not involve unstructured or flowing resources (e.g. concrete), and operations in which equipment and crew always move on fixed, well-defined piecewise linear paths.

The technology that enables simulation-driven 3D construction operations visualization is still relatively nascent and much work needs to be done before it can be fully exploited. Notwithstanding recent advancements, it is obvious from the above discussion that current simulation-driven visualization technology enables engineers to describe and depict only a moderate subset of all possible construction processes. A lot of work needs to be done before the body of knowledge provides engineers access to automated, simulation-driven methods to visually describe the entire spectrum of construction processes in realistic, smooth, continuous, and data-augmented 3D virtual worlds.

In addition, core construction technology is a perennially evolving science. New methods of construction, advanced pieces of construction equipment, and new materials and tools are continually being developed and adopted on construction sites. Several of these advancements are unique to a single project or company. Others are universally adopted across the entire construction industry. On another front, advances in computer graphics, hardware technology, and algorithm design allow engineers and scientists to visualize data and phenomena that were hitherto impossible to describe in 3D. The fallout of these
facts is that in general, there will always be construction processes that are more complex than those that can be visually described using existing techniques. Simulation-driven construction process visualization technology must thus keep pace with advances in both these areas to be continually relevant and useful.

Furthermore, we strongly believe that simulation-driven operations visualization technology will be the basis for advances that will enable virtual and immersive construction environments with process-level detail to be commonplace. In these virtual environments, objects under the control of simulation models will be aware of, and react to, humans and human controlled machines. The knowledge required to enable such environments is directly dependent on, or is greatly facilitated by advances in simulation-driven operations visualization technology, if the latter is based on a loosely coupled, general-purpose methodology. In addition to being independent of any particular simulation tool, an open and loosely coupled visualization methodology such as VITASCOPE’s can allow visualizations to be generated by software interfaces other than simulation models. Real-time data, for instance, could be fed to the visualization engine to visualize operations in real-time.

This discussion highlights the abundant open research questions and issues that need to be addressed to allow engineers to fully exploit operations visualization in construction operations planning and design. Many of the issues are urgent and warrant immediate attention. Others are not apparent at this time but will be as the technology in operations visualization, construction, and computer graphics evolves.

Exclusively advancing the state-of-the-art in simulation-driven construction operations visualization and addressing all open research questions apparent at this time and those that will surface in the near future is beyond the temporal and intellectual capabilities of a single researcher or research team. The domain of construction operations visualization is so vast and intricate that researchers can base their entire careers on the subject and still have abundant unanswered research questions at retirement. Similar to other aspects of knowledge, the advancement of simulation-driven construction visualization technology
requires the collective efforts and skills of many different researchers so that it incrementally evolves per the scientific method.

The current computing infrastructure for research in simulated construction operations visualization is however not conducive to the incremental advancement of the field. Engineers who wish to utilize operations visualization to communicate unique ideas or solve higher-level problems are constrained by the limitations in the state-of-the-art. Secondly, researchers who intend to explore advanced issues in simulated operations visualization to extend the technology must first repeatedly reinvent methods to support foundation-level dynamic animation. This precludes incremental scientific evolution of the field. In addition, this discourages and/or restricts engineers from using simulated construction process visualization technologies to make such important discoveries that may dramatically reduce the life-cycle costs of construction.

The presented work is strongly influenced by the above-described state of affairs and attempts to contribute towards alleviating the situation by laying the groundwork in simulation-driven construction operations visualization technology. The work, we expect, will allow scientists and researchers to leverage their skills and backgrounds to focus their efforts towards making significant incremental contributions that rapidly advance the state-of-the-art. We also expect the work to put in place the computing infrastructure that allows engineers to readily use operations visualization to solve higher-level relevant problems and to communicate their ideas.

3. Extensible and Scalable Framework

Whenever there is a family of applications that all solve substantially similar problems, there is an opportunity for a domain application framework (Booch 1994). A framework is an organized collection of methods that provide a set of services for a particular domain. A framework provides a number of individual mechanisms that clients can use, adapt, and extend without having to modify or fully understand the implementation of the core mechanisms themselves.
Such a scheme, by its very nature, facilitates the reuse of components and supports the scientific method by abetting the incremental evolution of knowledge through the collective efforts of many scientists (Fayad et al. 1999). This provides greater leverage to researchers by allowing them to concentrate on addressing higher-level research issues rather than having to repeatedly design and implement core domain facilities. It is obvious from the earlier discussion that the domain of simulation-driven construction operations visualization warrants the existence of such an extensible framework.

### 3.1 Challenges

Designing effective and efficient end-user mechanisms for extending automatic, simulation-driven methods for 3D visualization of construction processes and products presents numeric interesting challenges. The work involves the delicate balancing of and prudent trade-offs between competing groups of technical requirements (i.e. completeness, efficiency, and simplicity versus adaptability and flexibility).

#### 3.1.1 Extensibility

Extensibility of a 3D animation language for visualizing simulated operations implies that other researchers can seamlessly extend its capabilities. An extensible framework for visualizing simulated construction operations thus involves identifying and designing core, reusable animation methods and implementing them in a well-engineered environment that is adaptable, flexible, and easy to extend. Similar to other well-designed frameworks, the environment must be functionally complete, efficient, and simple.

From an implementation standpoint, the framework must present an add-on specification that allows existing animation methods to be extended seamlessly using high-level compiled languages and without the need to link statically with the framework. The specification must describe stable interfaces to core domain facilities and provide explicit hook methods that allow their extension (Pree 1995). The designed hook methods must systematically decouple the stable interfaces and behaviors of the domain from the variations required by instantiations of extension modules in particular contexts. Custom
animation constructs and extensions can then reside in dynamically shared libraries which can be accessed on demand at runtime during visualization.

### 3.1.2 Scalability

In any extensible framework, scalability is of the utmost importance. Scalability of a framework is the measure of its implementation’s ability to retain performance levels when additional features and functionality are added to the framework (i.e. it is extended). In terms of the implementation’s context, it is the ability to not only function well in the rescaled situation, but to actually take full advantage of it. From both standpoints, the rescaling is typically to a larger size or volume.

In the case of the animation language to visualize simulated construction operations, its implementation would be termed scalable if performance levels in a particular context (e.g. a particular hardware configuration) are maintained as more features and language constructs are designed and implemented. Similarly, in terms of the context, the language and its implementation would be scalable if they could be moved to a more powerful computer and took full advantage of the better system in terms of performance.

### 3.1.3 Portability

Another important challenge in designing and implementing an extensible application framework is making the core framework facilities and their implementation portable. In the context of the extensible language to visualize modeled construction processes, portability refers to the ability of the language and its implementation to be compatible with multiple hardware (and software) platforms ranging from high-end graphics workstations to commonly used desktops and laptops. This is of significance if the technology to visualize simulated construction processes is to be readily ported across a wide range of hardware platforms and display devices. In addition, portability ensures that researchers’ choices of computing platforms do not affect their ability and willingness to contribute to the methodology.
3.2 Desiderata

In addition to addressing the abovementioned challenges and satisfying required criteria, an extensible framework for visualizing simulated construction processes and the resulting products must conform to the following litmus tests of framework extensibility:

- **Concealed Complexity** – Researchers wishing to extend the visualization methodology must be able to design and implement new animation constructs without having to modify or understand the design and implementation of the underlying core animation methods. Designing such an easy-to-learn framework is a challenging prospect since it requires the definition of an interface and hooks that anticipate a wider, unforeseeable range of potential use cases (Huni 1995).

- **Uniform Client Interface** – Engineers utilizing the methodology to visualize modeled construction processes must feel no difference in the usage of core and extended animation methods. This implies that designed extensions must present the same interface to the end-user as do the core animation methods.

The issues we address in this work are thus threefold. First, we identify, design, and implement a set of statements that constitute the core, minimal animation language and are of the most general use from the extension viewpoint. Second, we investigate methods to design an add-on interface to the implemented animation methods. Finally, we validate the add-on interface and its scalability by implementing the framework on multiple hardware platforms and extending the core animation language via the designed interface.

4. Technical Approach

4.1 Identifying Foundational Abstractions

The first and most important research issue in the design of any extensible application framework is the identification and implementation of the foundational set of domain facilities (Biggerstaff and Perlis 1989). Such facilities constitute the core functionality for the domain as well as define the optimum set of building blocks from the reuse and extension point of view. In the case of the language to visualize modeled construction
processes, the elementary set of language constructs must consist of statements that 1) are collectively sufficient to visually describe a broad class of common construction processes, and 2) are at such a level of abstraction that they can be logically concatenated and/or used in combination to describe higher-level motion dynamics involved in performing construction.

These issues were carefully considered in designing the core statements of the VITASCOPE language. The first step in the direction was to identify an appropriate level of classifying construction to facilitate the logical design of the core language facilities. After researching several works on the taxonomy of field construction processes, we identified elementary geometric transformations as the only level at which instructions to perform construction sequences can be directly communicated to virtual pieces of equipment and craftsmen (see Chapter 2 for details). That work concluded that parametric geometric transformations must constitute the basic building blocks in the context of generally describing virtual construction processes.

The above-cited work also identified geometric transformations as the appropriate fundamental building blocks that can be logically concatenated to describe elemental motions. The elemental motions can in turn together be used to describe more elaborate construction tasks at higher levels in the hierarchical taxonomy of construction processes. Thus, in addition to being collectively sufficient to visually describe a broad class of common construction processes, animation constructs that describe elementary geometric transformations constitute a potent set of building blocks that are of the most general use from the extension viewpoint.

4.2 Designing an Extensible Interface

The next research issue that we addressed after identifying and implementing the minimal set of foundational abstractions is the design of the extension interface to those abstractions. In the case of the extensible framework to visualize construction processes, this involved the design of semantics that provide access to the core animation statements while satisfying the desiderata identified in the earlier sections.
Figure 1 presents the schema that describes an open, loosely-coupled methodology to visualize simulated construction processes. This schema, adopted in designing VITASCOPE, describes the relationships between the animation language, the corresponding virtual environment application, computer graphics facilities (e.g. algorithms, data structures etc.), and external authoring processes (i.e. simulation models).

The conceptual mechanics of how this schema allows automatic conversion of quantitative information communicated by simulation models into dynamic smooth continuous motion are relatively straightforward. Simulation models generate discrete process-level events that can be used to describe dynamic operations-level construction scenarios (Kamat and Martinez 2001). As figure 1 presents, discrete simulation events communicate a dynamic construction operation using animation statements to describe
chronologically (and sequentially), the things that happen in that operation. Such communicated statements can contain references to 3D CAD models created with other tools for this purpose.

Each of the communicated animation statements is interpreted by the implementation’s interpreter and appropriate computer graphics algorithms are invoked to visually depict the operation. The invoked algorithms and routines manipulate appropriate CAD models that represent construction resources after instantiating them in a 3D virtual world. For each statement that comprises the 3D animation language, the implementation’s interpreter provides code to decipher the statement’s parameters and convert them to a format that can be communicated to the underlying computer graphics algorithms and routines.

In order to make this schema extensible then, we designed methods that allow researchers to seamlessly augment the interpreter’s vocabulary without having to modify or fully understand the mechanics of the existing interpreter components. Moreover, while core foundational statements directly invoke computer graphics algorithms, language statements designed as extensions can choose to invoke concatenated sequences of the foundational statements to describe construction processes in a vocabulary that is at a higher level than elementary geometric transformations.

This approach, described schematically in figure 2, addresses the challenges and satisfies the desiderata outlined earlier. Researchers can design new animation statements and provide code to interpret those statements (represented as additional jigsaw puzzle pieces in figure 2) without having to modify or understand the code that comprises existing interpreter components. When simulation models communicate events using the new designed statements, the implementation’s interpreter automatically invokes the appropriate segment of code.
Such extension code describing one or more new statement can reside in a completely separate physical module on the computer and can be dynamically loaded at runtime as required during visualization. While shielding researchers intending to extend the language from undesired complexity, the adopted approach helps present a uniform user interface to client engineers i.e. engineers perceive no difference in the use of language statements that are foundational or extensions when describing process visualizations. Figure 2 makes this apparent by leaving the interface between simulation models and the interpreter unchanged from that in figure 1.

4.2.1 Achieving Smooth Animation
VITASCOPE’s animation language-based scheme for visualizing construction processes is based on converting discrete pieces of information communicated by simulation models into smooth continuous motion. While the communicated pieces of animation information (i.e. animation language statements) are processed instantaneously by the implementation’s interpreter, their effect in a smooth continuous 3D world evolves over a random, but finite period of visualization time.
For instance, consider a simulation model that communicates an event describing a dump truck (say Truck1) traversing a path (say LoadToDump) in 120 time units. The statement that contains this information will be processed (i.e. parsed and deciphered) almost instantaneously by the interpreter. The actual time-consuming event (Truck1’s journey in this case) will however unfold in an animation in the next 120 animation time units after it is processed. Thus, although statement processing is instantaneous, the resulting visualization is necessarily not. VITASCOPE’s visualization scheme achieves smooth animation by continuously monitoring objects set in motion and updating their position at frequent time instants, while at the same time processing any new communicated events (Kamat and Martinez 2002). This scheme is graphically presented in figure 3.

The introduction of an extension interface, however, presents an interesting challenge to the proper operation of this scheme. Scene objects that are set in motion in response to animation statements implemented as language extensions must also be continuously monitored and updated to describe their smooth motion during visualization. Any updating of scene objects that undergo motion in response to processed foundational statements is performed during the main computation loop presented in figure 3. In order to maintain a uniform user-interface and to avoid any discrepancies, objects undergoing motion in response to processed extension statements must also be similarly monitored and updated at the same time instants as their foundational counterparts.

In order to address this issue, we suitably augmented the object-updating process of the original computational scheme. The expanded view of the computation process box that updates moving objects, augmented to accommodate the dynamic motion updates of extension statement-moved objects, is presented in figure 4. This modified approach in effect provides each designed extension (add-on) an opportunity to update the scene objects its statements set in motion. This ensures that all scene objects are updated every cycle and move smoothly and logically regardless of whether they were set in motion by foundational animation statements or statements implemented as extensions.
A similar issue was encountered during the drawing (refreshing) of scene objects controlled by external add-ons. The process that drew each foundational statement-controlled scene object after every update was suitably modified to present each add-on with an opportunity to refresh locally controlled objects at every redrawn frame.
The following section now presents how we implemented the above-described approach to design an extension (add-on) interface to the VITASCOPE visualization system. The designed framework allows researchers and scientists to seamlessly extend VITASCOPE’s 3D animation language to enhance engineer’s ability to visualize modeled construction processes and/or design higher-level visualization tools.

5. VITASCOPE Add-On Interface

We have implemented an extensible framework to visualize modeled construction processes and evolving products as an add-on interface to the VITASCOPE 3D animation language. This interface provides clean, programmatic access to VITASCOPE’s foundational language statements and other core visualization facilities. This allows researchers and engineers to design and implement custom 3D animation statements and
functions that extend the VITASCOPE language and engineers’ ability to visualize construction processes. Such language extensions can be designed and implemented without having to understand, modify or interact with the code that implements VITASCOPE’s foundational statements or any of its other components.

Researcher’s intending to extend the language need only understand the mechanics of the add-on interface and comply with the semantic design rules it enforces. Designed language extensions can reside in dynamically shared runtime libraries that exist as separate physical modules on the computer. Such extension modules are dynamically loaded on demand during visualization each time the language interpreter encounters an extension statement and must execute the appropriate piece of code in response to that statement. VITASCOPE and its add-on interface have been implemented in ANSI C++ (Stroustrup 1999) on the MS Windows and SGI IRIX platforms. The following discussion includes simplified segments of C++ code that, from their context, should be legible even to readers not familiar with the language.

5.1 Loading Language Extensions

In order to allow engineers to use language statements defined in add-ons the same way they use foundational statements, VITASCOPE must dynamically load the physical computer modules that contain the implementation of any extension statements. VITASCOPE provides the LOADADDON statement to dynamically load physical add-on modules at visualization runtime. The syntax of this VITASCOPE statement is straightforward and consists of the name of the add-on module to be loaded preceded by the LOADADDON keyword as indicated below:

Syntax: LOADADDON AddOnName;

Example: LOADADDON KineMach;

When VITASCOPE’s interpreter encounters such a statement, it attempts to locate the physical computer module that implements the add-on indicated in the argument. In particular, VITASCOPE searches the host computer for a dynamically linked library
(DLL) named AddOnName.dll (e.g. KineMach.dll) on MS Windows or for a dynamically shared object (DSO) named libAddOnName.so (e.g. libKineMach.so) on SGI IRIX machines. This scheme conforms to the physical module naming conventions of each platform.

Upon locating the required module, VITASCOPE searches the module for a C++ function exported with the name “VitaAddOnInit”. Such a function represents the entry point to the add-on that VITASCOPE must follow to access the language statements and facilities defined in that add-on module. This extender-defined function must have a prototype similar to that presented in the code listing in figure 5.

```c
extern "C" int VitaAddOnInit(const char* szTraceFileName)
{
    // Perform initialization tasks that can call back
    // into VITASCOPE to register add-on defined
    // statements; or perform any other add-on
    // initialization tasks.

    // szTraceFileName points to a buffer that
    // contains the name of the animation trace file
    // being processed.

    // The function should return 0 if the add-on
    // could not initialize properly. Otherwise, it
    // should return a non-zero value.

    if(bFailed)
        return 0;
    else
        return 1;
}
```

**Figure 5: VITASCOPE Add-On Entry Point Function Prototype**

From within the VitaAddOnInit function, the add-on can call any of the functions exported by VITASCOPE’s add-on interface (described next). In addition to performing any required initialization tasks, an add-on should register the language statements it
defines with VITASCOPE from within this function. This is particularly important as VITASCOPE’s interpreter will otherwise not recognize them as statements when they are encountered during animation processing.

### 5.2 Appending the Interpreter’s Vocabulary

Add-on’s designed for VITASCOPE generally add new statements to the language and extend the vocabulary of its interpreter. Add-on’s could also redefine (override), if necessary, many of VITASCOPE’s foundational statements. In either case, the add-on must register its statements with VITASCOPE when the physical module they are contained in is dynamically loaded. By registering a statement with VITASCOPE, an add-on instructs the interpreter about the physical location of the computer code that must be executed when that particular statement is encountered during visualization. In addition to registering statements, an add-on must also instruct VITASCOPE on the location of the code that must be executed in order to 1) dynamically update the positions of objects set in motion by the add-on defined statements, and 2) refresh (redraw) those scene objects on the display in their updated positions.

VITASCOPE’s add-on interface provides C++ functions that client add-ons can use to register their statements as well as their update and draw routines. The prototypes of these registration functions are presented in the listing in figure 6 below.

```c++
// Register a statement with VITASCOPE.
const char* dcvRegStatement(const char* szAlias,
                            FPExtStatement pFunc, void* pD1=NULL, void* pD2=NULL);

// Register the add-on’s update function, if any.
int dcvRegUpdateFunc(FPUpdateFunc pFunc, void* pUserData=NULL);

// Register the add-on’s drawing function, if any.
int dcvRegDrawFunc(FPDrewFunc pFunc, void* pUserData=NULL);
```

**Figure 6: Function Prototypes for Registering Add-On Facilities**
When registering a statement with VITASCOPE, an add-on must supply the name of the statement (szAlias), the address of a C++ function in the add-on’s module (pFunc), and other optional add-on defined data (pD1, pD2). When VITASCOPE’s interpreter executes such a registered statement during visualization, it calls the C++ function whose address was supplied during the registration and passes back the optional add-on defined data as arguments. An add-on must similarly supply the addresses of C++ functions within the module to VITASCOPE when registering its update and draw functions if any. VITASCOPE then calls back those functions with the supplied data each time the dynamic scene is updated and refreshed respectively.

The listing in figure 7 presents a simplified (yet complete) scheme where an add-on registers its statements as well as the update and draw functions with VITASCOPE from within the add-on’s VitaAddOnInit function. This ensures that VITASCOPE’s interpreter immediately updates its lexicon with statements “DoSomething” and “DoMore” at the instant at which the physical add-on module is loaded. In addition, VITASCOPE uniformly updates and refreshes any scene objects controlled by these add-on statements.

5.3 Reusing Foundational Statements

The fundamental idea behind any extensible framework including the presented add-on interface to VITASCOPE is to facilitate the reuse of core foundational constructs in the design and implementation of higher-level domain facilities. In VITASCOPE’s case, this implies providing designed add-ons with the ability to seamlessly concatenate VITASCOPE language foundational animation statements to define higher-order methods to visualize simulated construction processes.

VITASCOPE’s add-on interface provides add-ons programmatic access to the entire VITASCOPE foundational 3D animation language. The designed add-on interface defines C++ functions that client add-ons can use to execute foundational animation statements from any routine within the add-on at any desired animation time instant. The prototypes of these functions that support foundational statement execution from within add-on modules are presented in the listing in figure 8.
extern "C" int VitaAddOnInit(const char* szTraceFileName)
{
    // Register add-on defined statements
    dcVRegStatement("DoSomething", DoSomethingFunc);
    dcVRegStatement("DoMore", DoMoreFunc);

    // Register the add-on’s update function
    dcVRegUpdateFunc(MyUpdateFunc);

    // Register the add-on’s draw function
    dcVRegDrawFunc(MyDrawFunc);

    // Do other add-on initialization
    if(bFailed)
        return 0;
    else
        return 1;
}

int DoSomethingFunc(const char* szArgs, void* p1, void* p2)
{
    // Execute code in response to the DoSomething statement
}

int DoMoreFunc(const char* szArgs, void* p1, void* p2)
{
    // Execute code in response to the DoMore statement
}

int MyUpdateFunc(void* puserData)
{
    // Update objects set in motion by this add-on’s statements
}

int MyDrawFunc(void* puserData)
{
    // Draw objects set in motion by this add-on’s statements in their updated positions
}

Figure 7: Add-On Facility-Registering Scheme
The function `dcvExecuteStatement` when called from within an add-on routine immediately executes the animation statement contained in the argument `szFullStatement`. The function `dcvScheduleStatementExecution` when called from within an add-on does not, however, immediately execute the statement contained in `szFullStatement`. Instead, the statement is scheduled for execution at a desired future animation time represented by the second argument `dEventTriggerTime`. In order to facilitate the scheduling of statement executions at future animation times, VITASCOPE’s add-on interface provides add-ons with access to the current animation time during visualization. Add-ons can access the current animation time maintained by VITASCOPE’s clock from within any routine via the function `dcvGetCurSimTime`.

The following simple example illustrates how these add-on interface-provided functions facilitate concatenation of foundational animation statements from within add-ons to provide engineers with higher-level animation constructs to visually describe simulated construction processes. Figure 9 presents an annotated VITASCOPE animation trace segment that describes the assembly, loaded travel, dumping, and empty return of a dump truck. Only foundational language statements are used in the description.

To present engineers with animation statements that describe and manipulate the same dump truck in construction parlance as opposed to geometric transformation terminology, we can design a simple VITASCOPE add-on that defines higher-level statements to

```c
// Retrieve the current simulation time from VITASCOPE.
const double& dcvGetCurSimTime();

// Immediately execute the passed statement.
int dcvExecuteStatement(const char* szFullStatement);

// Schedule the passed statement for future execution.
int dcvScheduleStatementExecution(const double dEventTriggerTime,
                            const char* szFullStatement);
```

**Figure 8: Function Prototypes for Executing Foundational Statements**

The function `dcvExecuteStatement` when called from within an add-on routine immediately executes the animation statement contained in the argument `szFullStatement`. The function `dcvScheduleStatementExecution` when called from within an add-on does not, however, immediately execute the statement contained in `szFullStatement`. Instead, the statement is scheduled for execution at a desired future animation time represented by the second argument `dEventTriggerTime`. In order to facilitate the scheduling of statement executions at future animation times, VITASCOPE’s add-on interface provides add-ons with access to the current animation time during visualization. Add-ons can access the current animation time maintained by VITASCOPE’s clock from within any routine via the function `dcvGetCurSimTime`.

The following simple example illustrates how these add-on interface-provided functions facilitate concatenation of foundational animation statements from within add-ons to provide engineers with higher-level animation constructs to visually describe simulated construction processes. Figure 9 presents an annotated VITASCOPE animation trace segment that describes the assembly, loaded travel, dumping, and empty return of a dump truck. Only foundational language statements are used in the description.

To present engineers with animation statements that describe and manipulate the same dump truck in construction parlance as opposed to geometric transformation terminology, we can design a simple VITASCOPE add-on that defines higher-level statements to
instantiate and manipulate the dump truck. The implementation of such a simplistic add-on (called DumpTruck) is presented in the C++ code listing in figure 10.

```c++
// Predefine Paths LoadToDump and DumpToLoad
CLASS TruckChassis MackChassis.wrl;
CLASS TruckDumpbed MackDumpbed.wrl;

SET CLASS TruckChassis AFTCLEARANCE 5;
SET CLASS TruckChassis RGP 5.25;

TIME 0;
CREATE Truck1 TruckChassis;
CREATE Dumpbed1 TruckDumpbed;
ATTACH Dumpbed1 Truck1 (-4,2,2,0);
PLACE Truck1 ON LoadToDump;

TIME 10;
MOVE Truck1 LoadToDump 110;

TIME 120;
ROTATE Dumpbed1 VERT 55 30;
TIME 160;
ROTATE Dumpbed1 VERT -55 20;
```

Figure 9: Dump Truck Assembly and Manipulation with Foundational Statements

The add-on’s VitaAddOnInit function registers the statement DumpTruck with VITASCOPE passing the address of the routine CreateDumpTruck contained within the add-on. This routine executes statements that assemble a truck and place it in the scene from within the add-on using the interface provided C++ functions. It also registers the statements DumpTruck.Haul and DumpTruck.Dump passing the addresses of two other add-on routines. Each of these routines in turn describe the hauling and the dumping of the truck by executing the same foundational VITASCOPE statements presented in the animation trace segment in figure 9.

This simple scheme now allows a dump truck to be visually described and manipulated using a higher-level, domain parlance. This is presented by the animation trace segment in the upper half of figure 11. The arrows pointing to the add-on module routines indicate the code that VITASCOPE executes when the particular statements are processed during visualization. The add-on routines in turn execute core foundational statements to
visually describe the proceedings. The animation trace segments in figure 9 and the top of figure 11 are thus functionally equivalent, albeit at different levels of abstraction.

```c
extern "C" int VitaAddOnInit(const char* szTraceFileName)
{
    // Execute Initial Statements
    dcvExecuteStatement("CLASS TruckChassis MackChassis.wrl");
    dcvExecuteStatement("CLASS TruckDumpbed MackDumpbed.wrl");
    dcvExecuteStatement("SET CLASS TruckChassis AF CLEARANCE 5");
    dcvExecuteStatement("SET CLASS TruckChassis RGB 5.25");

    // Register add-on defined statements
    dcvRegStatement("DumpTruck", CreateDumpTruck);

    return 1;
}

int CreateDumpTruck(const char* szArgs, void* p1, void* p2)
{
    // Execute foundational statements to assemble the truck
    dcvExecuteStatement("CREATE Truck1 TruckChassis");
    dcvExecuteStatement("CREATE Dumpbed1 TruckDumpbed");
    dcvExecuteStatement("ATTACH Dumpbed1 Truck1 (-4,2.2,0)");
    dcvExecuteStatement("PLACE Truck1 ON LoadToDump");

    // Register truck manipulating statements
    dcvRegStatement("DumpTruck.Haul", HaulDumpTruck);
    dcvRegStatement("DumpTruck.Dump", DumpTruckLoad);
}

int HaulDumpTruck(const char* szArgs, void* p1, void* p2)
{
    // Execute foundational statement to haul truck
    dcvExecuteStatement("MOVE Truck1 LoadToDump 110");
}

int DumpTruckLoad(const char* szArgs, void* p1, void* p2)
{
    // Execute foundational statement to raise dumpbed
    dcvExecuteStatement("ROTATE Dumpbed1 VERT 55 30");

    // Schedule foundational statement to lower dumpbed
    // 40 time units later
    double dCurSimTime = dcvGetCurSimTime();
    double dDumpbedLowerTime = dCurSimTime+40;
    dcvScheduleStatementExecution(dDumpbedLowerTime,
                                    "ROTATE Dumpbed1 VERT -55 20");
}
```

Figure 10: Higher-Order Statements for Dump Truck Assembly and Manipulation
The presented example is an oversimplified case included merely for illustration. The add-on presented assumes and supports the existence of only one dump truck that hauls and dumps a single load in the visualization. In practice, such an add-on would represent each dump truck as a C++ class that would register manipulation methods for every instantiated truck, thus allowing any number of trucks to be generally created and manipulated individually using a higher-level parlance and stochastic parameters extracted from the add-on function’s arguments. The illustration is however, only
intended to describe the extension scheme that VITASCOPE’s add-on interface describes without including too many C++ programming specifics.

5.4 Multi-Tier Extensible Model

VITASCOPE’s add-on interface supports a multi-tiered extensible model. As described in the previous subsection, first tier extension animation statements may be designed to extend VITASCOPE’s language by concatenating foundational language statements. In addition to implementing new statements, designed add-ons can augment, override, or redefine any previously existing VITASCOPE statement. VITASCOPE’s language can also be extended with new statements that perform specialized computations without reusing or modifying the core foundational language blocks.

The forte of VITASCOPE’s multi-tier extensible model is that, at any instant, designed add-ons have access to not only the foundational language, but any other add-on statements loaded previously. Designed add-ons can thus seamlessly reuse, redefine, or augment not only the foundational language, but also all subsequent add-on implemented statements as well. At any instant, add-ons thus have access to the entire lexicon of VITASCOPE’s interpreter regardless of whether the statements are foundational or have been appended by previously loaded add-ons. This powerful feature directly supports multiple efforts towards incremental evolution of the underlying technology.

In the following section, we now describe examples of implemented extensions to the VITASCOPE animation language. These extensions have been implemented using the presented add-on interface and extend VITASCOPE’s language by capitalizing on its core facilities where appropriate.

6. Validation of Extensibility

VITASCOPE’s add-on interface has been used to design and implement several extensions to the foundational animation language. The extensions have been implemented as software tools that contribute towards enhancing engineers’ ability to visualize modeled construction processes and products. In addition, from the perspective
of this work, they allowed us to repeatedly validate the extensibility and the scalability of
the visualization framework this paper presents. A common observation about extensible
framework design is that it requires iterations to substantially evolve (Johnson and Foote
1988). Designing and implementing the presented add-ons pointed out several weak parts
in VITASCOPE’s extensible framework that in turn led to significant improvements.

We provide here only a brief description of selected add-ons designed using the presented
VITASCOPE add-on interface. The purpose of the discussion is to highlight the nature of
the relevant and useful higher-level research issues that can now be addressed
incrementally by capitalizing on the work already in place. The specific research issues
addressed in designing and implementing each add-on in most cases warrant detailed
discussion with relevant examples that are beyond the scope of this paper. Such details
are presented in this dissertation in the manuscripts that follow this paper.

6.1 Example 1 - ParticleWorks

VITASCOPE’s geometric transformation-based foundational animation language does
not provide constructs that can allow the description of construction processes that
involve use of unstructured, fluid construction materials. Numerous unstructured
materials generally capable of flowing (e.g. concrete, dirt, mortar, sand, slurry, and
water) are however together central to most construction processes. Common processes
such as dumping dirt, distributing water, dewatering caissons, placing concrete,
sandblasting, shotcreting, and slurry-wall construction cannot be realistically visualized
in smooth, continuous 3D worlds unless methods to accurately represent dynamic
volumes of the involved unstructured construction materials are designed.

The ParticleWorks add-on for VITASCOPE presents efficient methods that engineers can
use to visualize such construction processes involving “fuzzy”, unstructured, materials
that are generally capable of flowing (see Chapter 4 for details). The work capitalizes on
a classical computer graphics concept called particle systems and VITASCOPE’s add-on
interface to design simple, parametric text methods to represent arbitrary dynamic
volumes of fuzzy construction materials in 3D virtual construction environments.
Engineers can use these add-on defined methods to instrument simulation models to automatically generate dynamic visualizations of any modeled operations that commonly handle and process fuzzy construction materials.

### 6.2 Example 2 - KineMach

KineMach implements “smart” pieces of virtual construction equipment that can be instantiated and manipulated using simple text statements in a higher-level, contextual, construction terminology (see Chapter 5 for details). Currently implemented generic pieces of equipment include a tower crane, a crawler mounted lattice boom crane, a crawler mounted backhoe, and a highway dump truck. Engineers can use KineMach provided statements to instantiate multiple pieces of this equipment and instruct them to perform virtual construction processes using a high-level construction work terminology.

KineMach is implemented as an add-on to VITASCOPE. It capitalizes on robust forward and inverse kinematics algorithms from robotics literature and VITASCOPE’s flexible add-on interface to design high-level statements for virtual equipment instantiation and manipulation. KineMach embodies a similar spirit to the simplistic dump truck add-on described in an earlier section. The actual algorithms used in its design and other implementation issues are however much more complex. KineMach’s statements provide engineers with a construction terminology to visually describe construction processes. In addition, KineMach greatly simplifies the operation modeling process of simulation models intended to automatically author visualizations, making 3D simulation-driven process animation more practical.

### 6.3 Example 3 - C-Collide

C-Collide is an add-on designed for VITASCOPE that engineers can use to identify and report any and all undesirable conflicts that can occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources in dynamic 3D construction process visualizations (see Chapter 8 for details). Common types of clashes that can occur on real construction sites and that C-COLLIDE can identify beforehand in process visualizations.
include 1) intersection among physical in-place components (i.e. design interferences), 2) intersection among in-place components, components in transit, and/or pieces of moving equipment during construction (i.e. constructability interferences), 3) craft interferences and accidents e.g. collision between two pieces of equipment operating in the same area, and 4) space intrusions e.g. any resource (worker or equipment) encroaching arbitrarily shaped hazard or protected areas of the jobsite.

C-Collide capitalizes on the presented VITASCOPE add-on interface and advanced documented algorithms for efficient collision detection between arbitrarily moving 3D geometric objects to design mechanisms for interference detection, control, and response in 3D construction process visualizations. C-COLLIDE’s interference detection capabilities dynamically check each motion of VITASCOPE scene objects to determine if any pairs of scene objects interfere undesirably. This provides engineers with a lucid understanding of all object motions and potential interferences in any area of activity on an animated construction job site.

6.4 Example 4 - ExcelWorks

The ExcelWorks add-on for the VITASCOPE visualization system is a tool designed to allow engineers to juxtapose dynamic displays of quantitative, numerical simulated operation data alongside 3D view ports during visualization. Within a 3D virtual world that presents a smooth, continuous construction operation, the dynamic display of relevant quantitative information can be useful and, at times, critical in conveying critical statistics about the operation being visualized. The dynamic quantitative information that can be graphically displayed when visualizing a simulated operation includes the numerous simulation run-time statistics maintained by a running process model.

ExcelWorks capitalizes on VITASCOPE’s add-on interface and the OLE automation features of MS Excel to design a text statement-controlled dynamic charting tool. Engineers can use ExcelWorks-implemented statements to define and update a dynamic graphical display of charts within an MS Excel window. Excel’s automation server allows VITASCOPE to take control of its sheets and display any user-defined
information graphically during visualization. ExcelWorks’ statements communicate the desired information to the add-on module. The information, after suitable manipulation, is then appropriately forwarded to Excel’s automation server for display in chart sheets.

The VITASCOPE add-ons described above have all been implemented as software tools on both MS Windows and IRIX (except ExcelWorks) platforms. Their implementation required no knowledge or modification of any VITASCOPE internal design features or its implementation code. The work required only an understanding of VITASCOPE’s add-on interface and competence in the C++ programming language in addition to relevant domain knowledge. It is a mere coincidence that we, having designed and implemented VITASCOPE and its add-on interface ourselves, were completely familiar with its internal design and implementation.

7. Future Work

Engineers’ ability to visualize construction processes can be enhanced and extended in endless interesting ways. As the presented add-on examples demonstrate, non-trivial advances in the state-of-the-art of simulation-driven construction process visualization can now be contributed incrementally through the collective effort of many researchers. In particular, VITASCOPE’s add-on interface allows researchers to extend the 3D animation language in endless ways to advance engineers’ ability to visualize simulated construction processes and products. In addition, the interface facilitates the design and implementation of higher-level tools for solving engineering problems that require engineers to visualize a construction operation.

VITASCOPE’s open and loosely coupled visualization scheme can allow visualizations to be automatically described by any external process capable of generating formatted text output in addition to the required contextual descriptive information. In addition to a wide variety of simulation tools, this opens the possibility of driving construction operation visualizations by other software and hardware interfaces. Future research can, for instance, capitalize on VITASCOPE’s add-on interface to design methods that manipulate and update scene objects in a virtual world based on real-time field sensor
data. Future research can also build upon VITASCOPE’s extensible framework to design methods that allow humans and human-controlled machines to interact with a simulation model-controlled virtual construction operation. We expect VITASCOPE’s add-on interface to progressively evolve as necessary in response to these research efforts.

8. Summary and Conclusions

The advancement of simulation-driven construction process visualization technology requires the collective efforts and skills of many different researchers such that it incrementally evolves per the scientific method. Acknowledging this fact, the presented work contributes the computing infrastructure for construction research that allows researchers to 1) seamlessly extend engineers’ capability to visualize modeled construction processes, and to 2) conceive and implement solutions to higher-level construction process visualization problems. The VITASCOPE language and the designed extensible visualization framework allows researchers to focus valuable attention and research resources on advancing the state-of-the–art rather than on repeatedly designing and implementing foundation-level visualization support.

An extensible and scalable framework to visualize modeled construction processes and the evolving products requires two key technologies:

1. Identification and implementation of a core, reusable, foundational set of visualization-describing building blocks.

2. Design of methods that provide suitable access to the foundational facilities and describe an extensible interface to their abstractions.

Research involved in designing VITASCOPE’s foundational 3D animation language proved that geometric transformation-based animation statements are 1) collectively sufficient to visually describe a broad class of common construction processes, and 2) are at a level of abstraction that they can be logically concatenated to describe several higher-level motion dynamics involved in performing construction.

In addition, experience gained implementing the presented add-ons using VITASCOPE’s add-on interface proved that an open, loosely-coupled visualization scheme and direct
interface methods to append the animation statement interpreter’s vocabulary with new add-on designed statements 1) allows language extensions without modification to or understanding of the underlying facilities, and 2) presents engineers with a consistent interface to visually describe construction processes. The presented examples of non-trivial VITASCOPE add-ons also confirm that an extensible construction process visualization framework greatly facilitates fast, incremental advances to the state-of-the-art and encourages discoveries that may dramatically reduce the life-cycle costs of constructed facilities.

9. References


Chapter 4
Dynamic 3D Visualization of Fluid Construction Materials

1. Introduction

In discrete-event simulation (DES) analyses, the ability to see a 3D animation of an operation that has been modeled allows for three very important things: 1) The developer of the simulation model can confirm that there are no errors in the coding (verification); 2) The domain experts, field personnel, and decision makers can discover differences between the way they understand the operation and the way the model developer understands it (validation); and 3) The model can be communicated effectively offering decision makers valuable insight on subtle operational details that would otherwise be non-quantifiable and non-presentable. This coupled with verification and validation, makes simulation models credible and encourages their use in making decisions (Law and Kelton 2000).

The current state-of-the art in scientific simulation-driven construction process visualization allows engineers to visualize a limited number of construction processes in 3D. These include processes in which resources (equipment and crew) always move on fixed well-defined paths, processes that do not require any physical deformation of rigid resources, and in particular processes that do not involve use of unstructured, fluid construction materials. Numerous unstructured materials generally capable of flowing are however commonly handled and processed on typical construction sites. Indeed, materials such as concrete, dirt, mortar, sand, slurry, and water are together central to most construction processes. Common processes such as dumping dirt, distributing water, dewatering caissons, placing concrete, sandblasting, shotcreting, and slurry-wall construction cannot be realistically visualized in smooth, continuous 3D worlds unless methods to accurately represent dynamic volumes of the involved unstructured construction materials are designed.
The measure of virtual realism necessary or sufficient to fully exploit scientific visualization of simulated construction processes is often a matter of subjective opinion. While simple, symbolic, iconic 2D animation is adequate for some, others strive for detailed 3D visualization in immersive, photorealistic virtual worlds. Notwithstanding the scientific community’s polarization, it is undeniable that in general, engineers’ expectations from computer graphics have increased manifold in recent years. This can be largely attributed to the impressive computer-generated sequences the motion picture industry creates in animated films and special effects.

What these expectations signify for scientific visualization of simulated processes is that displayed graphics must be faithful, comprehensive, and should not need any verbal or textual clarification (Rohrer 2000). Since visualization is principally a communication tool, engineers must be able to animate faithful, realistic depictions of all modeled real-world processes (Farr and Sisti 1994). Construction engineers, for instance, must be able to graphically see processes such as concrete being placed into virtual forms, dirt pouring out from virtual dump trucks, and mortar being projected at high velocity from virtual shotcrete applicators.

The research this paper presents explores several original concepts to design effective process simulation driven methods to describe the dynamics of unstructured construction materials in 3D virtual worlds. Using principles of Newtonian mechanics where appropriate, we designed techniques that allow engineers to express the accurate motion, changes of form, and dynamics of unstructured, non-rigid material volumes using simple parametric statements of text. The work thus puts in place the methodology that allows engineers to create realistic scientific construction process visualizations involving any operation that commonly handles and processes unstructured “fluid” construction materials.
2. Challenges

The design of automatic, simulation-driven methods to visualize fluid construction materials in smooth, continuous, dynamic, 3D virtual worlds presents numerous interesting challenges. Volumes of fluid construction materials such as concrete, dirt, gravel, mortar, sand, slurry, and water naturally do not have fixed or deterministic shapes and forms. Such materials flow under the influence of prevailing natural (e.g. gravity) and imparted (e.g. pump pressure) forces until physical equilibrium is established. Computational fluid dynamics literature provides classic models such as the Navier-Stokes equations to describe the motion and behavior of flowing liquids. Such models although highly accurate, require very intensive computation to solve. The dynamics of rigid bodies, on the other hand, can be described using classical Newtonian mechanics.

Methods designed to allow external authoring interfaces (e.g. discrete-event simulation models) to automatically describe processes involving fluid construction materials demands that the methods themselves be simple. Simplicity, in this context means that the syntax of the methods and the values sought by their parameters are both within the authoring capabilities of such external processes. In addition, the methods must incorporate rich semantics so that the necessary descriptions are communicated with minimal inter-process interaction.

Regular objects in virtual environments are generally represented using surface-based, polygonal CAD models. Such objects typically have fixed, well-defined, smooth surfaces. Fluid object surfaces, on the other hand, are irregular, complex, and ill defined and cannot generally be represented with smooth surfaces inside virtual worlds.

The issues we address in this work are thus threefold. First, we investigate feasible procedures to accurately describe the motion and dynamics of arbitrary volumes of such fluid materials (i.e. the physical model). Second, we design simple methods to encapsulate such physical descriptions in a finite number of parametric statements of text (i.e. the interface model). Finally, we explore techniques to represent the descriptions in dynamic 3D virtual construction worlds such that the results looks visually convincing,
move realistically, and can be animated in real-time (i.e. the implementation model). It is obvious that these issues are tightly intertwined.

3. Initiative

In attempting to address these issues, we found most promise in a technology based on the concept of the particle system. Particle systems represent a classical technique of describing fluid objects in interactive virtual environments. In addition, the semantics of the technique are such that the approach can incorporate any physical computational model to describe the appearance or dynamics of fluid objects. As the following discussion will clarify, this is a significant feature that immensely facilitates the design of generic methods to describe dynamic fluid construction materials in animated 3D worlds.

William T. Reeves (1983) coined the term “Particle System”. He used it to describe a method that uses an arbitrary number of geometric primitives (called particles) in virtual space to visually represent certain objects or effects that can not be represented using conventional techniques. He called an object represented by such a particle system a “fuzzy object”. Examples of physical phenomena and materials that can be called fuzzy objects include clouds, smoke, water, and fire. Reeves and his colleagues first applied this concept in creating the wall of fire element from the Genesis Demo sequence of the film “Star Trek II: The Wrath of Khan” (Paramount 1982).


Fluid objects by definition do not have smooth and well-defined surfaces. On the contrary, their surfaces are irregular, complex, and ill defined. Describing realistic visual representations of such fuzzy, flowing objects in virtual environments generally proves

Many unstructured materials such as dirt, sand, mortar, concrete, and slurry are commonly handled and processed on construction sites. Such construction materials may be appropriately termed fluid as they are not rigid objects and are typically capable of flowing until a stable physical equilibrium is established. The dynamics of such objects (e.g. concrete flowing from a bucket, dirt being emptied from a truck etc.) cannot be visually represented using simple geometric transformations commonly used in computer graphics. The concept of the particle system presents an ingenious technique to describe the dynamic and fluid changes in the shape and appearance of such material masses in virtual construction environments.

4. Particle Systems

A particle system is an abstraction for simultaneously operating on many similar objects that all move and change according to the same basic rules, no matter what the objects and rules are. A particle system is realized as a simple collection of many particles that together represent a fuzzy object. The important issue when describing a particle system is the overall appearance of the entire object it represents. Reeves (1983) describes the following three basic ways in which particle system representations differ from other techniques of describing physical objects:

- An object is represented as a cloud of primitive particles that define its volume, and not by a set of primitive surface elements, such as polygons or patches, that define its boundary.
- A particle system is not a static entity. Its particles change form and move with the passage of time. New particles are "born" and old particles "die."
- An object represented by a particle system is not deterministic, since its shape and form are not completely specified. Instead, stochastic processes specify parameters that are used to create and change an object's shape, appearance, and behavior.
4.1 Particle Attributes
Each particle contained in a particle system can be thought of as an entity with a set of attributes that ultimately dictate the particle's behavior and appearance. A particle’s attributes typically include Mass, Color, Velocity, Size, and Age. A particle in this context is therefore not necessarily a very tiny speck but can represent many different things. Each particle within a particle system may be represented as a geometric primitive (e.g. point, line, triangle, quadrilateral etc.), a water droplet, a grain of dirt, or even a complex multi-polygon CAD model (e.g. bird, human, blade of grass etc.).

4.2 Recursive Model
Figure 1 schematically presents the sequence of actions that must be performed to describe each frame in a motion sequence involving particle system represented fuzzy objects. The following minimal steps are performed:

- New particles are generated into the system; Each new particle is assigned its individual attributes
- Any particles that have existed within the system past their prescribed lifetime are destroyed
- An image of the remaining particles is rendered
- The future course of the particles is calculated according to their dynamic attributes
- The particle attributes are transformed to new values by integrating over the elapsed time interval

4.3 Particle Dynamics
Individual particles within a particle system move in three-dimensional space and may also change other attributes such as color and size over time to depict the evolution of the fuzzy object they represent. At each instant in time, particles can be moved from one position to the next by simply knowing their current position and current velocity (magnitude and direction).
The rules that govern how a particle system behaves and evolves over time may be specified and interpreted as a set of arbitrary forces that act on each particle within the system. A particle system typically uses an acceleration factor to modify the velocity of its particles at each instant. This attribute can be used to simulate external physical forces (e.g. gravity) that might cause particles to move in parabolic arcs rather than in straight lines for instance. The evolution of the particle system over time i.e. the shape and form of the represented fuzzy object is then the result of all specified forces acting on each constituent particle.
The forces acting on the particles at each instant can be resolved into their corresponding accelerations using Newton’s fundamental laws of mechanics. The accelerations themselves represent the rates of change in the velocities of the particles. Applying fundamental results from calculus and using numerical integration techniques, it is possible to obtain the new velocity and subsequently the new position of each particle in the particle system. In addition to specifying the forces that act on the system, it is also necessary to specify rules that are responsible for destroying particles in the system and emitting new ones as needed. Over a period of simulated time, particles are thus generated into a system, move and change form within the system, and die from the system.

The section that follows describes our approach in adapting the particle system paradigm to visually represent the dynamic behavior of fluid materials commonly handled and processed on construction sites.

5. Technical Approach

Particle systems can be used to visually represent phenomena that do not necessarily conform to the physical laws of nature we experience and understand. It is obvious that the “forces” each particle in a particle system is subjected to inside a virtual world need not necessarily have physical world counterparts. As a corollary, particles need not be subjected to even the obvious physical world forces (e.g., gravity). Particle systems can thus be used in unimaginable ways to represent interesting visual effects that might generally not occur in the real physical world. The motion picture industry effectively exploits this result to create dramatic special effects in feature films.

Construction operations however are performed in the real world. Scientific construction process visualizations must convey faithful representations of the corresponding real world operations. This includes accurate representation of the appearance and dynamic physical behavior of fluid materials commonly handled and processed in construction.
The men, machines, and materials involved in construction all obey the physical laws of nature as we understand them. For instance, we realize that concrete from a bucket flows into a form when the latch is released because of gravitational force. Gravity acts on each “globule” of fresh concrete, forcing it out of the bucket and downward towards the form. Similarly, consider the process of pumping water out of a caisson. The parabolic path the gushing water follows before striking a surface is a function of both gravity and the pressure with which the water is pumped out.

It is thus obvious that all construction materials (fluid or otherwise) follow the rules of natural physics. Examples of physical phenomena (in addition to gravity) that can influence the behavior of common materials include viscous drag (i.e. air resistance), friction, and resilience. Particle systems that represent fluid construction material masses must therefore evolve under the influence of such real world force counterparts.

5.1 Pertinent Mechanics

In order to visually describe fluid construction materials, we represent each particle in a particle system as a point mass acted upon by the prevalent natural and imparted physical forces. Included in Newton's results is the recognition that mass and velocity of the particles are what affect them (and a system). In particular, we can disregard the sizes and shapes of the particles and treat them as point masses. To describe the particle system’s dynamic behavior, we then resolve the resultant of the forces acting on each particle.

The mechanics of this method hinge on the fundamental relationship between the resultant force \( f \) imparted to each particle, the mass of that particle \( m \), and the acceleration \( a \) of that particle. This can be stated in the familiar Newtonian notation as \( f = ma \). Assuming that particle masses are fixed, from this law we can derive equations that express how position and velocity change over time.

The velocity \( v \) of a particle is simply the first derivative with respect to time of its position \( x \), i.e. \( v = \frac{dx}{dt} = \dot{x} \). A particle’s acceleration is similarly the first derivative
with respect to time of its velocity ($v$) and the second derivative with respect to time of its position ($x$), i.e. $a = \frac{f}{m} = \frac{dv}{dt} = \ddot{v} = \frac{d^2 x}{dt^2} = \ddot{x}$. In our context, given a set of forces acting on several particles at any time instant $t$, we must determine the location of each of the particles at the next time instant $t+h$ (i.e. after a small amount of time $h$ has passed). It is obvious that given all active forces and the masses of the particles, we can obtain the acceleration of the particle. We must then integrate that acceleration with respect to time ($t$) to determine the new velocity of each particle. By integrating again, we can calculate each particle’s new position.

Particle acceleration ($a$) is the second derivative with respect to time of its position ($x$). Introduction of the first derivative quantity (velocity ($v$)) allows us to consider only a system of first order ordinary differential equations (ODEs) that each determines a function of time. It is obvious from this discussion that describing the evolution of particles in a dynamic particle system is an initial value problem. Given an initial state (position and velocity) and active forces, the dynamic behavior of each particle in a system can be evolved over time.

### 5.2 Computation Scheme

We solve our initial value problem by numerical approximation. Several methods suitable for solving such class of problems exist in mathematical literature. Some classical single-step numerical integration methods include Euler’s method, the Midpoint method, and the higher-order Runge-Kutta methods. These methods generally differ in the amount of computation required to arrive at an approximation, the accuracy of the computed solution (i.e. the approximation error), and the stability of the computed solution. In general, higher accuracy and stability come at a corresponding computation cost. We refer the reader to Press et al. (1992) for additional discussion of these and other classes of numerical integration methods from the computation perspective.

Few numerical integration methods are however suitable for real-time particle system animation on the computer. Many methods adapt the integration step size to the function
itself which in this case is the path of each particle. However, giving each particle its own step size requires much more computation. In addition, this would also cause each attribute of a particle to require a different, conflicting step size.

We use Euler’s forward integration method in this research. This method is not considered to be exceptionally accurate or stable in mathematical literature. However, a survey of literature describing applications of the particle system concept to solve engineering problems suggests that this method is popular due to its simplicity and computational efficiency (McAllister 2001, Witkin and Baraff 2001). In addition, the method allows the accuracy of the computed solutions to be interactively adjusted by varying the step size of integration. This permits us to interactively elicit acceptable physically-based visual simulation accuracy in describing the behavior of fluid construction objects. We were thus naturally inclined to adopt this method in our work.

The reader will recall that Euler’s forward integration method simply approximates the next value of the function $x$ of $t$ by the sum of the first two terms of the Taylor series expansion. Mathematically, it is states that for sufficiently small values of the integration step size $h$, $x_{n+1} \approx x_n + h \left( \frac{dx}{dt} \right)_n$. The truncation error in $x_{n+1}$ obviously depends on $h$.

By selecting $h$ to be of sufficiently small size, we can force the difference between the real function value and the approximation to be less than some required error magnitude $e$ (Burden and Faires 1989). However there is a price to be paid. As the step size $h$ is decreased, the number of steps and computation required to bridge an interval increases.

We now use the vocabulary of our problem domain to demonstrate how we exploit the simple semantics of Euler’s method to solve an interesting engineering problem.

### 5.3 Iterative Algorithm

We use a discrete time approximation when applying forces to particles. This implies that forces are applied to the particles at a particular instant in time as if the forces’ effect accumulated over a small time interval $h$. The simulation clock is then ticked by the
length of the interval and the forces are then reapplied with the particles having their updated values. We consider only the unary forces gravity and viscous drag in our analysis. These forces act independently on each particle, either exerting a constant force, or one that depends on particle velocity. Gravity is a constant force that is applied to all particles. The viscous drag (i.e. air resistance) that each particle is subjected to is taken to be a function of the particle’s velocity.

Consider a mass of concrete flowing into a form from a bucket when the latch is released. For purposes of visual simulation only, we consider the flowing mass to be a stream of concrete globules each represented by a particle. Each globule (i.e. particle) is then taken to be an object falling under gravity with air resistance taken into account. For simplicity, we ignore the forces of spatial interaction such as adhesion, surface tension, and attraction that may act on any or all pairs of concrete particles, depending on their positions.

We represent all dynamic particle attributes as three-dimensional vectors that encapsulate both magnitude and direction where appropriate. For instance, we interpret particle velocity as a vector having its tail at the particle’s position and its tip at a particular point inside 3D Euclidian space. The vector’s length and direction together specify both the magnitude and the direction of a particle’s velocity. Any force acting on a particle and the resulting acceleration are similarly interpreted as directional vectors. A particle’s position, on the other hand, is interpreted as a three-dimensional vector that specifies only its location in 3D space.

In the subsequent discussion, we will denote all attributes interpreted as three-dimensional directional vectors with an arrowhead above the attribute’s symbol. All the ideas of integral curves, numerical approximations, etc. carry over intact to three-dimensional space. The only change is in our interpretation of the resulting trajectories. The force of gravity is represented by the familiar force law $\vec{F}_g = m\vec{G}$, where the constant acceleration $\vec{G}$ is $9.81 \, m/s^2$ acting downward towards the earth’s surface.
Viscous drag or air resistance is taken to be proportional to velocity and is represented as
\[ \vec{f}_{\text{drag}} = -k\vec{v}, \]
where the empirical constant \( k \) is the co-efficient of drag. The direction of the drag force is obviously against the velocity direction.

Based on our earlier discussion, we can express the following straightforward equation:
\[
\frac{d\vec{v}}{dt} = \frac{m\vec{G}}{m} - \left( \frac{k}{m} \right) \vec{v} = \vec{G} - \left( \frac{k}{m} \right) \vec{v}
\]

(1)

Using an Euler solver, the value of the velocity at the end of time step \( n+1 \) is now simply given by \( \vec{v}_{n+1} \approx \vec{v}_n + h\vec{c}_n \), where \( \vec{c}_n \) is the net acceleration as given by equation 1. If air resistance is neglected, the net acceleration \( \vec{c}_n \) is replaced by the constant \( \vec{G} \). Otherwise, the algorithm proceeds exactly as in the case of the non-uniform acceleration considered above. This is also true when forces other than gravity and drag are active. We simply accumulate all the active force vectors in formulating equation 1.

We then use a similar procedure to formulate the equation for the value of each particle’s position at time step \( n+1 \). We use our other first order differential equation to get
\[
x_{n+1} \approx x_n + h\vec{v}_n.
\]
We thus integrate \( \frac{dx}{dt} = \vec{v} \) at the same time we integrate \( \frac{d\vec{v}}{dt} = \vec{c} \), alternating as we go. The two coupled ODEs thus give us an iterative algorithm for computing particle positions resulting from any arbitrary resultant force on our particle systems.

To implement this algorithm, then, all we must do is specify the initial position \( (x_0) \) and the initial velocity \( \vec{v}_0 \) of each particle in our system. We then use \( \vec{v}_0 \), the velocity at time \( t_0 \), to calculate the value of \( \vec{c}_0 \), the net acceleration at time \( t_0 \). We then use these values, along with \( x_0 \), to calculate the position and velocity at time \( t_1 = t_0 + h \). These values along with the net acceleration at time \( t_1 \) are then used to calculate the values at \( t_2 \) and so on. We now describe our implementation of these algorithms. The implementation is a
powerful tool that allows simple, parametric text descriptions of dynamic volumes of fluid objects in 3D construction process visualizations.

6. ParticleWorks

Our implementation of particle systems and the algorithms that manipulate them is a tool that allows engineers to accurately represent masses of fluid, unstructured, flowing construction materials in smooth, continuous, dynamic 3D virtual worlds. This tool, called ParticleWorks, is implemented as an extension (add-on) to the VITASCOPE visualization system. VITASCOPE is a user-extensible 3D animation language designed specifically for visualizing simulated construction operations in smooth, continuous, dynamic 3D virtual worlds (see Chapter 2 for details). A limited subset of the VITASCOPE language and the corresponding prototype implementation were referred to as the Dynamic Construction Visualizer (DCV) in some early publications (Kamat and Martinez 2001, 2002). VITASCOPE is the tangible outcome of ongoing visualization research efforts at Virginia Tech that focus on designing automated, process simulation-driven methods to visualize construction processes and products in dynamic 3D virtual worlds.

The ParticleWorks add-on extends the VITASCOPE animation language. The add-on defines animation language statements that allow instantiation and interactive manipulation of dynamic particle systems. The parametric text statements can be used together in interesting ways to describe dynamic, realistic-looking, masses of several common construction materials such as dirt, concrete, slurry, and water in process visualizations. ParticleWorks provides statements that allow engineers to specify the static (e.g. mass, color, size) and dynamic (e.g. initial position, initial velocity) properties of the particles in an instantiated particle system. In addition, the add-on implements several statements that help define properties that influence the behavior of an instantiated system as a whole. Examples of such properties include surfaces in the virtual world off which particles bounce rather than sink in, surfaces which confine system particles rather than permit leakage/penetration (e.g. concrete forms), the number of particles to generate per time instant, and the lifetime of constituent particles. These
methods together allow engineers to use simple parametric text statements to create
dramatic visual representations of dynamic fluid construction materials. Table 1 presents
selected statements ParticleWorks implements and briefly indicates their usage. The
meaning and significance of each statement argument will be clear in the following
discussion. The detailed usage of all statements is presented in appendix G.

### Table 1: Usage of Selected ParticleWorks Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUZZYOBJECT [ObjName] [...]</td>
<td>Instantiate and initialize a particle system source</td>
</tr>
<tr>
<td>[ObjName].SetVelocity [DomainType] [Parameters];</td>
<td>Set the initial velocity of instantiated particles</td>
</tr>
<tr>
<td>[ObjName].SetSource [DomainType] [Parameters];</td>
<td>Set the source where particles are generated if not set with FUZZYOBJECT</td>
</tr>
<tr>
<td>[ObjName].SetRate [ParticlesPerUnitTime];</td>
<td>Set the number of particles entering the system at each time instant</td>
</tr>
<tr>
<td>[ObjName].SetAgeLimit [AgeLimit];</td>
<td>Set how long particles stay in the system before being destroyed</td>
</tr>
<tr>
<td>[ObjName].SetSink [SinkLevel];</td>
<td>Set the ground level below which entering particles are destroyed</td>
</tr>
<tr>
<td>[ObjName].SetColor [DomainType] [Parameters];</td>
<td>Set the color of instantiated particles</td>
</tr>
<tr>
<td>[ObjName].SetMass [Value];</td>
<td>Set the mass of instantiated particles</td>
</tr>
<tr>
<td>[ObjName].SetPrimitive [PrimitiveType];</td>
<td>Set what geometric primitive is used to draw particles (points or lines)</td>
</tr>
<tr>
<td>[ObjName].AddObstruction [DomainType] [Parameters];</td>
<td>Add a physical barrier surface obstructing particle flow (e.g. concrete forms)</td>
</tr>
<tr>
<td>[ObjName].SetGravity [NewGValue];</td>
<td>Change $\tilde{G}$ to visually simulate buoyancy underwater or loss of gravity in outer space.</td>
</tr>
</tbody>
</table>
6.1 Fluid Object Definition

ParticleWorks considers each instantiated fluid object (represented as a dynamic particle system) to be an imaginary, invisible particle source. It is only after the particle source emits visible particles that the fluid object itself is visible. For instance, consider a water sprinkler truck that distributes water over haul roads and fills on earthmoving jobsites. Water distribution is commonly used in earthmoving to increase the moisture content of the soil to the optimum range for compaction, and to reduce dust along haul routes (Peurifoy and Schexnayder 2002). Distributor trucks drive over haul routes and fills spraying water (at varying rates) from linearly arranged nozzles on rear-mounted spray bars.

In order to visualize this operation, the spraying water (the fuzzy object) can be described as a stream of particles emitted from a linear source particle system that is attached to a normal polygonal CAD model of a sprinkler truck. By linear source, we mean that particles in this system are stochastically generated along a straight line. In this example, the line corresponds to the arrangement of nozzles at the back of the sprinkler truck. The initial positions of the generated particles are thus random points sampled along this straight line.

The discerning reader will quickly realize that the initial velocities of the generated particles in this case are a function of the pressure (i.e. force) with which the truck sprays water. The number of particles to generate per time instant is similarly a function of the truck’s discharge rate. Figure 2 presents an annotated VITASCOPE animation trace segment that describes this operation. ParticleWorks defined statements as well as core VITASCOPE language statements are used in the description. The semantics of each argument are discussed in further detail in the following subsection.

By introducing a ParticleWorks fuzzy object in a scene (either by attachment to another scene entity or by global placement), we merely specify the source of particles in that system and the rules that govern their dynamics. It is only after the source begins to generate and emit particles as time passes that we see the actual volume of the fluid
material (spraying water in this example). The emitted particles are then continuously subjected to physical analysis to describe their dynamic, real world like behavior. Figure 3 presents a snapshot of the animation achieved by processing the above trace segment in the VITASCOPE virtual environment application.

![Figure 2: Defining a Water Distribution Truck](image)

In a real operation, the spraying water obviously travels out of the truck’s storage tank to the nozzles through pipes when the valves are opened. However, in a virtual world, we generate water droplets (i.e. particles that look like water droplets) just at the point where they are visible (i.e. at the linearly arranged nozzles).

It should be obvious from this discussion now, that in order to describe a mass of concrete pouring out from a bucket, all we must do is attach a circular disc shaped particle source on the underside of the concrete bucket model. When the latch is released, the source must emit particles that together resemble a mass of flowing concrete.

Figure 4 presents a trace file segment that describes such a process. A snapshot of the dramatic animation achieved by processing this segment is then presented in figure 5. Engineers cannot visualize such operations using only traditional surface based CAD models. Particle systems make all the difference.
Figure 3: Animation Snapshot of a Water Distribution Process

```plaintext
FUZZYOBJECT Concrete DISC (0,0,0,0,-1,0,0.2);
Concrete.SetModel ConcGrain.wrl;
Concrete.AddObstruction PLANE (0,-0.6,0,0,1,0,0);
Concrete.AddObstruction PLANE (-2.5,0.5,0,1,0,0);
Concrete.AddObstruction PLANE (2.5,0.5,0,-1,0,0);
Concrete.AddObstruction PLANE (0,0.5,-2.5,0,0,1);
Concrete.AddObstruction PLANE (0,0.5,2.5,0,-1,0);
Concrete.SetCoeffFriction 0.1;
Concrete.SetCoeffRestitution 0.05;
ATTACH Concrete Bucket1 (0,0,0);
Concrete.SetRate 30;
```

Figure 4: Describing Concrete Placement Processes
6.2 Fluid Object Control

In order to create a dynamic group of particles that exist and move at various times, ParticleWorks must initiate particles at the desired place at the desired time. In addition, the particles must exhibit a desired appearance and dynamic behavior. In order to achieve this, ParticleWorks must know the rules to be used for selecting initial values of particle properties (e.g. mass, source or initial position, initial velocity, age). ParticleWorks must also know the number of particles to generate each time step and the laws that govern the spatial aspects of their existence. With this data available, ParticleWorks can achieve a continuous stream of the desired particle flow.

As figure 1 suggests, a particle system can be instructed to perform any set of instructions at each time step. Because this is procedural, this approach can incorporate any computational model to describe the appearance or dynamics of the fuzzy object. In this research, we use 1) bounded stochastic processes to control the shape, appearance, and
initial conditions (e.g. position, velocity) of instantiated particles, and 2) physical analysis to control subsequent particle dynamics.

In addition, we deterministically specify any constant properties and physical constraints. ParticleWorks statements in effect provide access to particle and particle system parameters that control these operations. For instance, in cases where stochastic processes randomly select particle attributes, the selected values are constrained by user specified parameters. In general, each such parameter specifies a range in which a particle's value must lie.

### 6.2.1 Concept of Domains

ParticleWorks represents a range with an entity that literature refers to as a Domain (McAllister 2001). A domain can be conceived to be a region of space. For instance, the particle source of an instantiated fuzzy object is specified and interpreted as a domain. In the examples described above (water truck and concrete bucket), the particle source is specified as a straight line (water truck nozzle arrangement) and a circular disc (shape of the bucket opening) respectively.

The source domain describes the region in which new particles in a system will be generated. A random point within the domain is then chosen as the initial position of a new particle. Figures 6 (a) and (b) present this concept graphically for the line and disc domains respectively. The dots in the domain space indicate randomly chosen positions within each domain. The arrows indicate the magnitude and direction of the initial particle velocities. In addition to lines and discs, ParticleWorks supports points, triangles, planes, rectangles, cylinders, spheres, cones, cubes, and blobs as valid domain spaces. Domains may thus define one, two, or three dimensional regions in space.
In addition to describing the source (i.e. initial positions) of new particles, ParticleWorks uses domains to describe the ranges for other stochastically selected particle properties. For instance, the initial velocities of particles may also be described with a domain. In this case, the velocity vector is conceived as having its tail at the origin and its tip at a random point inside the specified domain thus specifying both the magnitude and direction. For instance, consider the indicated initial velocities for the linear source domain in figure 6(a). In order to specify initial velocities of the nature indicated, we can use another linear domain. Figure 7(a) presents a cross-sectional view of the linear source domain presented in figure 6(a). The velocity domain in this case is another straight line on which all the velocity vector tips (arrowheads) lie. Generated particles randomly choose a point in this domain to select their initial velocity (magnitude and direction).
We could also specify this velocity domain as a rectangle instead of a line if we desire greater variability in magnitude (but not in direction). This scenario is presented in figure 7(b). The reader will easily recognize that if all emitted particles in this example were to have the same velocity and the same direction, then the velocity domain is simply a single point (i.e. a point domain).

In addition to positions and velocities, domains are also useful in stochastically selecting the initial color of emitted particles. ParticleWorks is implemented in OpenGL (Woo et al 1997) wherein the full color space is defined by the zero to one (0.0 to 1.0) range along the red, green, and blue axes. For instance, a color represented as (0,0,0) indicates pure black, (1,1,1) indicates pure white, (1,0,0) is pure red etc. The result of specifying particle colors as a domain is that each created particle can select a random color that lies in that domain. For instance, a linear domain between the points (1,0,0) and (1,1,0) will choose points on a line between red and yellow giving each new particle a random color on that line. This property can be used in interesting ways to describe realistic material appearance.

6.3 Fluid Object Dynamics

Once the rules for selecting particle attribute values and particle generation are specified, ParticleWorks fuzzy objects begin emitting particles at the indicated rate at each subsequent time instant. The emitted particles originate at the indicated source and select all attributes in accordance with their designated values. Once emitted, ParticleWorks treats each particle as a point mass under the influence of prevailing forces. Each existing particle is updated at subsequent time instants according to the physically based analysis procedure discussed in the earlier sections.

In addition, ParticleWorks also implements simple collision detection and response of particles in a system. This is necessary to realistically represent many fluid construction materials and processes. For instance, as the animation snapshot in figure 5 presents, when concrete is placed into a form, we obviously want to see the concrete “particles” being physically constrained by the form panels. Similarly, when gushing water strikes a
surface, it is not instantaneously absorbed. Instead, water “particles” create a splash whose features depend on physical properties of both water and the surface.

6.3.1 Collision Detection and Response

For each specified impermeable surface (represented by a planar domain), ParticleWorks tests whether particles will pass from being outside the illegal domain to being inside it if the next positional update were to be applied at a particular instant. If they would pass through the surface of the domain, they are instead bounced off it. There are thus two parts to this interesting problem: 1) detecting particle-plane collisions, and 2) computing an appropriate response to them. General collision detection in virtual environments is a challenging subject and is a major research theme in computer graphics. However, as the following discussion clarifies, simple particle-plane collisions are relatively easy to detect.

Figure 8 presents a dynamic particle moving towards a planar surface. The particle’s current position and velocity vectors are given by $x$ and $\vec{v}$ respectively. The point given by vector $p$ lies on the plane. The vector $\vec{n}$ is normal to the plane and points toward the legal side of the surface.

Figure 8: Detecting Particle-Plane Collision
Using simple vector mathematics, all we must do to detect a possible particle-plane collision is compute the vector dot product \((x - p) \bullet \vec{n}\). It is obvious from figure 1 that the value of the dot product allows us to make the following inferences:

- A value greater than zero indicates that there is no contact and the particle is on the legal side of the barrier.
- A value less than zero indicates that the particle has penetrated the surface and is now on the wrong side of the barrier.
- An exact zero value indicates that the particle is in perfect contact with the barrier surface.

In addition, a particle in perfect contact with the barrier surface may not be colliding with that surface if the particle is moving away from it. We must thus determine the relative velocity of the two bodies by checking the vector dot product \(\vec{n} \bullet \vec{v}\). If the value is negative (indicating opposing directions), we can infer that the particle is in colliding contact with the surface.

When ParticleWorks detects a particle-plane penetration or contact, we compute the physical particle response to the collision as follows. The incoming velocity vector of the colliding particle \(\vec{v}\) is first decomposed into components normal and tangential to the barrier surface. As figure 9 presents, the normal component of vector \(\vec{v}\) can be simply computed as \(\vec{v}_n = (\vec{n} \bullet \vec{v})\vec{n}\). The tangential component is then given by \(\vec{v}_t = \vec{v} - \vec{v}_n\).

![Figure 9: Incoming Velocity Vector Decomposition](image)
The direction of the normal component is then reversed and multiplied by a coefficient of restitution \( k_r \) whose value can be interactively set to simulate collisions of varying elasticity (i.e. to control the particle’s bounce). The tangential component is similarly multiplied by \((1-k_f)\), where \(k_f\) represents the degree of friction the particle experiences when moving horizontally on the surface. The modified velocity components are then recomposed into a new velocity vector \( \vec{v}' \) heading away from the surface. Figure 10 presents this procedure graphically.

ParticleWorks provides language statements to set the values of the restitution and friction coefficients. Interesting combinations of elastic, non-elastic, frictionless, and non-frictionless collisions between particles and rigid surfaces can be described by simply varying the coefficients of restitution and friction in interesting ways. This can be used to visually simulate interaction between different types of fluid materials and rigid physical constraints. However, since flowing material masses are approximated as a group of individual particles, this method is obviously not accurate physically and is only intended to feign realism in visualizations. In addition, as the animation snapshot of a dewatering process in figure 11 indicates, the implementation does not produce particularly accurate visual results for pure liquids that splash rather than bounce.
7. Generic Extensible Approach

ParticleWorks implements general animation statements. The statements allow instantiation and manipulation of generic particle systems using simple parametric text statements. The statements provide access to all interesting particle system attributes that influence the appearance and behavior of that system. Engineers can use these parametric statements in many creative ways to describe several different kinds of fluid objects in virtual worlds.

ParticleWorks does not provide higher-order statements to directly create specific types of fluid construction objects such as volumes of concrete, water, or slurry. Instead, a generic fuzzy object is implemented, and all relevant properties of its constituent particle system are exposed. Keeping ParticleWorks’ statements general was an important design
decision we made for the significant reason that specificity decreases flexibility whereas
generality increases it.

By keeping ParticleWorks’ statements general, we provide engineers the flexibility to use
them in endless different creative ways, many of which we have not imagined yet. In
addition, VITASCOPE (which ParticleWorks extends) has a multi-layered extensible
model. In the first tier, animation statements designed to extend VITASCOPE may be
implemented (optionally) as a concatenation of core VITASCOPE language statements.
Subsequent tier extensions may use not only VITASCOPE’s core statements, but higher-
tier extension statements as well. This in effect allows us (or any other researcher) to
design and implement animation statements that describe specific types of fluid objects.
We explain this with an implemented example.

### 7.1 Precipitation

Precipitation such as rain and snow play an important role in construction. The
occurrence and intensity of such precipitation may affect outdoor working conditions
significantly, most often adversely. Visualizing occurring precipitation can therefore add
significant value to scientific construction process visualizations.

We concatenated ParticleWorks statements (first tier VITASCOPE extensions) to design
and implement two second-tier extension statements to the VITASCOPE language. These
statements, presented in table 2, allow engineers to use simple parametric text commands
to describe the type and intensity of existent precipitation on virtual construction sites.

<table>
<thead>
<tr>
<th>Table 2: Usage of Higher-Order ParticleWorks Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statement</strong></td>
</tr>
<tr>
<td>RAIN [Intensity]</td>
</tr>
<tr>
<td>SNOW [Intensity]</td>
</tr>
</tbody>
</table>
Each of these statements, when executed, in turn invoke a set of core ParticleWorks statements. A rain cloud, for example, is instantiated as a fuzzy object and placed at a high altitude on the virtual jobsite. The attributes of the cloud’s particle system are then set such that the emitted particles resemble rain drops, and their density resembles its intensity. Figure 12 presents the annotated pseudo-code that implements the RAIN statement.

```
RAIN [Intensity]
{
    FuzzyObject Cloud [Cover entire jobsite];
    Cloud.SetRate  [Intensity];
    Cloud.SetColor  [Blue];
    Cloud.SetSink  [Ground surface];
    Place Cloud [High above the jobsite];
}
```

**Figure 12: Concatenation of Animation Statements**

Precipitation of snow is implemented in exactly the same manner, except for the appearance specified to its particles. Figure 13 presents snapshots of animations with precipitation in progress.

**Figure 13: Describing Precipitation in Construction Process Visualization**
8. Future Work

This research can be extended in several interesting ways. The concept of particle systems can be further explored to describe several other processes common to construction. For instance, blasting operations are common in tunneling and quarrying. Visualizing explosions and the resulting debris caused by such operations is an interesting challenge that could be addressed using adaptations of the particle system concept. In particular, exploring methods to describe such processes in simple, parametric text statements calls for some very interesting work.

Particle systems may also be adapted to describe bounded, dynamic formations of generic simulation entities. Examples of such entities include vehicles traveling through congested construction work zones and people walking in crowded places. The reader will recall that particles may be represented in virtual worlds as any CAD model (e.g. car, truck, person etc.). Particle systems may thus present a unique technique for visually describing such bounded formations while simultaneously reducing the number of instructions necessary for their description.

Engineers are typically interested in the overall effect of congestion rather than any particular entity (e.g. vehicle or person). Techniques of describing random bounded group formations as particles can thus be of significant help in reducing the complexity and size of discrete-event simulation models that author such visualizations. Literature provides several examples of particle systems adapted to visualize such diverse things as flocking birds, herding sheep, and schooling fish (Reynolds 1987), in addition to pedestrian and road traffic (Schaefer et al. 1998). Designing methods to describe inter-particle avoidance and motion resulting from interaction with other particles are interesting research issues to explore in such adaptations.

Future research may also focus on improving the physics behind particle system dynamics. This includes considering additional physical world forces (e.g. adhesion, attraction, repulsion) during analysis and experimenting with improved, more accurate numerical integrations methods. In addition, improved methods of dynamic fluid analysis
may be adopted in the physical analysis computations. Methods such as those based on the Navier-Stokes equations perform accurate analysis of motions of viscous liquids in three dimensions (Stam 1999). The perpetual advances in computing power are certain to make this work feasible in the near future.

9. Summary and Conclusions

The presented research extends the state-of-the-art of scientific simulation-driven construction process visualization. The work puts in place the technology that engineers can use to automatically create dynamic, convincing visualizations of construction processes that involve unstructured, fluid, fuzzy materials.

The design of techniques to accurately animate the shape, form, motion and dynamic behavior of unstructured, fluid construction materials in 3D requires three key technologies:

1. Computationally feasible physical analysis to accurately describe the dynamics of fluid material volumes
2. Simple, parametric methods with semantics rich enough to convert discretely recorded animation descriptions into smooth motion
3. Efficient rendering procedures to represent the descriptions in 3D virtual worlds such that the results look visually convincing, move realistically, and can be animated in real-time.

Classical techniques of modeling fluid behavior are highly accurate but require very intensive computation. To achieve a compromise between visual realism, physical accuracy, and reasonable computational times, we treat fluid volumes as clouds of individual point masses whose dynamics are described using an elementary subset of Newtonian mechanics.

The approach has several advantages. First, it allows us to determine the finite subset of discrete static and dynamic components that contribute to the overall appearance and motion of arbitrary fluid material volumes. This guides the design of methods that
achieve smooth, continuous animation based on discrete information recorded at non-fixed time steps. This is critical since external visualization authoring processes such as discrete-event simulation models communicate with other processes only at discrete, but possibly random, sets of simulated time points.

In addition, this approach allows us to represent volumes of such dynamic, fluid construction materials in 3D virtual worlds using classical computer graphics techniques based on the particle system concept. The visually appealing results we obtain demonstrate that the approach is not only possible, but also quite effective.

10. References


Chapter 5
Practical 3D Animation of Multiply Articulated Construction Equipment

1. Background

Animation adds significant value to discrete-event simulation by facilitating the verification and validation of process models (Law and Kelton 2000). This is especially true in construction where typical decision makers are experts in their domain but are not generally proficient in simulation itself. In addition, visualizing modeled construction processes in 3D provides valuable insights into subtleties of planned operations that are otherwise non-quantifiable and non-presentable (see Chapter 9 for details).

Ongoing research efforts at Virginia Tech focus on designing automated, process simulation-driven methods to visualize modeled construction processes and the resulting products in dynamic 3D virtual worlds. The tangible outcome of this continuing work is the VITASCOPE visualization system. VITASCOPE is an acronym for VIsualizaTion of Simulated Construction OPErations. VITASCOPE is a user-extensible 3D animation language especially suited for visualizing modeled construction operations in smooth, continuous, dynamic 3D virtual worlds (see Chapters 2 and 3 for details).

VITASCOPE animates a modeled construction process by converting discrete pieces of quantitative information communicated at uneven time instants by a running simulation model into smooth continuous motion. In order to animate a modeled operation in 3D, a discrete event simulation model is instrumented to generate a time-stamped animation trace of events that occur during a simulation run using the syntax of VITASCOPE’s parametric animation language. VITASCOPE then post-processes the generated animation trace and depicts the modeled processes in 3D by instantiating and manipulating pre-created CAD models of the involved construction resources (equipment, materials, terrain etc.) in a virtual environment.
VITASCOPE’s parametric animation statements allow a simulation model to communicate the elemental motions involved in performing construction in a geometric transformation level parlance. Work involved in designing the VITASCOPE language identified geometric transformation-based elemental motions as the only appropriate level in the hierarchical taxonomy of construction processes at which a discrete-event simulation model can generally communicate a dynamic operation in 3D (see Chapter 2 for details). VITASCOPE’s primary motion-describing statements (e.g. MOVE, ROTATE, SCALE) each describe a single elemental motion that a construction resource undergoes as it performs work (e.g. A truck MOVEs along a haul road, the cab of a backhoe ROTATEs as it swings, a crane’s cable SCALEs as the hook is dropped or raised). A time-stamped sequence of an arbitrary number of such elemental motions communicated by a running simulation model describes a smooth, continuous 3D rendition of the construction operation being modeled.

1.1 Problem Description

The current state-of-the-art in scientific, simulation-driven construction operations visualization has two significant limitations in the interface between the 3D animation methods and discrete-event simulation models that use those methods to animate the processes they simulate. Existing methods of animating simulated construction operations:

1) Enforce that processes intended to be animated be modeled in significant, elemental motion level detail regardless of whether those details are relevant from the pure modeling and decision-making perspective.

2) Require simulation modelers to manually encapsulate geometric operational details about the modeled processes into simulation models to allow the schematic process models to intelligently communicate with the spatial 3D animation methods. Such geometric details are, in most cases, otherwise irrelevant in discrete-event simulation analyses.
1.1.1 Detailed Modeling for Animation

In discrete event simulation, the state of a model changes only at discrete, but possibly random sets of simulated time points (Schriber and Brunner 2001). These time points are typically the start or end of the model’s activities, and it is only then that a running discrete-event simulation model can communicate with other processes, or perform other actions such as perform output to an animation trace.

The consequence of this inherent discrete-event simulation characteristic and VITASCOPE’s elemental motion-describing animation vocabulary is that a simulation model intending to communicate a smooth, continuous animation in 3D must consist of individual activities that represent each elemental motion of the construction resources involved in the operation being modeled. This is necessary so that time-stamped animation statements describing the involved elemental motions can be communicated at the discrete time instants at which each of the corresponding individual activities start or end during a simulation run.

To illustrate this point, let us consider the example of a simulated structural steel frame erection operation using a crane. The top half of figure 1 presents a discrete-event simulation model created in STROBOSCOPE (Martinez 1996) that models the processes involved in erecting the steel frame in significant sub-task level detail. The lower half of the figure presents a VITASCOPE animation trace segment that when processed will depict the erection of a near column on the second tier in the frame.

The numbers in the callout boxes pointing to the model’s activities and the trace indicate which animation statements are communicated at the time instants at which instances of the respective activities start in a simulation run. In an animation trace that describes the construction of the entire steel frame, each of these statements will be communicated to the trace several times (depending on how many steel shapes are placed) albeit at different simulation time instants and with different numerical parameters for the statements’ arguments (depending on the particular steel shape being placed).
TIME 6770.00;
ATTACH Shape65 TheHook (0, -0.5, 0);
TIME 6770.00;
SCALE TheCable (0, -30.00, 0) 15.00;
SLIDE TheHook (0, 30.00, 0) 15.00;
TIME 6785.00;
TGTROTATE TheBoom HOR 151.93 20.00;
TGTSLIDE TheTrolley (17.00, 0, 0) 20.00;
TIME 6805.00;
TGTSCALE TheCable (1, 16.30, 1) 15.00;
TGTSLIDE TheHook (0, -16.30, 0) 15.00;
TIME 6805.00;
ROTATE Shape65 HOR 28.07 15.00;
TIME 6830.00;
DETACH Shape65;
PLACE Shape65 AT (0.00, 18.00, 0.00);
HORIZORIENT Shape65 0.00;
TIME 6830.00;
TGTSCALE TheCable (1, 14.00, 1) 10.00;
TGTSLIDE TheHook (0, -14.00, 0) 10.00;
TIME 6840.00;
TGTROTATE TheBoom HOR 135.00 5.00;
TIME 6845.00;
TGTROTATE TheBoom HOR 180.00 10.00;
TGTSLIDE TheTrolley (20.00, 0, 0) 10.00;
TIME 6855.00;
TGTSCALE TheCable (1, 34.55, 1) 10.00;
TGTSLIDE TheHook (0, -34.55, 0) 10.00;
...

Figure 1: Communicating 3D Animations with Elemental Motion Statements
The primary objective in using simulation to model construction processes is to evaluate and compare the outcome of alternative decisions (e.g. choosing the best construction technique and method, choosing the appropriate construction equipment, identifying the optimum allocation of labor at different levels, establishing operating logic strategies) to make the best choice. The level of detail at which a construction process is modeled depends on the specific purpose of the model and the amount and validity of the data available for the analysis.

For instance, in the case of the steel erection operation presented above, an engineer may choose to model the processes at a significantly lower level of detail if s/he deems that the insights provided by the simulation results are sufficient to make the decision being contemplated and/or if detailed statistical input data is not available for the analysis. Such a possible simulation model is presented in figure 2. This model simulates the same steel erection operation presented above albeit at a much lower level of detail by encapsulating all the involved sub-tasks into a single large activity.

![Figure 2: Modeling Steel Erection Processes at Task Level Detail](image)

A simulation model at this typical low level of detail (relatively speaking) is however unable to generate a 3D animation trace for lack of sufficient discrete time instants in a simulation run at which each of the involved tasks (e.g. Lower hook, Attach sling) could be communicated using elemental motion statements (e.g. SCALE cable, ATTACH shape). In order to be time stamped and communicated to an animation trace, each elemental motion-describing animation statement requires the occurrence of a separate
discrete event during a simulation run. A simulation model thus needs separate activities for each elemental motion so that discrete events are generated when those activities start or end at which time instants the running model can time stamp and communicate the respective animation statements.

In the case of the simulation model presented in figure 2, the only discrete events that can occur during a simulation run are the start and end of the main activity “Perform Lift”. The model generates no events and encapsulates no data about the sub-tasks (e.g. Lower hook, Attach sling, Swing to target) involved in performing each lift. At this typical level of modeling detail, there are not enough discrete events or sufficient operational data to generate animation statements that can describe the modeled processes in a smooth, continuous manner in 3D.

The fallout of this situation is that in order to be animated in 3D, simulation models must compulsorily be created at a significantly high-level of detail regardless of whether the detail is essential to the analysis from the decision-making point of view. In an earthmoving operation, for instance, a loading site must be modeled using separate activities for each elemental motion of the loader (Swing empty, Lower boom, Scoop dirt, Lift boom etc.) when a single activity (e.g. Load Dirt) in most cases is sufficient to model a loader otherwise (Kamat and Martinez 2001). Simulation models are often created at high levels of abstraction where each building block (i.e. activity) in the model represents a basic construction task (e.g. Load Dirt) and not elemental motions (Swing empty, Lower boom, Scoop dirt, Lift boom etc.).

1.1.2 Geometric Operational Details
In addition to enforcing detailed sub-task level modeling for animation, current methods of animating simulated operations in 3D require discrete-event simulation models to encapsulate several pieces of spatial, geometric operational data that a modeler would choose to otherwise ignore in the absence of 3D animation. A pertinent example is the amplitude and direction of each of the elemental motions a construction resource undergoes as it performs work simulated by a model’s corresponding sub-tasks. We
return to the structural steel frame erection operation presented in figure 1 to illustrate
this point.

VITASCOPE’s elemental-motion describing animation statements are parametric in
nature. While the same statements (e.g. MOVE, ROTATE, SCALE) may appear in an
animation trace several times, the target CAD object to manipulate and the numerical
value of the statements’ arguments are generally different in each statement. The
numerical values of the statement arguments represent the amplitude and direction of the
elemental motion the statement describes.

For instance, the fourth statement in the animation trace segment in figure 1 (SCALE
TheCable (0,-30.00,0) 15.0;) indicates that the cable of the virtual crane must
scale by 30 factors (indicated by -30 in the Y dimension) in 15 simulation time units so
that the virtual hook attached to the cable can reach the virtual steel shape to be picked.
The seventh statement (TGTROTATE TheBoom HOR 151.93 20.0;) similarly
indicates that the boom of the virtual crane must rotate until it is at a heading of 151.93
degrees (the target rotation) in the horizontal plane in 20 simulation time units so that the
loaded crane’s hook is above the point where the virtual shape is to be installed.

A discrete-event simulation model, even when created with a high level of detail as that
presented in figure 1, does not encapsulate and cannot automatically generate such spatial
operational information that might help quantify the amplitude of each elemental motion
virtual construction resources undergo as they perform work that corresponds to the
model’s activities. Discrete-event simulation models are schematic process models
having no inherent spatial knowledge about the processes they simulate.

In order to animate modeled processes in 3D then, all spatial numeric data required to
quantify the amplitudes and directions of the involved elemental motions of virtual
construction resources must be manually input to a simulation model. Discrete-event
simulation models must be explicitly made aware of the 3D virtual environment in which
the modeled processes will be animated. This cumbersome, time-consuming, and often
impossible activity is essential so that simulation models communicate in an informed manner with 3D animation methods and generate a coherent, spatially accurate animation trace during a simulation run.

1.2 Main Contribution

The research this paper presents directly addresses both the issues described above. We explore several original concepts to design and implement advanced 3D animation methods to visualize construction processes modeled at typical higher levels of abstraction (e.g. figure 2) in smooth, continuous, 3D virtual worlds. The designed methods allow discrete event simulation models to communicate animation instructions in a higher-level, construction work-like terminology without having to indicate (or be aware of) the elemental motions of the involved resources and their amplitudes.

Using principles of forward and inverse kinematics from robotics literature where appropriate, we designed and implemented generic pieces of multiply-articulated virtual construction equipment. Discrete event simulation models can configure and instantiate specific pieces of such equipment and instruct them to perform common construction tasks using simple, parametric statements of text. Once instructed to perform specific tasks (e.g. Load dirt), these “smart” pieces of virtual equipment (e.g. Backhoes) automatically decipher the best combination and amplitudes of the elemental motions their components (e.g. Boom, Bucket, Stick) must undergo to accomplish those tasks.

The designed animation methods build upon the existing state-of-the-art. The animation methods that allow simulation models to communicate higher level construction tasks to virtual pieces of multiply-articulated equipment are designed and implemented as an add-on to the VITASCOPE visualization system. This extensible add-on, named KineMach, currently implements a generic highway dump truck, a tower crane, a crawler-mounted lattice boom crane, and a crawler-mounted backhoe.

By designing animation methods to visualize simulation models created at typical, higher, task-level detail and relieving engineers from manually encapsulating
unnecessary (from simulation point of view) spatial details into simulation models, the presented work greatly simplifies the process of animating modeled operations in 3D thus making 3D animation more accessible and practical.

2. Challenges

The design of automatic, simulation-driven methods to communicate basic construction tasks to multiply-articulated virtual pieces of construction equipment using simple, parametric text statements presents numerous interesting challenges. Typical simulation models created using basic construction tasks as building blocks, when run, only generate discrete events at the time instants those basic tasks begin or end. The information such models can communicate to an animation trace is thus limited to the time instants at which instances of each of the constituent basic tasks begin or end.

When elemental sub-tasks involved in performing constituent basic tasks are not explicitly modeled, discrete event simulation models do not encapsulate and therefore cannot communicate information about when those elemental sub-tasks begin or end. This information is however critical to 3D animation because geometric transformation based elemental tasks are the necessary building blocks for describing the motions of various construction resources that perform construction virtually on the computer.

In addition to time instants at which constituent elemental tasks begin and end, methods of animating simulated processes in 3D require several other pieces of spatial information that are necessary to describe and depict the accurate motion of virtual construction resources as they perform work. Even when construction is modeled at the elemental sub-task level, such spatial information about the virtual construction environment and resources that will depict the modeled processes in 3D must be manually input into simulation models. Simulation models, in turn, use that information in communicating with the 3D animation methods. In models constructed at the basic task level, however, such spatial information, even if manually input to a simulation model, is irrelevant because the running simulation model lacks sufficient discrete events (i.e. simulation time instants) at which to communicate that information to the 3D animation methods.
The basic question we address in this work, then, is how to achieve realistic, smooth and dynamic continuous motion of virtual construction resources, based only on discrete, uneven-spaced pieces of information about when basic construction tasks begin and end. In addition, the methods that allow simulation models to communicate basic task level information to animation facilities must themselves be simple. Simplicity, in this context means that the syntax of the methods and the values sought by their parameters must both be within the authoring capabilities of simulation models.

The issues we address in this work are thus threefold. First, we design simple parametric text methods that simulation models can use to communicate basic construction tasks to virtual pieces of construction equipment in a high-level, construction work-like terminology. Second, we investigate techniques to compensate for simulation models’ lack of information about the sub-tasks that comprise each communicated basic task. Finally, we explore techniques to alleviate simulation models from encapsulating several spatial details about the modeled processes that in the absence of animation, would otherwise be ignored in simulation analyses.

3. Initiative

In attempting to address these issues, we found most promise in a technology based on the concept of Inverse Kinematics that is widely used for controlling industrial robotic manipulators. Engineers who design and implement computer programs to control robots used in the manufacturing industry are concerned with moving a robot’s gripper (i.e. the tool at the end of the robot’s arm) between specific desired positions and orientations (Korein and Badler 1982). A robotic manipulator is an articulated structure composed of a kinematic chain of linkages. The joints in a robot’s arm are controlled by servomotors that change the length of its various links or the orientation between them so that each link (particularly the end tool) is positioned as required defining a specific articulation for a particular task.

Robotics engineers can greatly benefit by being able to simply say where a particular robot’s tool must work and have some procedure compute how each link and joint in the
robot’s arm is to be arranged to achieve that goal (Madhavapeddy and Ferguson 1998). To study that problem, researchers in robotics have devoted a significant amount of work in investigating the problem kinematically.

3.1 Kinematics

Kinematics studies the geometric properties of the motion of points without regard to their masses or to the forces acting on them (McCarthy 1990). A set of points in which the distance between any pair never varies is called a rigid body or a rigid link. The position of such a rigid link in Euclidean space is fully qualified by six dimensions; three translations (i.e. the X, Y, and Z co-ordinates) and three rotations (i.e. the Roll, Pitch, and Yaw). A set of links connected by joints that constrain their relative movement is called a kinematic chain. Common industrial robots that are comprised of a set of links with one end attached to a rigid base are defined as open kinematic chains (McCarthy 1990, Craig 1989). A closed kinematic chain (also called a mechanism), on the other hand, may be attached to a rigid base in more than one place.

3.1.1 Forward Kinematics

A kinematic problem can be studied and analyzed in one of two ways. The case where all joint angles of a particular chain articulation are manually specified is termed Forward Kinematics (FK) (Watt and Watt 1992). In FK, the final shape of the articulated chain is dependent on the angles explicitly specified for each of its links. This can be mathematically stated as $X = f(\Theta)$, where $X$ is the resultant position and orientation vector of the free end of the chain (called the end effector) and $\Theta = (\theta_1, \theta_2, ..., \theta_n)$ is the state vector that describes the configuration of the chain by encapsulating the position and orientation vectors of each constituent link. In FK analysis, the configuration of all link joints is specified explicitly. The configuration of the end effector is thus determined indirectly by descending the tree of the structure and accumulating all leading link transformations to arrive at the only possible solution for the end effector.
3.1.2 Inverse Kinematics

Inverse Kinematics (IK), on the other hand, is the reverse of FK and is of significantly more interest to robotics engineers. Given the desired position and orientation of the end effector in a kinematic chain, IK is concerned with computing what angles and/or lengths each of the link joints need to be in to achieve that target (Zhao and Badler 1994, McCarthy 1990). Using the same terminology described above, the IK problem can be mathematically stated as $\Theta = f^{-1}(X)$.

Due to its non-linear nature, IK is significantly more complex than FK because there are usually multiple or infinite solutions to the problem especially when the number of links in a chain is large. In many cases, there are no feasible solutions at all when it is impossible for the linked chain to reach the target. This occurs when the sum of the lengths of a chain’s links is less than the distance between the target position and the rigid base of the chain (called the terminator).

3.2 Analogy to Robotics

In animating simulated construction processes in 3D, construction engineers have a very similar problem to that of robotics engineers when we try to control virtual pieces of multiply articulated construction equipment such as backhoes and cranes based on extremely limited information communicated by discrete event simulation models. Conceptually, each construction resource (piece of equipment or human craftsman) on a virtual construction site can be thought of as a robot performing a certain construction task. Just as robots perform specified tasks using real resources, virtual pieces of equipment and craftsmen perform construction on the computer using virtual resources.

The striking analogy between the two arises from the fact that both robots and virtual entities do not have any intrinsic knowledge about performing assigned tasks and need to be programmed in order to perform particular motion sequences. In terms of information needs, therefore, virtual pieces of construction equipment are very similar to industrial robots. The two may differ in context (real vs. virtual) and shape, but in both cases, engineers are basically interested in trying to manipulate a multiply articulated structure.
(i.e. a kinematic chain). As construction engineers animating modeled construction processes, we are interested in manipulating virtual pieces of construction equipment between specific desired positions on the computer to create an illusion (i.e. visualization) of construction work being performed.

Existing methods of animating simulated construction processes are based on FK technologies. At that start of each modeled elemental task (e.g. Lower boom), a simulation model must explicitly communicate the elemental motion (i.e. the target position and/or orientation) that a component (e.g. Boom) on the corresponding virtual piece of equipment (e.g. Backhoe) must undergo to visually depict the elemental task. Therefore, in cases where simulation models have no separate activities to represent elemental tasks and have no spatial information about the corresponding virtual equipment’s geometry, it is impossible to communicate a 3D animation trace.

In order to animate construction processes modeled at higher levels of abstraction (i.e. basic tasks), we need animation methods that accept instructions at the basic task level and then automatically figure out the elemental motions of the corresponding virtual pieces of equipment to visually depict those tasks. In other words, we need animation methods that can allow a simulation model to simply say (i.e. communicate) what basic task (e.g. Load dirt, Perform lift) a particular virtual piece of equipment must perform and have some procedures automatically compute how each link (i.e. component) in the equipment’s kinematic chain is to be arranged at each pose (i.e. articulation) necessary to accomplish that task.

It is obvious that IK technologies are directly relevant to our problem and present an ingenious technique of addressing it. For instance, by simply knowing where in the virtual world to dig (i.e. the target position), an IK procedure can compute the relative angles at which a virtual backhoe’s components (i.e. Cab, Boom, Stick, Bucket) should be such that the virtual bucket (the end effector in the kinematic chain) reaches the target position. After automatically computing the specifics of a particular required articulation,
existing FK based animation methods can be invoked to explicitly move the virtual equipment’s components to that articulation.

An IK based approach thus has several advantages. First, by automatically computing the articulation of virtual pieces of equipment necessary to accomplish basic construction tasks, simulation models are relieved of 1) necessarily being modeled at the detailed elemental motion level, and 2) encapsulating otherwise ignorable detailed geometric information about virtual pieces of construction equipment. Second, the approach facilitates conspicuous reuse of existing animation methods. As the following discussion will clarify, adopting IK based techniques to address this problem only requires the introduction of an additional layer of abstraction in the interface between simulation models and existing FK based methods of animating modeled processes.

4. Technical Approach

IK is a goal-directed process. When an end-effector is assigned a desired goal, it must travel towards the goal (assuming it is reachable) through a series of one or more iterative steps. The adopted IK procedure is then responsible for computing the transformations of the remaining kinematic chain joints. Most pieces of articulated construction equipment can be viewed as open kinematic chains comprised of a finite number of links that are joined together with simple prismatic or revolute joints, each of which typically allows only one degree of freedom to the next link down the hierarchy.

A prismatic joint is a sliding connection between two links that exhibit one or more degrees of translational freedom (i.e. the child link can slide with respect to the parent link in one or more dimensions). An automobile shock absorber, for instance, is a simple prismatic joint exhibiting only one degree of translational freedom. A revolute joint between two links, on the other hand, allows a child link to rotate around one or more axes with respect to its parent link. A door hinge, for instance, is a simple revolute joint that describes only one degree of rotational freedom between its two links.
The process of decomposing and viewing pieces of construction equipment as kinematic chains will now be obvious to the reader. A backhoe, for instance, can be described as a kinematic chain where pairs of components (i.e. cabin and the crawlers, boom and cabin, stick and boom, bucket and stick) are joined together by simple revolute joints, each describing only one degree of rotational freedom. When the crawlers are stationary, a backhoe’s links can be analyzed as an open kinematic chain i.e. the crawlers (the base link) can be assumed to be attached to a fixed base (the earth in this case). The leading edge of the backhoe’s bucket is then the end effector of the kinematic chain that must travel to certain points depending on where to dig (or dump).

Components in a tower crane can similarly be viewed as a collection of links joined together by revolute as well as prismatic joints. While the connection between the crane’s swinging boom and the tower is simple revolute, the joint between the boom’s arm and the trolley that suspends the crane’s hook is a simple prismatic joint. In addition, even though the crane’s hook is freely suspended from the trolley with cables, it can (for IK purposes) be viewed as having been joined to the trolley using another simple prismatic joint. In a tower crane’s case, the hook is the obvious end effector while the fixed base of the tower is the terminator.

4.1 Selecting Technique to Implement Inverse Kinematics

Several techniques used to implement IK are documented in the literature. The complexity of IK has perennially motivated research for computationally efficient algorithms. IK algorithms can be broadly classified into analytical and numerical methods. Analytical methods can find all possible solutions to an IK problem and are desirable for their speed and the exactness of their computed solutions. However, for most non-trivial kinematic problems, an exact solution(s) may not be possible and analytical methods prove inflexible (Tolani et al. 2000, Lander 1998, Madhavapeddy and Ferguson 1998).

Numerical IK methods, on the other hand, iteratively converge to a single solution based on an initial guess. Small adjustments are made to the chain’s joints to solve the IK
problem in a series of steps. The process is stopped when the chain’s end effector reaches the goal within some specified tolerance. The main advantage of numerical IK methods is that they are more general and can be used even in cases where the kinematic problem is ill-posed (Tolani et al. 2000). Literature documents several numerical techniques used for solving IK problems. Examples include 1) Application of the Newton-Raphson algorithm for solving systems of non-linear equations, 2) Converting the IK problem into a differential equation, and 3) Recasting the IK problem into a non-linear optimization problem (Welman 1993).

In the presented work, we adopted a general, computationally efficient, and relatively easy-to-implement IK algorithm based on non-linear optimization. This algorithm (called the Cyclic-Coordinate Descent) was first outlined in robotics literature and attempts to minimize the distance between a specified target and the end effector of the specified kinematic chain. The algorithm accomplishes this by iteratively adjusting the chain’s joints in a way that minimizes the distance.

4.2 Cyclic-Coordinate Descent Method

The Cyclic Coordinate Descent (CCD) method is an iterative heuristic search technique which attempts to minimize position and orientation errors between an end effector and its desired position by varying one joint variable at a time (Wang and Chen 1991). Each iteration involves a single traversal of the kinematic chain from the most distal link inward towards the chain terminator. Each joint variable $\theta_i$ is modified in turn to minimize an objective function. As a solution is obtained at each joint, the position and the orientation of the end effector are immediately updated to reflect the change. As a result, the minimization problem solved at any particular joint in the chain incorporates the changes made to more distal joints during iteration.

The relatively simple mechanics of the CCD IK method are best explained with a practical example which we present next. The mathematical formulation of the IK problem using the CCD non-linear optimization method, however, makes some interesting reading. We refer the mathematically inquisitive reader to Welman (1993)
who presents a concise mathematical formulation of the minimization problem encountered in the CCD IK method.

4.2.1 Basic Computation Scheme
Consider a virtual hydraulic backhoe excavating dirt from a pit. A cross-sectional side view of such a configuration is presented at the top of figure 3. Let us assume that the point marked “X” is the location where the virtual backhoe must “dig” in its next pass. For simplicity in introducing the algorithm, let us also assume that the backhoe’s cabin is already aligned to dig at point “X”. In other words, as the lower half of figure 3 presents, point “X” lies on the straight line that connects the backhoe’s boom, stick, and bucket when viewed from the top. In order to dig at point “X” then, only the backhoe’s boom, stick, and/or bucket need be manipulated. The goal of the CCD IK algorithm, then, is to minimize the distance between the leading edge of the bucket and point “X” by rotating the virtual backhoe’s bucket, stick, and boom iteratively.

In order to solve this minimization problem using the CCD algorithm, we start with the last link in the kinematic chain (i.e. the bucket itself). First, we create a vector from the root of the in-context link (B) to the current end effector position (E). We then create another vector from (B) to the desired target position (X). This is graphically presented in figure 4 (a). In this iteration, the aim of the algorithm is to compute the angle (A) by which the vector (BE) must be rotated in order to coincide with vector (BX).

The angle (A) can be computed in a straightforward manner using a standard result from elementary vector algebra. In particular, the dot product for the two vectors (BE and BX) can be defined by $BE \cdot BX = \|BE\| \|BX\| \cos A$. By computing the inverse cosine of the vector dot product, we easily obtain the magnitude of the angle (A) between the vectors. In order to coincide vector (BE) with (BX), however, we need another piece of information in addition to the magnitude of angle (A). In particular, we need to know the direction of rotation about point (B).
We compute that by using another elemental result from vector algebra. The cross product of the two vectors (BE and BX) is a third vector that is always perpendicular to both (BE) and (BX). The cross product vector thus represents the axis in 3D space about which to rotate the source vector (BE). Knowing the amount of rotation and the direction in which to apply it, we immediately modify the in-context link. This is presented graphically in figure 4 (b).

The algorithm then moves one link up the kinematic chain (the backhoe’s stick in this case) and repeats the procedure. The result of applying this next iteration at joint (S) is presented graphically in figures 4 (c) and (d). This procedure continues up the chain until
the base link in the chain is reached, after which the whole process is repeated, starting at the last link again.

![Figure 4: Cyclic Coordinate Descent Iterations](image)

The iterations continue until either the end effector (E) is close enough to the desired position (X) or a specified number of iterations have been performed. Specifying a limit for the maximum number of iterations is necessary to stop the computation in cases where the target position is unreachable. Note that in this particular configuration, the algorithm will not attempt to rotate the cabin as it is already assumed to be aligned with the target position. In addition, the crawlers represent the fixed terminator link in the chain and are not manipulated during the iterations.

In the presented example, all the involved joints are simple revolute. In cases when the kinematic chain consists of prismatic joints (e.g. in a virtual tower crane), the algorithm
proceeds in a very similar way except that an in-context link is attempted to be translated (and not rotated) to coincide the end-effector with the required target position.

### 4.2.2 Enforcing Geometric Constraints

A particular problem with using the CCD method (and IK techniques in general) to animate virtual pieces of construction equipment is the constraining of angles between specific joints to prevent unrealistic behavior. For instance, all the joints in a backhoe’s kinematic chain are simple revolute joints exhibiting only one degree of freedom each. The joint between a backhoe’s cabin and its crawlers allows the cabin to rotate in only one plane (i.e. about the Y axis in a right-handed co-ordinate system).

The revolute joints between the other pairs of components (cabin and boom, boom and stick, stick and bucket) similarly restrict rotation to only about the local Z axis in the parent link’s local co-ordinate frame. To prevent the visual depiction of any unrealistic behavior that can contradict the structural integrity of real pieces of construction equipment, joint constraints must be considered and implemented. This is achieved by imposing appropriate limits on the degrees of freedom of each individual joint.

In the absence of any explicitly specified restrictions, the CCD algorithm, by its very nature, assumes each joint in a kinematic chain to be an unrestricted ball-and-socket type joint that allows unlimited angular motion between the links in any direction. At each iteration, the CCD algorithm computes an angle of rotation and an arbitrary axis in 3D space about which to apply that rotation so that a source vector coincides with the target vector. Applying that rotation about the computed axis may however violate the structural integrity of a corresponding real piece of construction equipment.

For instance, in the example presented above, assume that the cabin of the backhoe was not aligned to dig at the target position (X). This is presented graphically in figure 5 (a). In the very first iteration, then, the CCD algorithm will suggest that the backhoe’s bucket twist with respect to the stick so that it aligns itself with the target position (X). This is presented graphically in figure 5 (b). Applying that rotation obviously violates the
structural integrity of a real backhoe that allows a bucket to rotate only about the Z axis in the stick’s local co-ordinate frame. We correct such discrepancies by enforcing geometric constraints after each CCD iteration.

![Figure 5: Violation of Geometric Constraints](image)

Geometric constraints are relatively easy to apply in the CCD method because each joint is handled as a single analytical geometry problem. As a result, any geometric constraints that exist at each joint can be simply figured into the problem. After every CCD iteration and before the computed rotation is applied to the in-context joint, we perform a test to determine if the suggested rotation is outside the joint’s limits. If it is, then the joint rotation is clamped to those limit angles. The rest of the joints (in later iterations) are then used to satisfy the problem. For instance, in the example presented above, the bucket is restricted from twisting about the stick and the desired rotation necessary to align with the target is achieved when processing the cabin link which, in fact, is allowed to rotate about the horizontal plane with respect to the crawlers.

In order to enforce joint limits, we convert each CCD suggested rotation (represented as an arbitrary axis and a rotation amount) into its axial components (i.e. X, Y, and Z components) using Euler’s rotation theorem (Goldstein 1980). We then check if the suggested rotation’s axial components violate a piece of equipment’s structural integrity in any plane of rotation. For instance, in the backhoe bucket’s case, we check if the suggested rotation has any non-zero X and Y components. If any such undesirable
rotation components are suggested, they are either zeroed out or clamped to the extreme values as appropriate.

For instance, while any X and Y components in a backhoe bucket’s suggested rotation would be zeroed out, the rotation component about the Z axis (permissible rotation axis) would be clamped if the suggested rotation causes it to rotate beyond that allowed by a real backhoe’s hydraulics. The adjusted axial rotation components are then converted back into a single axis-angle rotation and applied to the in-context link in the current iteration. Geometric constraints at prismatic joints are similarly enforced by limiting the translation components of the in-context link appropriately. This augmented CCD algorithm thus keeps individual joints in a kinematic chain from rotating (or translating) into a position that is physically impossible for real pieces of construction equipment.

4.2.3 Incorporating Damping

At each iteration, the CCD method will attempt to rotate an individual joint to any angle necessary to satisfy the IK problem at that step. Since the algorithm starts from the last joint in the kinematic chain and works up the hierarchy, the method tends to favor later joints that are deep down the kinematic hierarchy. This bias, in most cases, may not look natural, especially when manipulating virtual pieces of construction equipment. We limit such effects by incorporating damping into our computation.

During each iteration, we limit the amount of rotation that a joint can undergo to a specified empirical value that is often dependent on the type of equipment and the joint in context. This limit is enforced after geometric constraints have been applied and before the in-context link is updated. By incorporating damping, we essentially trade computational efficiency for more realistic effects. While the number of CCD iterations required to solve an IK problem increases as damping is enforced, the resulting solution presents much more smoother and realistic effects. We now describe our implementation of the CCD IK algorithm. The implementation is a powerful tool that allows engineers to instantiate and manipulate “smart” pieces of virtual construction equipment using simple parametric text statements in a higher-level, construction work-like terminology.
5. KineMach

Our implementation of IK techniques to animate simulated construction processes is a powerful tool that allows discrete event simulation models created at typical high levels of abstraction to communicate a smooth, continuous operation in 3D using a correspondingly high, construction task-level vocabulary. This tool, called KineMach, is implemented as an extension (add-on) to the VITASCOPE visualization system.

5.1 Parametric Animation Methods

KineMach implements “smart”, generic pieces of virtual construction equipment and provides simple parametric text statements that can be used to issue task-level instructions to that equipment to visually depict the performance of construction work. Currently implemented generic pieces of equipment include a tower crane, a crawler mounted lattice boom crane, a crawler mounted backhoe, and a highway dump truck. These virtual pieces of equipment are generic in that more than one piece of a particular equipment (having different dimensions, if necessary) can be instantiated and independently manipulated on the same virtual construction site.

Tables 1 and 2 present selected parametric statements KineMach implements. In particular, only statements used to instantiate and manipulate virtual crawler mounted backhoes (Table 1) and crawler mounted lattice boom cranes (Table 2) are presented. KineMach statements that implement generic tower cranes and dump trucks are similar in spirit and are excluded from this presentation in the interest of space. The parametric statements KineMach presents embody a construction work-like terminology making them very readable. Their meaning and significance should be obvious to the reader. The detailed usage of all statements is presented in appendix H.

KineMach’s statements are designed to represent the common construction tasks that the corresponding real pieces of construction equipment perform on real construction sites. Many statements are designed using standard, documented, and commonly used terminology. For instance, in the case of the implemented generic cranes, most KineMach statements have a direct one-to-one correspondence with standard crane hand signals.
used in real crane operations. Few other statements (e.g. PutThatThere) represent an improvised (albeit equally readable) terminology that we expected would be of particular utility in addressing the current problem.

Table 1: KineMach Implemented Backhoe Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKHOE [HoeName] [CrawlerWidth];</td>
<td>Instantiate, name, and size a virtual backhoe</td>
</tr>
<tr>
<td>[HoeName].DigAt [Source] [OperTime];</td>
<td>Instruct an instantiated hoe to dig at a specified source in a given time</td>
</tr>
<tr>
<td>[HoeName].DumpAt [Sink] [OperTime];</td>
<td>Instruct an instantiated hoe to dump at a specified sink in a given time</td>
</tr>
<tr>
<td>[HoeName].DigHereDumpThere [Source] [Sink] [OperTime];</td>
<td>Instruct an instantiated hoe to perform a full loading pass in a given time</td>
</tr>
<tr>
<td>[HoeName].Travel/BackUp [Trajectory] [OperTime];</td>
<td>Instruct an instantiated hoe to travel forward or backward on a specified trajectory in a given time</td>
</tr>
</tbody>
</table>

5.2 Animation Schema

KineMach’s intention is to be able to animate modeled construction processes in 3D at the same smooth, continuous level detail afforded by existing animation methods. The merit behind KineMach’s design, however, is that the same level of detailed animation is achieved with significantly lower operational details being communicated by authoring simulation models themselves.

KineMach thus essentially describes an additional layer of abstraction in the interface between discrete event simulation models and existing methods of animating operations in 3D. This layer of abstraction automatically generates the significant amount of elemental, operation-level details that would otherwise have to be communicated explicitly by simulation models.
Table 2: KineMach Implemented Lattice Boom Crane Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAWLERCRANE [CraneName] [CrawlerWidth] [BoomLength];</td>
<td>Instantiate, name, and size components of a virtual crawler-mounted crane</td>
</tr>
<tr>
<td>[CraneName].HookAt [TargetPosition] [OperTime];</td>
<td>Instruct an instantiated crane to move its hook to a specified position in a given time</td>
</tr>
<tr>
<td>[CraneName].PutThatThere [ObjName] [TargetPosition] [OperTime];</td>
<td>Instruct an instantiated crane to perform a full lift in the given operating time</td>
</tr>
<tr>
<td>[CraneName].RaiseBoom/LowerBoom [Amount] [OperTime];</td>
<td>Instruct an instantiated crane to raise or lower its boom by a specified amount in a given time</td>
</tr>
<tr>
<td>[CraneName].Travel/BackUp [Trajectory] [OperTime];</td>
<td>Instruct an instantiated crane to travel forward or backward on a specified trajectory in a given time</td>
</tr>
<tr>
<td>[CraneName].SwingBy/SwingToward [Amount] [OperTime];</td>
<td>Instruct an instantiated to swing by a specified amount or towards a specific object in a given time</td>
</tr>
<tr>
<td>[CraneName].HoistBy/LowerBy [Amount] [OperTime];</td>
<td>Instruct an instantiated to hoist or lower its hook by a specified amount in a given time</td>
</tr>
</tbody>
</table>

Figure 6 graphically presents the relationship between discrete event simulation models, existing methods of animating simulated processes, and KineMach. A running simulation model, which would otherwise have to communicate elemental motion-level animation instructions, can now communicate with KineMach in a higher construction task-level vocabulary. KineMach deciphers the communicated instructions and uses IK techniques to compute the elemental motions involved in performing the requested tasks. KineMach then invokes existing elemental motion-describing animation methods to depict the performance of the requested construction tasks in 3D. We return to the crane example presented earlier in figures 1 and 2 to further elucidate this relationship.
A discrete event simulation model created using high level construction tasks as building blocks (e.g. figure 2) can now communicate instances of those tasks to KineMach in the pertinent piece of equipment’s vocabulary. For instance, each time a steel shape is placed (i.e. the activity “Perform Lift” takes place), the simulation model communicates a KineMach statement (e.g. Crane1.PutThatThere Shape65 (0,18,0) 60;) to the animation trace requesting the instantiated virtual crane to perform the task.

![Figure 6: KineMach Animation Schema](image)

When KineMach processes each such statement, it first computes the elemental crane motions necessary to accomplish the task and apportions the total task time (communicated by the simulation model) to the individual elemental motions using proportions determined empirically. KineMach then generates elementary motion-describing animation statements in the VITASCOPE language and forwards them to the
visualization engine to graphically depict the operation. Other pieces of KineMach implemented equipment work in much the same way.

6. Generic Extensible Approach

KineMach implements generic pieces of virtual equipment. The statements KineMach exposes allow multiple pieces of the same equipment to be instantiated and independently manipulated on a virtual construction site. This, for instance, allows two (or more) backhoes to operate on different virtual loading sites when animating modeled earthmoving operations.

In addition, the statements that instantiate the virtual pieces of equipment are themselves parametric. Two instantiated pieces of the same equipment (e.g. two crawler mounted cranes) need not be of the same physical dimensions. Each crane could have, for instance, a boom of different dimensions (and capacity). Instantiated backhoes, tower cranes, and trucks could also be similarly defined with different dimensions for their components. Figure 7 presents an animation trace with statements that instantiate similar (but differently sized) pieces of equipment. The visual representation of that equipment can be observed from the lineup in the snapshot below.

Parameterization of this nature is particularly useful because equipment pieces of several different sizes and capacities (e.g. different models of backhoes) can be instantiated and manipulated generically based on the type of machines used in the underlying simulation analyses. When particular pieces of equipment are instantiated with specific dimensions, KineMach automatically scales and assembles appropriate CAD models of the equipment components to create a machine of the specified dimensions.

KineMach is also designed to be extensible. Currently implemented equipment pieces (i.e. backhoe, tower crane, crawler mounted crane, dump truck) represent but a small portion of the entire equipment fleet used in performing construction. KineMach is thus designed to be extended by implementing new generic (e.g. wheel loaders) or specific (e.g. a particular backhoe model) pieces of equipment. This allows KineMach’s
equipment fleet to incrementally evolve over time through the collective effort of several other researchers and engineers.

```
LOADADDON KineMach;
CRAWLERCRANE Crane1 4 20;
CRAWLERCRANE Crane2 6 35;
BACKHOE Hoe1 4;
BACKHOE Hoe2 6;
Crane1.PlaceAt (0,0,10);
Crane2.PlaceAt (0,0,0);
Hoe1.PlaceAt (0,0,-10);
Hoe2.PlaceAt (0,0,-20);
```

- Initialize the KineMach Add-On
- Instantiate Crane1 with 4m Wide Crawlers and a 20m Boom
- Instantiate Crane2 with 6m Wide Crawlers and a 35m Boom
- Instantiate Hoe1 with 4m Wide Crawlers
- Instantiate Hoe2 with 6m Wide Crawlers
- Place the Instantiated Equipment Pieces in the Scene

**Figure 7: Instantiation of Multiple Equipment Pieces with Varying Dimensions**
7. Future Work

The obvious possible extension to KineMach is the design and implementation of animation methods to instantiate and manipulate other types of generic equipment pieces in a high level, task-based language. KineMach and the underlying VITASCOPE animation language are both designed to be seamlessly extended by other researchers without having to modify or fully understand the existing implementations. Animation methods to manipulate other virtual pieces of commonly used construction equipment (e.g. dozers, scrapers, front-end loaders, articulated trucks, draglines etc.) can thus be designed and seamlessly integrated into KineMach’s existing vocabulary.

Another interesting area of future research is incorporating dynamics into KineMach’s computations. The current implementation only analyzes the motions of the equipment pieces kinematically without regard to the forces that cause those motions. A KineMach-instantiated crane, for instance, would never tip over if the crane’s hook is overloaded. In fact, such computation is not considered or performed at all. Introducing the notion of forces and loads into the computation process would cause KineMach’s equipment pieces to not only be kinematically intelligent, but dynamically “smart” as well.

KineMach significantly reduces the amount of information a simulation model must communicate in order to animate the processes it models. Notwithstanding, several pieces of otherwise ignorable spatial information must still be encapsulated in simulation models to allow them to communicate construction tasks to KineMach. For instance, in the steel erection operation’s case, the simulation model must still instruct KineMach’s crane on what (i.e. which shape) to put where (i.e. the target installation location). Future research can concentrate on alleviating this information load from simulation models altogether by extracting such necessary spatial information (e.g. each steel shape’s final configuration) from product models based on standard protocols such as the Industry Foundation Classes (IFCs).
8. Summary and Conclusions

Existing methods of animating modeled construction processes in 3D require that simulation models be compulsorily created at elemental motion-level sub-task detail regardless of whether those details are essential to the analyses from the decision-making point of view. In addition, discrete event simulation models’ natural ignorance about the corresponding virtual construction environments that can animate modeled processes requires engineers to manually input spatial, geometric operational details into models so that 3D animations can be logically communicated during simulation runs.

The presented research addresses both these critical issues and extends the state-of-the-art of scientific, simulation-driven 3D construction process visualization. The work puts in place the technology that engineers can use to animate discrete event simulation models created at typical higher, task level abstractions (i.e. with low modeled details), without having to explicitly populate models with otherwise ignorable spatial, geometric, virtual world information.

The design of 3D animation methods to visualize simulated construction processes modeled and communicated at high, task level detail requires three key technologies:

1. Simple, parametric text methods (i.e. statements) with semantics rich enough to succinctly represent the description of common construction tasks.
2. Automated dissection of the communicated construction tasks into constituent elemental motions of relevant virtual equipment pieces.
3. Generation of spatial, geometric information to compute the amplitudes and directions of the deciphered elemental motions to visually depict the performance of construction work.

Designing statements to instantiate generic pieces of construction equipment that accept pertinent task-level instructions (also in parametric text statements) provides a logical basis of generally organizing 3D animation methods intended to communicate construction in a higher level, domain specific terminology. In addition, the application of IK techniques provides an ingenious framework to automatically compute the
elemental motions articulated machines undergo as they perform specific construction tasks. Furthermore, IK techniques (by their very nature) automatically generate the geometric transformations necessary to describe the computed elemental motions thus making the explicit communication of detailed, spatial, geometric information unnecessary.

The designed 3D animation methods allow engineers to visualize discrete event simulation models created at typical, higher, task-level detail. In addition, they relieve engineers from manually encapsulating unnecessary (from simulation point of view) detailed geometric specifics into simulation models. By greatly simplifying the process of animating modeled operations, the work puts in place the computing infrastructure to make simulation-driven construction operations visualization more straightforward, accessible, and practical.

9. References


1. Introduction

The ability to see a 3D animation of an operation that has been simulated as a discrete-event process model allows for three very important things: 1) The developer of the simulation model can confirm that there are no errors in the coding (verification); 2) The domain experts, field personnel, and decision makers can discover differences between the way they understand the operation and the way the model developer understands it (validation); and 3) The model can be communicated effectively offering decision makers valuable insight on subtle operational details that would otherwise be non-quantifiable and non-presentable. This coupled with verification and validation, makes simulation models credible and encourages their use in making decisions (Law and Kelton 2000).

The necessity to effectively communicate modeled construction processes and the resulting evolving products (i.e. constructed facilities) is the motivation behind ongoing visualization research efforts at Virginia Tech. These efforts focus on designing automated, process simulation-driven methods to visualize construction processes and evolving products in dynamic 3D virtual worlds. The tangible outcome of this ongoing research is the VITASCOPE visualization system. VITASCOPE is an acronym for VIvisualizaTion of Simulated Construction OPErations. VITASCOPE is a user-extensible 3D animation language designed specifically for visualizing simulated processes (construction operations in particular) in smooth, continuous, dynamic 3D virtual worlds (see Chapters 2 and 3 for details). A limited subset of the VITASCOPE language and the corresponding prototype implementation were referred to as the Dynamic Construction Visualizer (DCV) in some prior publications (Kamat and Martinez 2002, 2001).
1.1 Research Motivation

All 3D visual simulations (i.e. visualizations) require 3D CAD models of the involved simulation entities. These models are instantiated and manipulated in virtual environments to represent dynamic, 3D synthetic worlds. In the case of 3D construction process visualizations, the required CAD models include the terrain (i.e. the jobsite), various pieces of construction equipment (e.g. trucks, loaders, and cranes), construction materials (e.g. steel beams, concrete blocks, and rebar), and humanoids to represent construction workers. 3D CAD modeling is, by definition, a time-consuming activity and has been cited in the literature as a deterrent to 3D visualization of simulated construction processes (Zhang et al. 2002).

Acquiring 3D models of equipment, materials, and humanoids does not, however, pose a significant problem today as several CAD model vendors market libraries of pre-created models at reasonable prices (e.g. www.3dcadbrowser.com, www.viewpoint.com). These 3D models can be instantiated at varying levels of detail in virtual worlds and can be manipulated in a straightforward manner by monitoring and modifying their geometric transformations (i.e. translation and three-dimensional orientation) (Kamat and Martinez 2002). In addition, because of their generic nature, acquired and/or created CAD models of equipment, materials, and workers can be easily compiled into a library and directly reused in other, future visualizations.

Depicting the jobsite terrain in 3D is, however, a challenging proposition in the scientific 3D visualization of simulated construction operations. This can be attributed to the following factors:

1. Most construction work is carried out outdoors and every jobsite is different (i.e. each jobsite has a different layout, topography, and scenery). Thus, a separate, detailed, 3D CAD model of the terrain must be available for every jobsite on which construction processes are visualized.

2. The layout and topography of the jobsite terrain is dynamic and can change shape dramatically over the course of construction (Lipman and Reed 2000). Many
construction processes (e.g. digging and dumping dirt, trenching, tunneling, blasting etc.) directly alter the topography of the jobsite terrain.

The current state-of-the-art in scientific, simulation-driven, 3D construction process visualization has the following limitations related to the jobsite terrain:

1. 3D CAD models that represent jobsite terrains must be manually created (i.e. hand-modeled) using 3D modeling programs (e.g. AutoCAD, 3D Studio) and then exported in platform-independent formats (e.g. VRML) that can be later loaded at visualization runtime. This activity typically requires a significant amount of effort, consumes a correspondingly large amount of time, and is often impossible to execute due to the unavailability of data (e.g. terrain elevations) in the appropriate form or the inability to process it manually.

2. There are no mechanisms to visually depict the deformation and evolution of the jobsite terrain in response to the virtual construction operations being visualized. For example, in visualizations of modeled earthmoving operations, virtual loaders appear to dig at loading sites. However, the terrain itself does not deform, change shape, or evolve over time. This compromises realism in animating operations such as earthmoving, wherein the changing topography is central to the involved processes. In addition, this limitation of being unable to modify the terrain CAD database and visually depict a deforming jobsite terrain precludes existing animation methods from describing operations such as tunneling and blasting that dramatically alter the jobsite topography.

1.2 Main Contribution

The research this paper presents directly addresses the abovementioned issues. First, we describe mechanisms that were designed for automatically generating photorealistic, digital, 3D terrain CAD databases to represent construction jobsite terrains in visualizations. The work capitalizes on the availability of detailed topographical (e.g. Digital Elevation Map – DEM) and aerial imagery (e.g. National Aerial Photography Program - NAPP) data that is readily available in digital format from several government (e.g. United States Geological Survey - USGS) and private organizations.
Second, we present designed animation methods that allow engineers to describe the evolution of virtual jobsites by depicting deformations to the loaded terrains in response to common construction operations such as earthmoving and trenching. By implementing techniques to integrate many sources of information and automatically generate photorealistic virtual jobsite terrains, our first contribution alleviates the need to manually create 3D terrain CAD models for use in visualizations as has been the case hitherto. By designing methods to describe the deformation and evolution of instantiated terrain CAD databases, our second contribution 1) enhances realism in the visualization of several common construction operations, and 2) allows the visual depiction of several other simulated construction processes that were hitherto impossible to describe in 3D virtual worlds.

2. Challenges

The design of simulation-driven methods to automatically generate and then visualize the deformation and evolution of jobsite terrains in response to common construction processes in smooth, continuous, dynamic, 3D virtual worlds presents numerous interesting challenges. Digital elevation and aerial imagery data is readily available for the entire United States and several other parts of the world. However, this data is archived in several different digital formats at varying levels of detail (Burrough and McDonnell 1998). In order to automatically generate virtual jobsite terrains from disparate sources of archived digital data, we must implement techniques to 1) parse (i.e. read) and interpret streams of elevation and imagery data in all commonly available formats, and 2) convert interpreted elevation and imagery data into a standard internal representation that can be visually depicted in 3D virtual worlds and can also be locally modified as needed to describe actions such as terrain deformation.

The resultant locally deformed shape and appearance of a virtual terrain in response to an animated elemental construction task (e.g. digging) cannot be determined a-priori by animation-authoring processes (e.g. discrete-event simulation models) or by the visualization engine itself. Each deformation a virtual terrain must undergo in response to an animated construction task (e.g. digging) depends on 1) the type, size, and
configuration of the involved piece of virtual equipment (e.g. backhoe), and 2) the amplitude of the motion of its components (e.g. boom, stick, bucket) in the particular animated instance of that task.

For instance, simulation models that describe earthmoving operations in VITASCOPE’s 3D animation language indicate the location (as a 3D coordinate) where a virtual piece of equipment (e.g. backhoe) must “dig” in each loading pass (see Chapter 5 for details). However, the exact trajectory that a piece of equipment’s digging implement (e.g. bucket edge) will follow in each virtual digging stroke cannot be determined beforehand by either the authoring process (i.e. simulation model) or the visualization engine. The computation that determines the shape of an evolving terrain in response to animated construction processes must thus be performed during animation in real-time and is solely the responsibility of the visualization engine.

Finally, from the implementation point, virtual terrains are computationally intensive to maintain and render. Terrains are typically composed of several thousand textured polygons that in the absence of any optimizations would create a graphics pipeline bottleneck (Garland and Heckbert 1995). Rendering terrains in real time at interactive frame rates has thus been an important and long-studied problem in the field of computer graphics (Turner 2000). In order to depict large pieces of dynamic terrain in 3D animations, the number of polygons that the graphics pipeline needs to display must be limited without compromising the detail of the terrain. This is particularly important in the visualization of construction operations because unlike many other visual simulations (e.g. flight simulators), the terrain on a virtual construction jobsite is seen and manipulated at close range. In construction, since most of the action takes place on the ground, the simplification of the terrain database to improve performance cannot compromise the fidelity of the terrain’s appearance.

The issues we address in this work are thus four-fold. First, we implement techniques to automatically generate terrain CAD databases by combining data from disparate elevation and imagery data sources into a unified, consistent data set. Second, we
investigate feasible data structures to internally maintain and locally manipulate (on demand) the generated terrain databases. Third, we design methods to compute the amount, location, and the resultant shape of deformations that virtual terrains undergo in response to animated construction processes and apply those deformations to the terrain databases in real-time. Finally, we identify and adopt computationally efficient techniques of maintaining and rendering the dynamic 3D terrain CAD databases in a way that precludes performance bottlenecks in the graphics pipeline and helps maintain interactive animation frame rates. It is obvious that these issues are tightly intertwined and related.

3. Digital Representation of Terrains

A terrain can be mathematically conceived as the graph of a continuous function \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \) (De Berg and Dobrindt 1995). The surface of the earth is a continuous phenomenon (field) rather than a discrete object. However, to fully model even a small portion of the earth’s surface continuously on the computer, we would need an infinite number of points. In order to describe continuous surfaces digitally on the computer using a finite amount of storage, terrains are represented in discrete formats that are generically called Digital Terrain Models (DTMs) (Giger 2002).

A commonly used format of digitally representing a terrain surface (i.e. DTM) is the Digital Elevation Model (DEM). The term DEM has often been used interchangeably with DTM to describe “any digital representation of the continuous variation of relief over space” (Burrough and McDonnell 1998). Strictly speaking, however, a DEM specifically refers to a raster or regular grid of spot heights (USGS 2001). DEMs consist of a sampled array of elevations for a number of ground positions at regularly spaced intervals to describe an axis-aligned grid of terrain. The distances between the sampled positions (i.e. the grid spacing) defines the resolution of a DEM and is the major determinant of its accuracy (USGS 2001).

The DEM format is a compact and efficient way of representing terrain surfaces and lends itself well to computer computations. A variety of DEMs are readily available from
several government and private organizations. For instance, the United States Geological Survey (USGS) freely provides DEM data for the entire United States (USGS 2002a). The best resolution commonly available in the USGS DEMs is 10m, with a vertical resolution of 1m. In addition, several private organizations (e.g. quarry owners) regularly update and maintain detailed (i.e. with fine resolution) DEMs of their property for use in planning, quantification, and record-keeping (Zalubowski 2002).

A DEM is generally archived in one of two common digital formats. The standard method of archiving a DEM is through the use of numerical height maps. A numerical height map is simply a two dimensional array of numerical values. Each numerical value in the array indicates the elevation of the terrain’s surface at that array position. This is graphically presented in figure 1. The table on the left presents an example of a numerical height map. The image on the right presents the interpretation of the data in that height map. A digital terrain is interpreted as a mesh generated by iterating through the two planar indices of the height map array and setting the height of each vertex to the numerical value of the height map (i.e. elevation) at that point.

Another common method of archiving DEMs is through the use of digital, grayscale image height maps. In grayscale images that represent DEMs, the brightness of each image pixel corresponds to the height of the terrain at that point. Generally, dark regions on the image represent lower elevation values and lighter regions indicate high elevation.
values. The calibration of the grayscale (i.e. the numerical elevation values of pure black and pure white regions) is often implementation dependent. Figure 2 graphically presents a grayscale image height map. The image on the left is comprised of various shades of gray. The mesh on the right depicts a possible interpretation of the image height map. The reader will observe that the black areas on the image are interpreted as low-lying areas whereas the brighter alphabets are interpreted as peaks.

Figure 2: Interpretation of Grayscale Image Height Maps

4. Visualizing Dynamic Terrains

In order to capitalize on the availability of archived digital terrain data in visualizing simulated construction operations, the following key technologies are necessary:

1) Generation of 3D CAD terrain databases (i.e. CAD models) from DEM data
2) Maintenance of the generated 3D terrain models in efficient, modifiable format
3) Computationally efficient rendering of the current 3D CAD terrain databases

4.1 Generating 3D Terrains from DEMs

The geometry of uneven, three dimensional surfaces is generally represented on the computer as a continuous mesh of polygons (usually triangles) (Woo et al. 1997). The process of converting DEM data into a 3D terrain CAD database thus requires interpretation of the input data and construction of a continuous, 3D polygon mesh corresponding to that data. The coordinates of each vertex in the constructed mesh can be directly obtained from the DEM if it is archived as a numerical height map (e.g. figure 1).
In cases where the DEM data is archived as a grayscale image (e.g. figure 2), the extraction of coordinates from the height map requires two, implementation-dependent calibration factors.

The first is the numerical value of the horizontal spacing (i.e. distance) between sampled positions. This value defines the horizontal resolution of the grid by indicating how the distance between adjacent pixels in the input image must be interpreted. The second calibration factor required to extract 3D coordinates from an image height map is the vertical scaling factor. This factor defines the vertical resolution of the image height map. For instance, if the color scale of a particular implementation assigns 0.0 to pure black and 1.0 to pure white and the vertical scaling factor is 1000m, then a pure white pixel on the grayscale image height map is interpreted to have an elevation of 1000m. A pure black pixel is similarly interpreted as a 0m elevation value. Shades of gray (color values ranging between 0.0 and 1.0) are linearly interpolated to decipher the elevations indicated by each pixel in the image height map.

Once the coordinates of the vertices are extracted from the image height map, a 3D polygon mesh can be constructed in a similar manner to that of numerical height maps. A grayscale image height map must thus be converted to a numerical height map before 3D mesh vertex coordinates can be extracted from the data. In both cases, the extraction of vertex coordinates from the height maps and construction of a corresponding 3D mesh generate a computer-interpretable geometrical shape of the terrain segment that is archived in the input DEM.

4.1.1 Specifying the Appearance of Generated Terrains

The 3D polygon mesh constructed from DEM elevation data merely specifies the shape of the geometrical object that represents a patch of terrain. In order for this mesh to “look” like a real terrain when instantiated in virtual worlds (i.e. visualizations), we need to specify information that indicates its appearance. A common computer graphics technique for determining the appearance of 3D polygonal objects is to use texture mapping. Texture mapping is a classical technique for image synthesis wherein a two-
A dimensional (2D) image (called a texture map) is superimposed on a three-dimensional (3D) geometric object (Heckbert 1989, Haeberli and Segal 1993). The visual outcome of this process is that the 3D geometrical object acquires a surface appearance similar to that of the superimposed 2D image. Conceptually, texture mapping is thus the electronic equivalent of applying wallpaper, paint, or veneer to real world objects.

Texture mapping is almost exclusively used to specify the appearance of 3D terrain CAD objects in virtual environments. This is accomplished by superimposing a 2D image that represents an aerial view over the corresponding 3D geometrical terrain mesh. This process is graphically presented in figure 3. The geometry (i.e. the 3D mesh) and the appearance (i.e. the texture map) together define the visual representation of terrain objects in visualizations. In the presented work, where the terrain’s geometrical representation can be constructed from actual DEM data, texture mapping presents interesting possibilities of specifying the appearance of instantiated 3D terrain objects.

![Figure 3: Texture Mapping 3D Terrain Meshes](image)

For instance, constructed terrain geometry can be overlaid by satellite or aerial imagery corresponding to the same geographic area represented in the input DEM. This powerful technique can yield impressive images and useful results. Similar to elevation data, several government (e.g. USGS, National Aeronautics and Space Administration - NASA) and private agencies have archived digital, high-resolution aerial imagery for the
entire United States. In addition, since the applied texture map is simply a 2D image representing the overhead view of the modeled area, custom 2D images may readily be used to drape constructed 3D terrain meshes. This is useful in cases where actual aerial color imagery at the desired resolution is inaccessible or when specific, synthetic appearances must be applied to instantiated 3D terrains during visualization.

4.2 Maintaining 3D Terrain Databases

In order to display virtual terrains on the computer at each frame in visualizations, constructed 3D terrain databases (i.e. models) must be stored in efficient, accessible formats. At each frame, rendering (i.e. drawing) algorithms access the databases and render an image of their current state. In general, 3D terrain models are unique compared to other 3D geometrical objects because of their typical large size and extent (compared to other virtual objects). For this reason, efficient data structures for storing and maintaining 3D terrain databases have been widely studied in computer graphics (Kofler 1998).

In general, there are two commonly used formats for storing and maintaining 3D terrain models in computer memory. The first format is a regular grid height field that is very similar in structure to regular grid DEMs. The height field storage model stores a regular grid of 3D vertex coordinates to encapsulate the geometry of terrain surfaces (Guedes et al. 1997). Figure 4 (a) graphically presents an example of this storage model. The vertices are connected by lines to form connecting triangles (i.e. polygons) that together describe the surface geometry. In the case of terrains generated from regular grid DEMs, the height field storage model simply stores each 3D vertex coordinate extracted from the input data. The constructed (and rendered) 3D polygon mesh retains the original vertices extracted from the base DEM data.

The second common format of maintaining 3D terrains on the computer is the Triangulated Irregular Network (TIN) model. A TIN model represents a terrain surface as a set of non-overlapping, contiguous triangular faces that are of irregular size and shape (Peucker et al. 1978). Figure 4 (b) presents an example of such a representation. In
terrains stored as TINs, irregularly spaced vertices are adapted to the terrain surface, with more points in areas of rough terrain and fewer in smooth, even terrain areas. Thus, in terms of vertex count, TINs are often more efficient at maintaining terrain surfaces.

![Figure 4: Terrain Database Storage Models](image)

The process of storing a terrain surface constructed from a regular grid DEM as a TIN model consists of heuristically choosing “important” points on the surface and discarding other extracted vertices (Chen and Guevara 1987). Few points are chosen in areas of smooth terrain, and large numbers of points in areas of rough terrain. The points that remain are assumed to be connected together to form triangles. Thus, in the case of TINs, the constructed (and rendered) 3D polygon mesh may not retain all the original vertices extracted from the base DEM data. Storing 3D terrains as TINs is essentially optimizing the generated terrain databases so that minimum triangles are retained for rendering.

4.2.1 Choosing the Appropriate Storage Model

The choice of a particular terrain database storage model for a visualization application depends on the type of terrains that are instantiated in virtual worlds and on the expected interaction of other 3D scene objects with the terrain databases. For instance, in several common visual simulations (e.g. flight simulators), the instantiated terrain databases only serve as static 3D models that are viewed in the background. In such visualizations, there is typically no interaction between other scene entities (e.g. airplanes) and the terrain
databases. In addition, the 3D terrain models themselves remain unmodified over the course of visualization. Such 3D terrain databases whose physical properties (geometry and appearance) remain unchanged over the course of visualization are termed Static terrains (He 2000).

In the visualization of simulated construction processes, the 3D terrain models are typically expected to be modified as a result of their interaction with different pieces of construction equipment (e.g. excavators, dozers) or due to the performance of certain virtual construction processes (e.g. blasting) that in the real world drastically change the surface topography. Such 3D terrain databases whose geometric and other properties may change during the process of visualization are termed Dynamic terrain surfaces (He 2000).

In terms of computer graphics, dynamic terrains involve possible modifications to the geometry, color, and texture of the 3D terrain databases during visualization. The ability to modify (i.e. deform) terrain databases in response to virtual construction processes requires unobtrusive and efficient interfaces to the stored 3D terrain models since such manipulation of the terrains must be accomplished in real-time without an adverse effect on the animation frame rate.

In addition, in the case of simulated construction process visualization, the modification interfaces to the stored terrain databases must be generic in nature. This is necessary because unlike other terrain-modifying visual simulations (e.g. equipment training simulators), the possible range and types of interactions between virtual equipment pieces and the terrain databases cannot be determined a-priori. In simulated construction operations visualization, the type and range of interaction between other scene entities and the terrain databases depends on 1) the particular pieces of virtual equipment active in the visualization, and 2) the types of processes modeled in the underlying simulations that drive the visualization.
The ability to efficiently modify (i.e. deform) and render instantiated 3D terrain databases in visualizations largely depends on the storage model that is adopted to maintain the terrain models on the computer (Lipman and Reed 2000). We adopted the regular grid height field terrain database storage model in the presented work. This design decision is attributed to the following factors:

- Regular grid terrain models are much easier to track and locally manipulate (i.e. deform) than TIN models (He 2000)
- For a given amount of computing power, regular grid terrain triangulation models perform better than TIN models (Kumler 1994)
- Terrain rendering algorithms for regular grid terrain models are superior to the algorithms available for rendering TIN models (Lindstrom and Pascucci 2001)
- Regular grid terrain models are the foundation of hierarchical data structures that are required to be built by rendering algorithms for efficiently drawing 3D terrains at interactive frame rates (He 2000)
- Regular grid models are easier to create than TIN models as most accessible DEM elevation data is available as regular grid height maps making extraction of regular grid vertices straightforward (USGS 2001)
- Literature suggests that regular grid terrain models are the de-facto standard for storing 3D terrains in dynamic visual simulations (Rognant et al. 1998)

4.3 Rendering 3D Terrain Models

The major challenge with any 3D terrain model rendering approach is to depict a visually accurate and detailed representation of the current terrain database while maintaining interactive frame rates (ideally > 24 frames per second). The simplest way to render a regular grid height field terrain model is to generate strips of triangles to cover the entire set of model vertices (e.g. figure 4 (a)). However, this seemingly straightforward approach is not practical when used with typically large regular grid terrain models. For large terrains, the amount of geometry that must be rendered at each frame if such an approach is used will cause severe performance bottlenecks even on the comparatively powerful computing platforms available today (Lindstrom and Pascucci 2001).
Several optimization methods, called level-of-detail (LOD) techniques, have been designed over the years to reduce the amount of terrain geometry that must be rendered at each frame in visualizations (e.g. Lindstrom et. al 1996, Duchaineau et al 1997, Hoppe 1998). The general strategy used in all LOD terrain rendering techniques is that portions of a terrain model that are farther away from the visualization viewer require less geometry to be accurately represented than do terrain portions that are comparatively closer to the viewer. In addition, terrain portions that are very rough and uneven need more detailed geometry (i.e. triangles) to be accurately represented than do terrain portions that are smooth, flat, and/or even.

4.3.1 View-Dependent Terrain Model Triangulation

In interactive animations such as the visualization of simulated construction processes, the viewpoint of the viewer(s) can change over time as the viewer(s) navigate around the virtual construction sites. Thus, the amount of geometric detail (i.e. number of triangles) necessary to accurately represent the 3D terrain models can change over time. Therefore, terrain rendering LOD techniques are often dynamic in nature. Such techniques allow portions of the terrain that are currently far away to be rendered with few triangles. However, the same portions of terrain are dynamically rendered in more detail (i.e. with more triangles) as the viewer(s) navigate closer to those portions. Such dynamic rendering methods that gradually change a terrain’s LOD are often called continuous level-of-detail (CLOD) techniques (Ogren 2000, Roettger et al. 1998).

In general, the aim of view-dependent terrain model rendering is to create a simplified version of the 3D terrain model using a small subset of the original mesh triangles such that for a given view, the simplified triangulation is a good approximation of the original, dense mesh. This computation is performed continuously during visualization each time the viewpoint changes. At each frame, then, the simplified version of the terrain model for the current viewpoint is rendered (i.e. drawn) on the screen. For any viewpoint, the number of original mesh triangles retained in the simplified terrain triangulation increase as the distance between the viewer and the triangles decrease. In addition, the triangulation also increases in areas where the terrain is comparatively irregular.
View-dependent CLOD techniques thus present efficient methods of rendering large 3D terrain models at interactive animation frame rates in dynamic virtual environments.

In the presented work, we have adopted a method of real-time, view-dependent, adaptive tessellation (i.e. triangulation) to render dynamic 3D terrain models that represent virtual construction jobsites. The adopted technique helps maintain interactive frame rates during visualization by dynamically reducing visually insignificant regions of instantiated 3D terrain models to fewer polygons (i.e. triangles). This computation is performed in real-time as viewer(s) navigate around virtual construction jobsites. Thus, at each drawn frame, a high-fidelity image of the current 3D terrain model is rendered with the fewest possible polygons for that particular view.

5. Technical Approach for Computing Terrain Deformations

Several common construction processes such as excavation, filling, blasting, and tunneling are performed with the deliberate intention of altering the jobsite topography. In such operations, the evolving terrain topography is central to the continuous performance of the involved processes. For instance, excavators change their position frequently depending on how digging progresses. Similarly, haul distances and routes dynamically evolve on earthmoving jobsites as cut and fill operations progress. Accurate visualization of the evolving construction jobsite terrain is thus of significant importance to the credibility of several animated construction processes.

The need to dynamically modify instantiated 3D terrain models introduces two challenges to the visualization of simulated construction processes. The first is the computation of the deformations that 3D terrain models must undergo in response to the performance of virtual topography-modifying construction processes. The second is the application of the computed deformations to the current 3D terrain models in real-time.
We considered several strategies in designing methods to visualize dynamic jobsite terrains in simulated construction process visualizations. The initially contemplated technique was the design of animation statements to allow running simulation models to explicitly communicate the simulation time, location, and extent of the deformations to be applied to the terrain models. For instance, each time an excavator’s digging stroke is communicated (using existing animation methods), the location and amount of deformation to be applied to the terrain could also be explicitly communicated by the running simulation models.

This seemingly straightforward approach is however impractical to implement. A discrete-event simulation model that communicates a piece of equipment’s digging stroke cannot determine beforehand, the resultant locally deformed shape and appearance of the terrain model in response to the communicated construction task (i.e. digging stroke). The deformation that the virtual terrain model must undergo to accurately depict the outcome of a particular digging stroke depends on the 1) the type, size, and configuration of the digging implement of the pertinent virtual piece of equipment (e.g. backhoe), and 2) the amplitude of the motion of its components (e.g. boom, stick, bucket) in the particular animated instance of digging.

In addition, the deformation of the terrain in response to tasks such as digging is a continuous process. For instance, the deformed shape of the terrain in response to an excavator’s digging pass evolves continuously during the performance of the digging stroke and is not an instantaneous discrete event. Discrete-event simulation models can only communicate when a particular instance of digging began and how long it lasted. A simulation model is unaware (i.e. encapsulates no knowledge) of the exact trajectory that a virtual equipment’s digging implement (e.g. bucket edge) will follow in any particular instance of digging. We therefore concluded that the computation to determine the shape of the evolving terrain in response to communicated construction processes must be performed during animation in real-time and must be relegated to the visualization engine.
5.1 Detecting and Displaying Surface Penetration

Figure 5 outlines the strategy we adopted to compute deformations to virtual jobsite terrains in construction process visualizations. This scheme relies on the authoring discrete-event simulation models to communicate construction tasks (e.g. digging) to virtual pieces of instantiated construction equipment (e.g. backhoes). Then, during visualization of those communicated construction tasks, we continuously monitor the position of the involved virtual equipment’s digging implement (e.g. bucket edge) to detect whether or not the virtual terrain has been penetrated.

![Figure 5: Terrain Deformation Computation Scheme](image)
To dynamically update deformation changes to the current 3D terrain database when surface penetration is detected, we adopted a generalized form of the simple implement-terrain interaction model outlined in Lipman and Reed (2000). At each instant during the visualization of a communicated digging stroke, we compute the global position of the involved digging implement (e.g. bucket edge) inside the 3D virtual world. Once an implement’s position is determined, we compare it’s height to the elevation of the virtual terrain below. If the elevation of the terrain at that point is greater than the implement’s current height, we conclude that the implement has penetrated the terrain’s surface.

5.1.1 Computing Digging Implement Positions

Typical pieces of construction equipment can be thought of as articulated hierarchies. For instance, as figure 6 presents, a backhoe can be kinematically described on the computer as consisting of a bucket that is connected to the stick, which in turn is connected to the boom. The boom is further attached to the slewing cabin that in turn rests on the crawler tracks.

Figure 6: Description of Kinematically Articulated Hierarchies
This is a hierarchical arrangement where each component can move independently affecting components lower in the hierarchy. For instance, the position of a virtual backhoe’s bucket inside a 3D world depends on the rotations of all the backhoe components (i.e. stick, boom, and cabin) higher in the hierarchy, and on the position and orientation of the backhoe’s crawlers (Lipman and Reed 2000). The computation to determine the global 3D position of a virtual digging implement (e.g. backhoe bucket’s edge) must then be performed using these inputs.

In order to determine each component’s position and orientation relative to its parent component in a kinematic hierarchy, we adopt a mathematical notation and result from elementary computer graphics literature. In particular, we associate four 4x4 matrices with each component in a virtual equipment’s kinematic hierarchy. Figure 7 presents such a matrix set for the bucket on a virtual backhoe. These matrices encapsulate a component’s position, horizontal rotation (yaw), vertical rotation (pitch), and side rotation (roll) with respect to its parent component’s local space (i.e. coordinate system). In case a component (e.g. backhoe crawlers) is at the root of a hierarchy (i.e. has no parent component), then the associated matrices represent its position and orientation in global 3D space (i.e. the global coordinate system).

In the bucket’s position matrix P, the 3D coordinate defined as \((T_x, T_y, T_z)\) represents the bucket’s position (i.e. translation) relative to the backhoe’s stick (i.e. the bucket’s parent component). The bucket’s rotation matrices (H, V, and S) similarly encapsulate the bucket’s axial rotations relative to the virtual stick. In particular, the angles \(\alpha\), \(\beta\), and \(\gamma\) represent the bucket’s relative rotations about the stick’s local Y, Z, and X axes respectively. In a virtual bucket’s case, \(\alpha\) and \(\gamma\) will have a value of zero since a backhoe’s bucket only has one degree of freedom (in the vertical plane) relative to a backhoe’s stick.

The mathematical result we exploit in our computation is that the product of a component’s matrices, \(L = S \times V \times H \times P\) completely describes the configuration of that component relative to its parent. For instance, if \((x, y, z)\) is any 3D point on the virtual
bucket, its position relative to the parent stick can be mathematically computed as 
\((x_r, y_r, z_r) = L \times (x, y, z)\) (Woo et al. 1997).

![Figure 7: Backhoe Bucket’s Matrix Set](image)

\[
P = \begin{bmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
P \times H \times V \times S
\]

In addition, the product of the stack of matrices of all components higher in the hierarchy completely describes the configuration of a component in global 3D space. In case of the same point \((x, y, z)\) on the bucket, its position relative to the global coordinate system (i.e. in global 3D space) can be mathematically computed as
\[(x_g, y_g, z_g) = L_{Bucket} \times L_{Stick} \times L_{Boom} \times L_{Cabin} \times L_{Crawlers} \times (x, y, z) .\]

The process by which we compute the global 3D position of a piece of equipment’s digging implement (e.g. bucket edge) will now be obvious to the reader. We simply choose critical points (i.e. coordinates) on the virtual digging implement and multiply them by the stack of matrices of all components higher up in the equipment’s hierarchy. This computation provides us with the global 3D position of those points which we can then compare with the elevation of the terrain model at that projected location.
5.1.2 Updating Terrain Models

To reflect the outcome of any detected surface penetrations in the 3D terrain database, we then set the elevation value of the terrain model at that point to be equal to the digging implement’s current global height. The adopted regular grid terrain database storage model makes such local elevation modification straightforward. We simply update the numerical elevation value of the pertinent point(s) in the regular grid terrain model being maintained. The visual outcome of continuously performing this computation and updating the terrain database during visualization is a terrain deformation whose shape conforms to the digging implement’s trajectory during the particular digging stroke.

Deposition (i.e. dumping) of dirt is similarly computed and depicted during the visualization of dumping strokes. At each instant, we monitor the position of a piece of equipment’s bucket during dumping. We then increase the elevations of projected terrain points along the trajectory followed by the involved bucket as it dumps dirt. The visual outcome in this case is the depiction of a heap whose shape and size is proportional to the virtual bucket dumping dirt.

5.2 Influence of Grid Resolution on Deformation Computations

The results obtained by employing the presented computation scheme are significantly influenced by the grid resolution of the 3D terrain model being manipulated. In particular, the technique only produces accurate results if the horizontal resolution of the maintained terrain grid is significantly dense compared to the width of virtual digging implements that deform the terrain models. In general, finer the terrain grid model, better are the produced visual results (Lipman and Reed 2000). This is obvious because as grid resolution increases, the computation scheme can manipulate more, closely-spaced points on the terrain models in depicting the shape of deformations. On the other extreme, the algorithm may have no terrain points to manipulate if the resolution of the grid is larger than the width of a virtual digging implement.

The elevation data from which 3D terrain models are intended to be automatically constructed is, however, rarely available in the order of such high grid resolutions. For
instance, the reader will recall that the highest horizontal resolution available in most USGS DEM data is 10m, which is significantly large compared to the width of digging implements (e.g. bucket edges) on common construction equipment pieces. We address this issue by constructing high resolution meshes while building 3D terrain models from input elevation data.

In particular, we construct a mesh of high, user-defined resolution from the comparatively lower resolution input elevation data. This is accomplished by simply interpolating data points along the two horizontal axes of the input data. This operation generates additional, intermediate elevation data points so that the resolution of the resulting grid is sufficiently finer than the width of virtual digging implements expected to be utilized in particular visualizations.

6. ViTerra

Our implementation of techniques that automatically generate and then manipulate 3D terrain models in visualizations is a powerful tool that greatly increases the credibility of several visualized simulated construction operations. This tool, called ViTerra, is implemented as an extension (add-on) to the VITASCOPE visualization system.

The ViTerra add-on extends the VITASCOPE animation language by implementing the animation statements presented in table 1. In particular, the add-on defines animation language statements that automatically construct 3D terrain models by combining disparate elevation and imagery data sources. In addition, the add-on presents animation statements to manipulate the created terrain databases by explicitly changing the elevation at any point on the instantiated terrains. For reasons described in the previous section, these latter methods are, however, not intended to be directly used in animation trace files by visualization-authoring simulation models. Instead, they are implemented so that other VITASCOPE extensions (add-ons) can manipulate instantiated 3D terrains by executing these statements from within their add-on modules. The detailed usage of all statements is presented in appendix J.
Table 1: Usage of ViTerra Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERRAIN [ElevationDataSource] [TextureDataSource];</td>
<td>Construct a 3D terrain model from the indicated elevation (geometry) and texture (appearance) data sources</td>
</tr>
<tr>
<td>TERRAIN.RaiseElevation [X] [Z] [DeltaY];</td>
<td>Raise the elevation of the specified point on the terrain by the indicated amount</td>
</tr>
<tr>
<td>TERRAIN.LowerElevation [X] [Z] [DeltaY];</td>
<td>Lower the elevation of the specified point on the terrain by the indicated amount</td>
</tr>
<tr>
<td>TERRAIN.SetElevation [X] [Z] [TargetY];</td>
<td>Set the elevation of the specified point on the terrain to the indicated value</td>
</tr>
</tbody>
</table>

6.1 Terrain Model Construction

ViTerra automatically constructs a 3D terrain model by combining the data sources specified as arguments to the TERRAIN statement. In particular, the TERRAIN statement takes two arguments as input. The first argument is a string that specifies the source of the digital elevation data (i.e. DEM) to be used in constructing the terrain model geometry. The second argument is a reference to the aerial image that is to be draped on the constructed geometry to describe its appearance. An example of a TERRAIN statement is presented below:

    TERRAIN '4349CATD.DDF,10,1' '4349Tex.JPG';

In the above statement, the terrain geometry is to be constructed from a DEM that is archived in the popular USGS Spatial Data Transfer Standard (SDTS) format (USGS 2002b). In this case, the elevation data argument specifies the root SDTS filename (*CATD.DDF), the horizontal resolution of the input DEM (10m), and its vertical resolution (1m). The second argument to the statement specifies the image (in standard JPEG format) that is to be draped over the constructed terrain model to describe its appearance.
ViTerra can construct the geometry of 3D terrain models from several different DEM sources (numerical and grayscale image height maps). Appendix I provides a listing of all DEM input formats ViTerra supports. ViTerra exploits the services of the Geospatial Data Abstraction Library (GDAL) for parsing and interpreting input DEM height fields. GDAL (Warmerdam 2003) implements algorithms to translate raster DEMs in several common geospatial data formats presented in appendix I.

ViTerra can similarly set the appearance of constructed terrain geometries from image files archived in several common formats. ViTerra relies on the Simple DirectMedia Layer (SDL) library (Lantinga 2003) for reading the texture image files specified as arguments to the TERRAIN statement. A specified input texture image is clamped to the corners of the constructed 3D terrain geometry. Thus, in cases where the specified texture is an actual aerial image of the terrain area, we must ensure the exact lateral correspondence between the image corners and the input DEM.

### 6.2 Terrain Model Manipulation

The algorithms that manipulate ViTerra terrains to visually depict surface deformations in response to performed construction tasks are implemented in the VITASCOPE KineMach add-on. KineMach implements “smart” pieces of virtual construction equipment that can be instantiated and manipulated in visualizations using simple text statements in a higher-level, contextual, construction terminology (see Chapter 5 for details). Engineers can use KineMach provided statements to instantiate multiple pieces of equipment such as backhoes and instruct them to perform virtual construction processes using a high-level construction work terminology.

Figure 8 presents an animation trace with statements that instantiate a virtual backhoe and instruct it to perform construction tasks (i.e. digging and dumping). During the performance (i.e. visualization) of each digging and dumping stroke, KineMach monitors the position of the virtual backhoe’s bucket using the approach described in the previous section. KineMach then internally invokes ViTerra’s elevation-setting statements (TERRAIN.SetElevation) to instantly modify the 3D terrain model at points
corresponding to the backhoe bucket’s trajectory. In addition, at each terrain point modified in response to digging, ViTerra splats (i.e. pastes) a texture image that gives the terrain model an appearance of digging implement ridge marks.

```
LOADADDON ViTerra;
LOADADDON KineMach;

TERRAIN ‘4349CATD.DDF,10,1’ ‘4349Tex.JPG’;

BACKHOE Hoe1 5;
Hoe1.PlaceAt (135,-195);

TIME 0;
Hoe1.DigAt (152,-202) 10;

TIME 10;
Hoe1.DumpAt (140,-177) 10;
```

**Figure 8: Communication of Excavation Tasks**

Figures 9 and 10 present strips of animation snapshots taken during the visualization of the virtual digging and dumping tasks respectively. The snapshots presented are not successive computer frames observed during visualization. The discretely captured frames are displayed in a filmstrip format merely to depict a sense of motion. The smooth motion of the backhoe during visualization and the corresponding, smooth deformation (i.e. evolution) of the virtual terrain cannot be fully captured in static snapshots. Only the animation can convey that information.

### 6.3 Terrain Model Rendering

ViTerra utilizes the Demeter terrain engine for the data structures required to store (i.e. maintain) constructed 3D terrain models on the computer as regular grid height fields. The Demeter library (Fowler 2002) implements data structures and algorithms to visualize vast 3D terrain models in virtual environments using OpenGL (Woo et al. 1997). ViTerra also relies on Demeter to depict a visually accurate and detailed representation of the current 3D terrain database at interactive frame rates. In particular, ViTerra adopts the Demeter-implemented view-dependent, adaptive triangulation algorithm to render the dynamic 3D terrain models it maintains. At each frame in the
visualization, the implemented algorithms render a high-fidelity image of the current 3D terrain model with the fewest possible polygons (i.e. triangles) for that particular view.

Figure 9: Animation Snapshots of Virtual Digging
7. Future Work

DEM data and the uniform grid height field terrain models constructed from that data both describe a terrain surface by sampling elevation values at regularly spaced positions along two perpendicular directions in the horizontal plane. Such a representation works well for relatively flat and even terrains. However, several problems arise when the terrain being modeled is highly irregular. In particular, the uniform grid terrain model is unable to accurately represent steep, vertical cliffs and overhangs in modeled 3D terrains.

Figure 11 graphically presents the source of this problem. In the case of vertical cliffs, sampled positions on the terrain’s surface can each have two different elevations depending on where the elevation values are sampled (i.e. at the cliff’s base or peak). The problem is even more acute in the case of overhanging terrains. In the latter case, sampled points potentially need to be described with multiple elevations to accurately represent the true surface.
The presented work does not consider such extreme cases. In particular, only methods of constructing regular, even 3D terrains from corresponding input DEM data have been designed and implemented. However, such topography can often be encountered on construction sites. Research is ongoing to design DEM formats that can accurately archive such terrain configurations. Future work can capitalize on these new DEM formats and design methods to construct accurate 3D models of dynamic terrain that feature the abovementioned conditions (i.e. cliffs and overhangs).

Another interesting opportunity for future research is improving the physics in describing the interaction between virtual digging implements (e.g. backhoe buckets) and the 3D terrain models. The presented work implements a simple soil-equipment interaction model to compute the deformations that terrain models are subjected to while performing virtual construction. The adopted model has no physical significance (e.g. conservation of dirt) and is intended purely for visual simulation purposes. Recent research (Park 2002) formulates an improved mathematical model of virtual excavator digging and presents physically meaningful soil-bucket interaction techniques for virtual reality equipment simulators. Future work can explore techniques of incorporating such detailed, physics-based models within a simulated construction process visualization framework.

A third interesting research opportunity exists in designing techniques to automate the procedure of matching disparate elevation and imagery data sources in creating 3D terrain models. In the presented work, the task of matching an aerial image with the elevation data used in constructing terrain databases is manual (i.e. the responsibility of the user). In the case of actual aerial imagery textures, the presented terrain construction
methods assume a lateral correspondence between the two (i.e. elevation and imagery) input data sources. However, several problems are encountered in achieving such lateral correspondence.

First, high-resolution aerial and satellite imagery of any desired area of interest is not as readily and freely available as is digital elevation data (Childs 2002). In addition, proper lateral alignment of an aerial image with a DEM database (called GeoReferencing) is difficult to achieve because most of the available aerial images have no reference data such as latitude and longitude or Universal Transverse Mercator (UTM) coordinates to allow for easy correspondence with a particular DEM database. The design of such matching techniques is also an active area on research in other geodetic sciences. Future work can capitalize on this research to facilitate automated matching of elevation and imagery data in 3D terrain model construction.

8. Summary and Conclusions

The presented research extends the state-of-the-art of scientific simulation-driven construction process visualization. The work puts in place the technology that engineers can use to 1) automatically create 3D terrain databases (i.e. CAD models) from archived digital elevation and imagery data, and 2) dynamically manipulate constructed terrain models to visually depict landscape deformations in response to virtual construction processes such as digging and dumping dirt.

The design of techniques to automatically generate dynamic, deformable 3D terrain databases in simulated construction process visualizations requires the integration of four key technologies:

1. Combination of data from disparate digital elevation and imagery data sources into a unified, consistent, and accurate 3D terrain database
2. Efficient data structures (i.e. storage model) to internally maintain and locally manipulate the constructed 3D terrain databases on demand
3. Effective equipment-terrain interaction model to compute the amount, location, and resultant shape of terrain deformations in response to performed virtual construction tasks
4. Fast, efficient rendering procedures to draw the dynamic terrain models in 3D virtual worlds such that the results look visually convincing, and can be animated in real-time

Commonly available DEM elevation data is archived in simple, regular grid numerical or grayscale image height maps. The geometry of a 3D terrain database for any geographic area can be extracted from the readily available DEM elevation data for that region. An actual or synthetic aerial image of the corresponding area can then be superimposed on the constructed geometry to specify the appearance of the constructed 3D terrain model. The availability of input elevation data in regular grid formats presents several distinct advantages. First, it allows straightforward extraction (from DEM archives) of 3D positional coordinates required to construct the geometric meshes that represent the terrain surfaces. This facilitates and guides the critical choice of the storage model to be adopted in maintaining constructed 3D terrain databases in a dynamic, modifiable format on the computer during visualization.

The adopted regular grid height field storage model, in turn, guides the design of the interaction scheme between the 3D terrains and virtual pieces of construction equipment. In particular, the adopted storage model facilitates the design of a simple deformation computation scheme that monitors virtual digging implements during visualization and locally manipulates the terrain model elevations to visually depict the deformed landscape. The regular grid height fields also form the foundation of hierarchical data structures that are required to be built by rendering algorithms for efficiently drawing 3D terrains at interactive frame rates. The implemented view-dependent, dynamic terrain triangulation and rendering algorithm and the visually appealing results it displays demonstrates that the approach is not only possible, but also very effective.
9. References


Chapter 7
Path Generation for Variable-Speed Resource Motion in Animations of Discrete-Event Process Models

1. Introduction
Visualization of simulated operations can be of significant help in the verification and validation of discrete-event process models (Law and Kelton 2000). This is especially true in construction where typical decision makers are experts in their domain but are not generally proficient in simulation itself. Visualization can also provide decision makers with valuable insight into subtleties of planned construction operations that are otherwise non-quantifiable and non-presentable. The necessity to effectively communicate DES models in 3D is the motivation behind ongoing visualization research efforts at Virginia Tech.

These efforts focus on designing automatic, simulation-driven methods to visualize modeled processes and any evolving products in smooth, continuous, dynamic 3D virtual worlds. Methods have been designed to describe animated 3D worlds that show how simulated processes are carried out, using simple parametric text statements and references to 3D CAD models (see Chapter 2 for details). This simple text animation description language, meant to be written out by end-user programmable tools such as DES systems, allows a computer to create a dynamic 3D virtual world that shows people, machines, and/or materials interacting as they perform the modeled processes.

1.1 Research Motivation
Synthetic, process simulation-driven 3D virtual worlds are spatially and temporally faithful to the underlying discrete-event models that author the visualizations. Notwithstanding, the 3D visual representations (i.e. visualizations) of several modeled processes digress in time and space accuracy from the corresponding real-world operations. These discrepancies can be attributed to 1) the inherent characteristics of
DES, and 2) limitations in the current state-of-the-art of scientific, simulated-driven process visualization.

1.1.1 DES Enforced Temporo-Spatial Inconsistencies

In DES, the state of a running model changes only at discrete, but possibly random sets of simulated time points (Schriber and Brunner 2001). These time points are typically the start or end of the model’s activities, and it is only then that a running DES model can communicate with other processes, or perform other actions such as output to an animation trace. A DES model is only concerned about the time instants at which instances of modeled activities begin or end, and chooses to ignore everything (e.g. rate of activity performance) that happens in between. The information that a running DES model can communicate to external 3D animation methods is thus limited to the start times and durations of all activity instances that occur in any simulation run. The animation methods must then use these meager pieces of discrete information to generate a smooth, continuous, dynamic 3D virtual world representation that depicts the modeled activities being performed.

In any modeled activity that involves motion of simulation entities (e.g. a hauling truck, an airplane taking off), the only kinetic property that can be computed from the pieces of communicated activity instance information (i.e. start time and duration) is the average speed of the simulation entity (e.g. truck, airplane) in that particular activity instance (e.g. haul dirt, take off). This is precisely the computation existing 3D animation methods perform in describing the smooth, continuous motion of simulation entities in virtual worlds.

In particular, the motion of a simulation entity is depicted by transforming (i.e. moving) the pertinent instantiated CAD model of the entity at the computed average velocity. The simulation entity-representing CAD model is smoothly and linearly interpolated on a 3D trajectory that represents that entity’s motion path (e.g. haul road, runway) in the communicated activity instance. Thus, a virtual airplane taking off appears to travel down a runway at a constant (average) speed for the activity instance’s sampled time duration.
(i.e. runway occupancy time). Similarly, a loaded truck hauling dirt on a virtual earthmoving jobsite travels at a constant speed for the entire duration of the haul regardless of the grades on the 3D motion trajectory (i.e. virtual haul road) it travels on.

The temporal and spatial accuracy of simulation model-generated dynamic 3D virtual worlds is commensurate with the detail of the communicated information from which the visualizations are created. The existing animation scheme is faithful to DES in that animated activity instances inside a 3D virtual world begin and end at the exact time instants dictated by the visualization-authoring simulation models, with smooth, continuous, “constant-speed” intermediate motion of the involved virtual entities (i.e. simulation objects). Such visualization of simulation objects performing the modeled (and communicated) tasks at constant speed is often sufficient to verify and validate several DES models (see Chapter 9 for details).

However, the depicted constant velocity profiles of moving virtual simulation objects are not an accurate representation of reality. For instance, an airplane taking off on a runway obviously does not travel at a constant velocity. Instead, as figure 1(a) presents, the airplane continuously accelerates as it races down a runway and takes-off. Similarly, as figure 1(b) presents, the typical velocity profile of a truck that hauls dirt is a function of several disparate factors such as engine power, load being hauled, and the rolling resistance and grade of haul road segments. Due to the limited operational information available from underlying DES models, existing methods of animating simulated processes are, however, unable to adopt such realistic velocity profiles in describing the motion of virtual simulation entities. Instead, commensurate with the available pieces of information, simulation objects are assumed to be moving at constant, average speeds with straight-line velocity profiles such as those superimposed on figures 1(a) and 1(b).
This assumption (and portrayal) of constant-speed entity motion can often hinder the validation of modeled processes in cases where the relative segmental speeds of moving simulation entities and/or their acceleration/deceleration influence their evolution and inter-object interactions in a modeled and visualized system (e.g. airport, earthmoving jobsite). In addition, such constant-speed visualization of modeled processes can frequently fail to elicit credibility for simulation models, especially from domain experts and decision makers who are not familiar with the mechanics of DES and/or are skeptical about simulation analyses beforehand.

1.1.2 Animation Method-Caused Spatial Discrepancies

The spatial accuracy of animated simulation model activities is also compromised by the current state-of-the-art’s inability to accurately represent 3D motion trajectories upon which virtual simulation entities travel. As figure 2 presents, existing animation methods represent an entity’s motion path as a single, piecewise-linear, 3D trajectory constructed by connecting two or more 3D coordinates with imaginary (i.e. not visible during visualization) line segments. This single-trajectory representation of a 3D motion path precludes the accurate spatial orientation of virtual simulation entities that traverse that path.
The configuration of any moving or static object in 3D virtual space is fully described by four parameters. These are the object’s position (represented by a 3D coordinate), horizontal rotation (yaw), vertical rotation (pitch), and side rotation (roll). As figure 3(a) depicts, an object’s yaw is simply the horizontal direction in which it faces as it travels or stands still. The pitch of an object is similarly the amount by which it faces up or down and is a function of the grade of the surface on which it stands or travels (figure 3(b)). Finally, as figure 3(c) presents, an object’s roll is the side-to-side tilt it experiences when standing or traveling on an uneven surface (e.g. cars banking on superelevated curves).

A single trajectory motion path provides a geometric basis for describing only three configuration parameters for objects that traverse that path. In particular, as figure 2 presents, the position, yaw, and pitch of an object traveling on a single trajectory path can...
be determined by straightforward interpolation along a path’s trajectory. At any given instant, the position of an object traversing such a path is a 3D coordinate that lies on one of the path’s linear segments. A traveling object’s yaw is simply the horizontal orientation of the path’s current line segment (figure 2(a)). Similarly, the vertical orientation of the path’s current line segment (figure 2(b)) directly provides the pitch for an object that traverses the path.

This single trajectory 3D path representation scheme, however, provides no geometric basis for determining the side roll of objects that traverse that path (figure 2(c)). As such, existing animation methods conservatively assume a zero side roll for all objects that traverse 3D motion trajectories in visualizations. This assumption (and portrayal) of simulation entities traveling on 3D motion paths without any side-to-side tilting does not adversely influence the visualization of all modeled systems. In particular, systems where the motion paths of simulation entities lie on largely even terrain and are defined by only horizontal and vertical components (e.g. airport runways and taxiways), the inability to visually depict simulation entities tilting sideways is not significantly noticeable and does not hinder the verification and validation of the underlying simulation model’s logic.

However, in visualizing modeled systems such as earthmoving jobsites, this limitation causes objectionable visual artifacts and undermines visualization’s ability to help verify and validate simulation models. In particular, earthmoving equipment such as dozers, backhoes, and haulers operate on highly uneven terrain whose topography itself can change dramatically as operations progress. In such environments, the involved equipment pieces negotiate several uneven heaps of dirt as they perform work causing the equipment’s frame to continuously tilt along all three axial directions. As figure 4 presents, the inability to accurately compute and depict an object’s side roll as it travels in animations of such modeled processes results in the depiction of highly unrealistic visual artifacts.
In addition, the fact that motion trajectories are represented as concatenated straight lines further undermines the accurate visual depiction of several modeled processes. Simply stated, few entities in real life move on paths that resemble connected straight line segments. Most motion paths traversed by real-world entities include linear as well as smooth, varying-degree curved segments. The current segmental straight-line, single-trajectory 3D path notation and the incorrect object motion portrayal it enforces precludes the visual validation of several modeled processes. In particular, existing animation methods are unable to visually elicit much credibility when issues such as space constraints (e.g. turning space), visibility, and/or maneuverability must be validated through 3D visualization.

1.2 Main Contributions

The research presented here directly addresses all the abovementioned issues. First, we design simple, parametric text, simulation model-authorable methods to describe and manipulate curved trajectories of any desired shape and length to represent 3D motion paths in visualizations. This addresses the problem of accurately describing the profiles of routes that typical simulation entities traverse as they virtually perform communicated simulation tasks. The work capitalizes on a technique of producing a very general class of
interpolating cubic splines whose shape can be locally or globally controlled by simply modifying three high-level control parameters.

Second, we implement methods to accurately describe the three-dimensional spatial configuration of virtual simulation entities as they travel on 3D motion trajectories in animations. By capitalizing on an innovative virtual terrain-following algorithm, we implement a computation scheme that correctly calculates and portrays the orientation of simulation objects along all three axes (i.e. yaw, pitch, and roll) as they travel or stand still inside animated 3D virtual worlds.

Finally, we present an innovative technique that allows engineers to portray variable-speed motion of virtual simulation objects as they travel along 3D motion paths, using only a one-time, simulation model-authorable parametric text definition of the desired velocity profile and per-instance activity timing information (i.e. start time and duration). The technique employs a unique computation scheme that prudently gleans discrete pieces of operational information from an underlying simulation model and processes it to describe a realistic velocity profile that is unique to the traveling object/3D path pair in the particular communicated activity instance. This, combined with the accurate spatial orientation of simulation entities on realistic motion trajectories significantly enhances the efficacy of 3D visualization in verifying, validating, and accrediting DES models.

2. Challenges

The design of techniques to address the problem of describing the accurate, variable-speed motion of simulation objects on realistically shaped motion trajectories in visualizations of simulated processes presents numerous interesting challenges. The shape of a trajectory that a virtual simulation object must travel on during visualization is arbitrary (i.e. depends on the configuration of the system being modeled and the pertinent simulation object) and cannot be defined and/or stored a-priori. The trajectories on which simulation objects move during the visualization of a DES model must thus be defined by the model itself using parametric text-based animation methods that are both simple (so
that a DES model can author them) and powerful (so that they can flexibly describe and manipulate a 3D motion trajectory of any complexity).

A simulation object traveling on a defined motion trajectory must be correctly oriented along all three coordinate axes in order for its motion to appear accurate and convincing. A motion path defined by the commonly used single 3D trajectory notation, however, encapsulates no geometric information that can guide the computation of a simulation object’s complete orientation as it traverses potentially uneven virtual terrain. While a simulation object must always be guided by the motion trajectory it travels on, its interaction with the surface of the virtual terrain that lies beneath must also be realistically described in order to accurately portray that object’s motion. This is particularly important in cases where the surface of the terrain itself deforms dramatically over the course of visualized simulation processes.

DES models that author 3D visualizations only provide information (to the animation methods) about when activity instances begin and end. The models can provide no guidance on the rate at which the tasks in communicated activity instances are performed. Such information is however required to describe the realistic, uneven velocity profiles with which simulation objects move in reality. If such information is formulated externally (i.e. outside DES models), we must ensure that the temporal integrity of the underlying DES models is not violated i.e. any externally computed temporal variables must perfectly coincide with the underlying DES model’s event times (activity start and end times).

The issues we address in this work are thus three-fold. First, we examine techniques of mathematically describing arbitrarily complex curves that can be easily edited locally and/or globally to describe a realistic motion trajectory of any shape and length. Second, we investigate geometric approaches of describing the interaction of moving simulation objects with virtual 3D terrains in order to accurately compute and depict their spatial configuration during motion. Finally, we explore techniques of compensating for the lack of operational information (about activity instances) available from DES models and
design methods to move objects with uneven, arbitrarily-shaped velocity profiles that comply with a DES model’s temporal integrity.

3. Representing Accurate Motion Trajectories

In 3D animation, spatial trajectories are generally used for two common purposes: 1) to define paths over which virtual objects move, and 2) to define routes for the user’s viewpoint (i.e. avatar) to travel over. The latter is equivalent to moving the imaginary scene camera over a path and is thus functionally equivalent to a path-traversing virtual object. A 3D path can be defined either as a piecewise-linear trajectory of joined straight-line segments or as a smooth trail of joined, continuous curved segments. In addition, a 3D path may also be defined by a combination of such straight-line and curved segments.

In either case, a path is defined by specifying a set of control points (i.e. coordinates) through which the constructed linear and/or curved trajectory passes. When an object travels along such a constructed path, the computer calculates its intermediate (i.e. between any two control points) positions by interpolating between pairs of path control points. This supports the classical technique of computer animation that is often referred to as key-framing (Watt and Watt 1992). A path’s control points explicitly specify the position of moving objects at certain key animation frames and leave it to the computer to calculate the rest of the intermediate positions through a chosen interpolation scheme.

Existing methods of animating simulated processes exclusively define motion paths as piecewise-linear trajectories. In such cases, straightforward linear interpolation between control points is used to calculate a moving object’s intermediate positions as it travels along a path. In addition, the moving object’s yaw and pitch are simply set to the horizontal and vertical slope of the path’s current linear segment. The most objectionable outcome of linearly interpolating intermediate object positions between path control points (i.e. using piecewise-linear paths) is the lack of smoothness in the motion (Kochanek and Bartels 1984). In particular, as figure 5 depicts, a moving object can be potentially subjected to sudden changes in direction as it transitions from one linear segment to the next.
Current methods of animating simulated processes partially address this issue by utilizing the concept of the Rear Guide Point (RGP). Instead of specifying an object’s position with a single 3D coordinate, an additional point on the object (called the RGP) is chosen such that it trails the object’s local origin by a specified amount. Then, when that object traverses a path, the computer is instructed to ensure that both the object’s local origin and its RGP lie on the path at all times. As figure 6 presents, this reduces the sudden directional changes a moving object might undergo at linear path segment junctions.
In the case of curved path segments, intermediate positions of traveling objects can be computed by fitting a set of interpolating splines through the specified path control points. An interpolating spline in this context is a function that is defined on an interval to approximate a smooth curve passing through the specified path control points with a suitable degree of smoothness (Smith 1983). The moving object’s yaw and pitch, in this case, can be deduced by computing the smooth curve’s tangent in the projected horizontal and vertical planes respectively. Such an approach almost always results in a more accurate path representation for the pertinent mobile object. In addition, as figure 7 presents, the depicted object motion that can be achieved is significantly smoother when compared to that on segmental linear paths passing through the same set of control points.
Mathematically, a spline is a piecewise polynomial satisfying continuity conditions between the pieces (Farin 1997). It is a technique of generating a single geometric object (i.e. a continuous curve) from pieces. Several different types of splines exist in the body of mathematical knowledge. However, few of them are suitable for representing motion paths in computer animation (Smith 1983). Piecewise cubic polynomials are particularly suitable for representing 3D motion trajectories. Such curves that consist of a succession of different cubic polynomial segments joined together with certain continuity constraints are called Cubic Splines (Bartels et al. 1987) with the curve’s control points often being referred to as knots or knot values.
A parametric, three-dimensional curve such as that defined by a spline segment is, by definition, a function of a single variable and can be mathematically stated with three univariate functions as \( Q(u) = (X(u), Y(u), Z(u)) \), where \( 0 \leq u \leq 1 \). As the variable \( u \) varies from 0 to 1, the functions traverse the defined curve segment in 3D space. To define a spline curve for a range of values for the parameter \( u \), say from 0 to 3, we simply splice together curve segments defined over intervals of values for \( u \) (e.g. 0 to 1, 1 to 2, and 2 to 3). This process is graphically presented in two dimensions in figure 8. The four points (i.e. \( p_0, p_1, p_2, \) and \( p_3 \)) then form the control point set (i.e. knots) of the resulting piecewise, continuous curve.

![Figure 8: Piecewise Cubic Spline Curve](image)

Since a cubic spline is a polynomial of degree 3, we can also write its description mathematically as \( Q(u) = a_0 + a_1u + a_2u^2 + a_3u^3 \). In defining 3D motion trajectories, it is however inconvenient to represent spline curve segments directly using the coefficients \( a_0 \) to \( a_3 \) because the relationship between the shape of the defined curve and the polynomial coefficients is not clear and intuitive (Watt and Watt 1992). In particular, it is challenging to define a curve of the desired 3D path segment shape (i.e. control endpoints and derivatives) using a spline polynomial’s coefficients.

In order to avoid manipulating polynomial coefficients directly to edit the resulting curve’s shape and to make path shape-editing more intuitive and straightforward, a spline’s polynomial form can be rearranged in terms of the user-specified path control
points and a collection of linearly independent polynomials called Basis Functions. Thus, the spline segment’s cubic polynomial \( Q(u) = a_0 + a_1 u + a_2 u^2 + a_3 u^3 \) can be restated as
\[
Q(u) = \sum_{i=0}^{3} p_i b_i(u) = \sum_{i=0}^{3} p_i b_i(u),
\]
where \( p_i = (x_i, y_i, z_i) \) are the curve segment’s knots (2 endpoints and their adjoining knots) and \( b_i(u) = u^i \) are its basis functions.

A spline’s basis functions can be interpreted as weights by which its knots will be multiplied to determine points along the resultant curve. Different choices of basis functions generate different spline curves for the same set of knot points (Farin 1997). A spline’s basis function set (collectively called its basis) thus governs how a specified control point set influences the resultant curve and the flexibility with which that curve’s shape can be edited. Different bases may have certain properties that are useful in different contexts. A spline’s basis defines its spatial character and its choice depends on the features required of the resulting curves.

Spline curves intended to describe arbitrary, 3D motion trajectories must possess the following characteristics:

1) Versatility, so that arbitrarily complex 3D trajectories can be flexibly defined by specifying a finite set of control points
2) First order continuity between pieces, so that the resulting spline passes through all the specified control knot points
3) Local control, so that changes to curve segments (i.e. pieces) can be locally made without having significant distorting effects on remote pieces
4) Computational efficiency, so that they can be readily adopted in CPU-intensive, dynamic virtual environments

In attempting to identify an appropriate spline basis that satisfies all the abovementioned constraints in the resulting curves, we found most promise in a technique that produces a very general class of interpolating cubic splines whose shape can be locally or globally controlled by simply modifying three high-level control parameters. This class of smooth,
flexible curves, called Kochanek-Bartels splines use Hermite interpolation basis functions and were specifically designed to model 3D animation paths (Kochanek and Bartels 1984).

### 3.1.1 Suitability of Kochanek-Bartels Splines

Given a set of control points \( \{p_i\}_{i=0}^{N-1} \), the Kochanek–Bartels splines implement a cubic interpolation between each pair \( p_n \) and \( p_{n+1} \) with varying, user-definable properties specified at the endpoints. These high-level, per-knot curve shape-governing properties are called the Tension, Continuity, and Bias (collectively referred to as TCB). These splines are therefore often referred to as TCB splines.

A knot’s tension controls how sharply the curve bends at that control point. The continuity parameter provides a smooth visual variation in the curve’s continuity at a knot. For instance, a continuity value of zero yields derivative continuity at a knot point, whereas non-zero values yield discontinuities. A knot’s final parameter, its bias, controls how the direction of the curve at that knot changes by varying the weighted combination of one-sided derivatives at that control point. Valid values for a spline’s tension, continuity, and bias generally range from -1.0 to 1.0 for each setting, with fractional values being acceptable.

The Hermite interpolation basis used in driving Kochanek–Bartels (i.e. TCB) splines is given by the functions

\[
H_0(u) = 2u^3 - 3u^2 + 1, \quad H_1(u) = -2u^3 + 3u^2, \quad H_2(u) = u^3 - 2u^2 + u, \quad \text{and} \quad H_3(u) = u^3 - u^2.
\]

A parametric curve passing through any pair of control points \( p_n \) and \( p_{n+1} \) with tangent vectors \( T_n \) and \( T_{n+1} \) is then given by:

\[
Q(u) = H_0(u)p_n + H_1(u)p_{n+1} + H_2(u)T_n + H_3(u)T_{n+1}
\]

As figure 9 presents, in the case of intermediate endpoints (e.g. \( P_2 \)), the components of the tangent are derived from the adjoining knots on either side of the curved segment. In the case of extremities (i.e. knot pairs containing the first and/or last control point), the boundary conditions may be defined by including the extreme points twice (Eberly...
2001). Optionally, additional control points may be specified at the ends. These scenarios are graphically presented in figure 9 for knots $P_0$ and $P_5$ respectively. In the latter case, $P_6$ is a dummy knot defined merely for specifying the outgoing component of the tangent at $P_5$.

![Figure 9: Deriving Tangent Components at Control Points](image)

The spatial effect of modifying the TCB values is largely intuitive, changing the spline’s shape in a manner consistent with many naturalistic motions. Figure 10 presents a montage of a TCB spline curve that describes the effects of varying each of these parameters at a control point. The tension at a knot is controlled by varying the length of both the incoming and outgoing parts of the tangent vector equally. A knot’s continuity is changed by allowing the source and destination components of the tangent vector to differ as a function of the specified continuity parameter. Finally, a knot’s bias is manipulated by assigning different weights (default being 50:50) to the chords from which the incoming and outgoing parts of the tangent are constructed. The full mathematical formulation of the TCB factors is very interesting reading that is described in complete detail in Kochanek and Bartels (1984).
A spline knot’s TCB parameters control both the shape of the curve and the parametric spacing around that control point. Thus, modifications to any knot parameters only have an effect on the shape of two adjacent spline segments in the local neighborhood of that knot. This allows flexible local curve-shape control, making TCB splines highly suitable for our purpose of defining smooth, arbitrarily complex, customizable 3D trajectories to represent object motion paths in visualizations of simulated operations.

3.1.2 Parameterization by Arc Length

In moving an object along a 3D trajectory, we are concerned with placing that object at desired downstream distances along the path as dictated by the object’s velocity profile (i.e. key-framing). In the case of piecewise-linear trajectories, such key-framing is straightforward to achieve because the linear trajectories are naturally parameterized by the length along the straight-line path segments.

However, the natural parameterization of splines does not advance uniformly with respect to distance along the curve. In particular, different segments of a spline with the same
parametric length can have different physical curve lengths. For instance, as figure 11 presents, a point on a curve segment obtained by choosing a value of 0.5 for the parameter $u$ does not necessarily lie halfway along that segment in terms of traversed distance ($s$) along the curve. In general, equal steps in $u$ result in unequal distances ($s$) traveled along the generated curve i.e. $u_2 - u_1$ is not proportional to $s(u_2) - s(u_1)$, where the arc length of the curve, $s(u)$, is given by 

$$s(u) = \int_{u_1}^{u_2} \sqrt{x_u'^2 + y_u'^2 + z_u'^2} \, du.$$ 

Figure 11: Relation between Natural Spline Parameterization and Arc Length

Distance-based control is however essential for animating objects (with desired velocity profiles) on 3D curved trajectories. We address the abovementioned problem by creating a parameterization where the parametric distance is proportional to the arc length along the curve. In other words, we re-parameterize a TCB spline in terms of the curve arc length by following three steps. Given a spline $Q(u) = (X(u), Y(u), Z(u))$, we first numerically compute the arc length of the spline as a function of $u$: $s = A(u)$. We then find the inverse of $A(u)$: $u = A^{-1}(s)$. Finally, to implement a motion path parameterized by arc
length \((s)\), we substitute \(u = A^{-1}(s)\) into \(Q(u)\) to give \(Q(s) = Q(A^{-1}(s))\). By allowing retrieval of points (i.e. 3D positions) at any specified downstream distance along a spline, this re-parameterization provides the necessary control required to move objects with any desired velocity profile along the smoothly-curved motion paths.

### 3.2 Describing 3D Paths in 2D

A flexible curve such as a TCB spline can theoretically describe a 3D trajectory of any desired shape and length to represent a motion trajectory. However, several other issues adversely influence their unmodified adoption to represent 3D motion trajectories in animations of simulated processes. In particular, the direct adoption of 3D splines to describe motion trajectories in animations results in many visual artifacts that are particularly objectionable when the virtual terrain (e.g. construction jobsite, factory floor) on which the modeled processes are performed is irregular (i.e. not flat). The issues pertain to the correct 3D orientation of simulation objects that traverse defined motion trajectories and are equally applicable to piecewise-linear and/or curved paths.

The first issue concerns the computation of an object’s side roll as it travels along a path. As described in an earlier section, a path represented as a single 3D trajectory provides no geometric basis for computing the side roll of virtual simulation objects (figure 2(c)). The second problem concerns the selection of control points to define a motion path trajectory. The manual identification and selection of 3D coordinates (i.e. knots) that all lie exactly on an uneven terrain surface is a challenging prospect.

In addition, as figure 12(a) presents, the choice of knots that all float on a terrain surface does not guarantee that the trajectory (linear and/or curved) that passes through those knots also drapes the terrain. Finally, the topography of several terrain surfaces (e.g. construction jobsites) can change dramatically during animation of modeled construction tasks such as digging. In such cases, a pre-defined 3D motion trajectory can lose its integrity during the course of visualization as presented in the example scenario in figure 12(b).
When any object travels on a motion path, its position is governed by the path’s trajectory at all times (i.e. the object’s position lies on the trajectory). However, the object’s orientations (yaw, pitch, and roll) are affected by both the trajectory’s curvature as well as the geometry of the terrain surface on which the object travels. In particular, how an object is oriented at any point on a terrain surface depends on the geometry of the relief and on the location of surface contact points (e.g. tires, crawlers) of the object. In order to correctly orient traveling objects in 3D, we must thus base our computation on inputs derived from the motion path trajectory as well as the terrain surface geometry.

We designed such a computation scheme by allowing the definition of motion path trajectories in planar resolution (i.e. in 2D). We then superimpose the defined 2D trajectory onto the geometry of a virtual terrain surface over which the modeled processes are to be animated. This process, presented graphically in figure 13, directly addresses the object orientation issues identified above. In particular, the technique alleviates the need to manually identify 3D path control points by automatically projecting the 2D knots onto a terrain surface. In addition, the path superimposition ensures that any position on the motion trajectory lies on the virtual terrain’s surface. Since the superimposition can be computed on demand in real-time, the integrity of a 3D trajectory is maintained if and as the terrain surface undergoes deformations.
The scheme also provides a geometric basis for accurately computing all parameters of an object’s 3D configuration. The yaw of any moving object is now fully governed by the curvature of the parameterized 2D trajectory in the horizontal plane. The object’s 3D position is determined by projecting its location on the 2D trajectory onto the terrain surface. Finally, the pitch and the roll of the object are computed by projecting the contact points (e.g. tires, crawlers) of the positioned and horizontally-aligned object onto the surface of the virtual terrain. This process is commonly referred to as Terrain-Following in computer animation parlance (Barrus and Waters 1997).

4. Describing Variable Speed Motion

The only temporal information about an activity-instance that a running DES model can communicate to external processes such as 3D animation methods is that instance’s start time and its duration. Due to inherent modeling features, a DES model can provide no
information on the rate at which the task(s) in any activity instance are performed. This is unlike Continuous Simulation, where the state of a model (and hence the rate of activity performance) is continuously monitored at every time instant using differential equations of motion (Law and Kelton 2000). The performance rate of an activity, however, provides precisely the temporal information needed to describe the velocity profile(s) of simulation object(s) that move (i.e. travel) while performing the task(s) in a particular activity instance.

4.1 Hypothetical Hybrid Animation Approach

In order to describe variable-speed motion of virtual DES objects then, the first possible technique we considered exploring was to describe a parallel continuous simulation system that would tightly integrate with the methods of animating DES models in 3D. In particular, we considered the possibility of externally formulating an in-context simulation object’s pertinent kinetic properties (e.g. nominal acceleration/deceleration, maximum permitted velocity) and using that information along with each communicated activity instance (i.e. start time and duration). During visualization, the integrated continuous simulation mechanism would use the formulated kinetic properties to compute the involved simulation object’s velocity profile (i.e. its temporal evolution) beginning at the indicated activity instance start time.

This hybrid animation approach is, however, impossible to achieve in a DES framework. In particular, such a strategy would only work if a DES model communicated an activity instance’s start time and enforced no restrictions on when it ended. That information (i.e. activity instance end time) could then be determined in real-time as the involved simulation object’s temporal evolution was computed continuously during visualization. A DES model must, however, explicitly enforce a communicated activity instance’s end time (i.e. its duration). This is obvious because the start of instance(s) of other simulation model activities (often involving the object that is in context in the current activity instance) is explicitly tied to the completion of the current communicated activity instance. Once instantiated, any instance of such a successive activity will attempt to
exclusively manipulate the in-context virtual simulation object to visually describe the performance of that latter communicated task.

The hypothetical continuous simulation system computing the simulation object’s temporal evolution in the instance of the previous activity would, however, be unable to guarantee the completion of its motion at the exact precise instant at which the successive activity starts. Stated differently, it is mathematically impossible to externally formulate kinetic object properties and compute a unique, valid, continuous velocity profile using a set of differential equations if the motion start and end times (i.e. the lower and upper bound of the integration interval) and the distance traversed (i.e. the area under the resulting curve) are both explicitly enforced.

This is however the case in DES. As such, any computation scheme (for visualizing simulated processes) that wrests the temporal and spatial control of simulation objects away from the underlying DES models cannot portray the modeled operations correctly in dynamic 3D virtual worlds. The description of any arbitrary velocity profiles to be applied to mobile simulation objects during animation must thus be sought from the DES models that author visualizations. However, a DES model obviously does not encapsulate any such information (e.g. an object’s kinetic properties) simply because the rate of performing any modeled activity is generally irrelevant to the model from the simulation analysis perspective.

4.2 Time-Based Scaling of Velocity Profile Shapes

In order to visually describe variable speed motion of animated simulation objects then, we devised a unique computation scheme that prudently shares the responsibility of describing a moving simulation object’s arbitrary velocity profile between the underlying DES model and the 3D animation methods. In particular, the general shape of the velocity profile to be applied to a moving object is sought from the visualization-authoring DES model. The shape of this general profile can be explicitly defined (and input into a DES model) by a modeler, or it can be the result of computations performed within a running model.
Then, at each communicated activity instance, the animation methods heuristically scale
(up or down) the previously defined velocity profile in such a way that the mobile
simulation object to which it is applied traverses an indicated motion path in a time
interval that is exactly equal to the communicated activity instance duration. This flexible
technique allows for two vital things: 1) Any arbitrarily shaped velocity profile resulting
from any DES model-defined or externally performed computation can be explicitly
applied to a moving simulation object, and 2) The temporo-spatial control of all
simulation objects remains entirely with the underlying simulation model since the
animation methods merely scale a defined velocity profile to fit the duration of a
communicated activity instance.

For any defined motion path trajectory, a simulation model defines the shape of a desired
object velocity profile by specifying an arbitrary number of velocity-distance pairs. The
specified velocity values can span any positive numerical range and the corresponding
indicated distances are the percentile (0 to 100) arc lengths along the path. The shape of
the profile is deduced by plotting the path’s percentile arc distance on the abscissa and
the corresponding velocity values on the ordinate. This is graphically presented in figure
14. No limitations are placed on this definition except that the distance value in the last
specified velocity-distance pair must equal 100 percent (i.e. the current path’s total arc
length).

As a first pre-processing step, the defined velocity versus percentile distance profile is
converted to a velocity versus actual distance curve. As figure 15(a) depicts, this is
accomplished by simply replacing the percentile arc lengths on the abscissa by the
corresponding actual arc distances for the current path. The specified velocity values are
left unchanged. The total time $T_o$, required to traverse this converted profile in its
unmodified form can be given by:

$$
T_o = \frac{S_1}{V_{o_1}} + \frac{S_2}{V_{o_2}} + \frac{S_3}{V_{o_3}} + ... \frac{S_n}{V_{o_n}} = T_{o_1} + T_{o_2} + T_{o_3} + ... T_{o_n}
$$
Figure 14: Definition of Velocity Profile Shape

As figure 15(a) indicates, $V_{o_1}, V_{o_2}, V_{o_3}, \ldots V_{o_n}$ are the average velocities at which the respective path segments $S_1, S_2, S_3, \ldots S_n$ are traversed. In the case of this converted, unmodified curve, the total original travel time works out to be 195.66 seconds. Now, for any communicated activity instance of duration $T_i$ (say 150 seconds), we segmentally scale the described velocity profile up or down such that a simulation object traveling that path with the resulting modified (i.e. scaled) velocities reaches the end of the path in exactly $T_i$ time units. This is graphically depicted in figure 15(b). We describe this scaling procedure as:

\[
\frac{(T_{i_1} + T_{i_2} + T_{i_3} + \ldots T_{i_n})}{T_i} = \frac{(T_{o_1} + T_{o_2} + T_{o_3} + \ldots T_{o_n})}{T_o}
\]

\[
\therefore (T_{i_1} + T_{i_2} + T_{i_3} + \ldots T_{i_n}) = \frac{T_i}{T_o} (T_{o_1} + T_{o_2} + T_{o_3} + \ldots T_{o_n})
\]

\[
= T_{o_1} \left( \frac{T_i}{T_o} \right) + T_{o_2} \left( \frac{T_i}{T_o} \right) + T_{o_3} \left( \frac{T_i}{T_o} \right) + \ldots T_{o_n} \left( \frac{T_i}{T_o} \right)
\]
Figure 15: Derivation of Time-Scaled Velocity Profiles

(a) Original Velocity Profile

(b) Adjusted Velocity Profile
The underlying assumption we make in segmentally scaling the original converted velocity profile (figure 15(a)) to accommodate the currently specified activity instance duration (figure 15(b)) is that

\[
T_i = T_o \left( \frac{T_i}{T_o} \right), \quad T_i = T_o \left( \frac{T_i}{T_o} \right), \quad T_i = T_o \left( \frac{T_i}{T_o} \right),
\]

… \( T_i = T_o \left( \frac{T_i}{T_o} \right) \). With this assumption, the equation that distributes the communicated activity instance duration over the different velocity segments can be written as:

\[
T_i = T_i + T_i + T_i + \ldots T_i = \frac{S_1}{V_i} + \frac{S_2}{V_i} + \frac{S_3}{V_i} + \ldots + \frac{S_n}{V_i}
\]

The fact that \( T_i = \frac{S_1}{V_i}, \quad T_i = \frac{S_1}{V_i}, \quad T_i = \frac{S_1}{V_i}, \quad \ldots \quad T_i = \frac{S_1}{V_i} \) provides the basis for computing the average segmental velocities \( V_i, V_i, V_i, \ldots V_i \), from which scaled, modified, activity instance-specific velocity profile can be constructed (figure 15(b)). A simulation object that follows this modified velocity profile is guaranteed to traverse the path in the exact communicated activity instance duration \( T_i \).

We now describe our implementation of all the described algorithms and techniques. The implementation is a powerful tool that allows engineers to accurately describe the 3D motion of virtual DES objects on realistic-looking, smoothly-curved motion trajectories.

### 5. PathFinder

The algorithms that allow engineers to define flexible, smooth, curved motion path trajectories and then move virtual simulation objects on those trajectories with desired velocity profiles are implemented as a powerful software tool named PathFinder. This tool has been designed as an extension to the VITASCOPE visualization system. VITASCOPE is a user-extensible 3D animation language designed specifically for visualizing simulated processes (particularly construction operations) in smooth, continuous, dynamic 3D virtual worlds (see Chapter 2 for details).
### 5.1 Defining Motion Paths and Velocity Profiles

The PathFinder add-on extends the VITASCOPE animation language by appending the statements presented in table 1 to its vocabulary.

#### Table 1: Usage of PathFinder-Implemented Animation Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH2D [CurvedPathName] SPLINE [ControlPoints];</td>
<td>Define a TCB spline trajectory passing through the specified knots</td>
</tr>
<tr>
<td>[CurvedPathName].SetTension [KnotNumber] [Value];</td>
<td>Modify the Tension parameter of a particular path control point</td>
</tr>
<tr>
<td>[CurvedPathName].SetContinuity [KnotNumber] [Value];</td>
<td>Modify the Continuity parameter of a particular path control point</td>
</tr>
<tr>
<td>[CurvedPathName].SetBias [KnotNumber] [Value];</td>
<td>Modify the Bias parameter of a particular path control point</td>
</tr>
<tr>
<td>[CurvedPathName].SetVelocityProfile [VelocityPctDistancePairs];</td>
<td>Set the shape of the default velocity profile for objects traversing this path</td>
</tr>
<tr>
<td>PATH2D [LinearPathName] LINEAR [ControlPoints];</td>
<td>Define a piecewise linear trajectory passing through the specified control points</td>
</tr>
<tr>
<td>[LinearPathName].SetVelocityProfile [VelocityPctDistancePairs];</td>
<td>Set the shape of the default velocity profile for objects traversing this path</td>
</tr>
<tr>
<td>MOVE [ObjName] [PathName] [Time] [VelocityPctDistancePairs]opt;</td>
<td>Move a simulation object on a defined path in the specified time units. Optionally override the path’s default velocity profile shape</td>
</tr>
</tbody>
</table>
In particular, the add-on defines parametric text animation language statements that allow the definition and manipulation (in 2D) of smooth, curved motion trajectories of arbitrarily complex shapes. PathFinder also implements statements that allow simulation models to 1) specify a default per-path arbitrary velocity profile shape, and 2) override (if necessary) the default velocity profile shape for any communicated instance of simulation object motion. The detailed usage of all statements is presented in appendix K.

In addition, PathFinder also implements the algorithms that superimpose defined 2D paths onto the virtual 3D terrain, the terrain-following techniques that help orient moving objects correctly in 3D, and the velocity profile scaling methods that heuristically resize the current velocity profile shape to accommodate the duration of a communicated activity instance. PathFinder thus presents the technologies that allow engineers to accurately describe the 3D motion of virtual discrete-event simulation objects on realistic-looking, smoothly-curved motion trajectories.

Figure 16 presents an animation trace with statements that define a smooth motion trajectory and then move an object over it with an indicated velocity profile. We define motion trajectories in 2D planar resolutions by specifying a series of 2D (i.e. \(x\) and \(z\) coordinate only) control points and then manipulate (if needed) the shape of the resultant spline curve by adjusting the TCB values of one or more knots. A default velocity profile (if any) can also be specified as part of a path’s definition. In the absence of an explicitly indicated velocity curve for a path, PathFinder assumes a default, constant-speed profile for objects that traverse that path. The defined 2D trajectory is then superimposed on the 3D terrain model that represents the underlying simulated system’s landscape (figure 13).

Simulation objects can obviously override a path’s default profile during an activity instance (i.e. motion) communication. This can, for instance, allow simulation models to specify a unique velocity profile that is a function of the properties (e.g. engine power, loaded mass) of the in-context simulation object (e.g. dumptruck) in a communicated instance of an activity (e.g. haul dirt). When a simulation object is instructed to move on a particular path in the specified simulation time units, the current (path default or object
overridden) velocity profile is appropriately scaled such that the time to traverse the path with that profile is equal to the communicated activity instance duration. The simulation object then traverses the superimposed 3D path trajectory while following the terrain surface as closely as possible.

```
LOADADDON PathFinder;

PATH2D Path1 SPLINE '(55,-187)' '(25,-25)'
'(75,-30)' '(155,-187)';

Path1.SetVelocityProfile
'(0,0)' '(2,20)' '(2,75)' '(0,100)';

Path1.SetTension 1 1;
Path1.SetContinuity 2 0;
Path1.SetBias 3 0.5;

TIME 0;
CLASS Dozer Deere450H.csb;
CREATE Dozer1 Dozer 1.5 1.5 0 2;
PLACE Dozer1 ON Path1;

TIME 5;
MOVE Dozer1 Path1 230
'(0,0)' '(2.5,24)' '(2.5,60)' '(0,100)';
```

**Figure 16: Moving Simulation Objects on Defined Motion Paths**

Figure 17 presents a strip of animation snapshots taken during the visualization of the motion statement from the animation trace in figure 16. The 3D projection of the described 2D trajectory passes through highly uneven terrain. However, as the snapshots depict, PathFinder’s terrain-following algorithms ensure that the object (i.e. dozer) is correctly oriented on the terrain as it travels the path with the scaled velocity profile. The snapshots presented are not successive computer frames observed during visualization. The discretely captured frames are displayed in a filmstrip format merely to depict a sense of motion. The smooth motion of the dozer during visualization and its non-constant velocity cannot be fully captured in static snapshots. Only the animation can convey that information.
Figure 17: Animation Snapshots of Terrain-Followed Object Motion
5.2 Describing Object Motion with Terrain-Following

Defined and modified 2D motion path trajectories are converted to their 3D representation by projecting them onto the 3D terrain model that describes the simulated system’s landscape as depicted in figure 13. At any instant, a moving object’s current, adjusted velocity profile dictates the downstream distance on a path at which it is currently located. Given the current downstream distance, the object’s planar position (i.e. $x$ and $z$ coordinate) is determined by parametrically retrieving (on the defined 2D path) the point which corresponds to that arc length (i.e. downstream distance). Then, the object’s current yaw is calculated by simply computing the tangent (in the 2D plane) to the curved path trajectory at that determined position (i.e. $x$ and $z$ coordinate).

To determine the 3D position of the object inside the virtual world, the computed planar position point is projected on the 3D terrain model of the simulated system’s landscape i.e. on the superimposed 3D path. This is accomplished by retrieving the elevation (i.e. height) of the terrain model at that horizontal plane location i.e. we retrieve the $y$ coordinate of the terrain point that corresponds to the $x$ and $z$ values of determined 2D position. This describes the traveling object’s current 3D position at that time instant. Since this computation is performed dynamically at visualization run-time, the 2D path’s 3D projection always drapes the terrain surface even if its shape deforms during animation i.e. the procedure always retrieves the current terrain elevations (heights) below 2D position points.

Given the 3D position and yaw of the moving object on the terrain surface, the goal of terrain-following is to now orient that object correctly (i.e. compute the pitch and roll) such that its virtual contact points (e.g. a truck’s tires) all touch the terrain’s surface as closely as possible. To keep moving objects correctly oriented on a terrain’s surface, it is necessary to find the locations where that object’s contact points touch the virtual terrain. This can be done using geometric collision detection techniques. However, this is inefficient because general collision detection inherently involves more complex, CPU-intensive computations than merely computing an object’s terrain contact points (Barrus and Waters 1997).
To enforce terrain-following in a moving simulation object, we adopt a generalized technique that systematically computes that object’s pitch and roll by projecting its contact points on the terrain model in a manner similar to that used in computing the 3D position. In particular, the 2D positions corresponding to an object’s surface contact points (e.g. tires, crawler edges) are projected onto the 3D terrain model to determine the positions where they intersect the surface.

The pitch of the object is calculated first by determining the mean terrain elevation (i.e. height) along the object’s front and rear edges. In particular, the mean elevation along the front edge can now be obtained by simply averaging the $y$ coordinates (i.e. heights) of the calculated front contact points (e.g. front tires). The mean elevation at the object’s rear edge is similarly the average of the $y$ coordinates of the rear contact points. The object’s pitch at that animated instant is then given by the direction (i.e. vertical orientation) of the 3D vector constructed by joining the computed mean elevation points at the object’s front and rear edges.

The side roll of the object is finally calculated using an exactly similar procedure. In particular, the side roll is given by the direction of the vector constructed from the mean elevation positions along the object’s left and right edges. These individual computation steps are obviously not visible during an animation. At each animated instant, the object is drawn on the screen in its final, fully-oriented position. PathFinder thus computes a moving object’s accurate 3D configuration by prudently synthesizing inputs from the defined 2D motion path trajectory and the terrain model on which the simulated operations are animated.

6. Future Work

The variable speed motion of simulation objects traversing motion trajectories during the performance of communicated activity instances is purely based on kinetics. In particular, no physical constraints (e.g. mass of an object, its locomotive power, grades of the terrain, gravity) are considered in the computation that describes the simulation object motion. Any depicted 3D process (e.g. a loaded truck continuously accelerating uphill a
steep haul road) is a faithful representation of the information communicated by an underlying DES model regardless of whether that process (accelerating when traveling uphill loaded) can be accomplished in real life. Methods that can provide such feedback on physically-impossible simulated processes during visualization can, however, be of significant help within a framework intended to validate modeled (and animated) processes. Future work can explore techniques of designing such methods by incorporating dynamic physical variables in computing the motions of simulation objects.

Such a dynamics based approach can also lead to improved computation techniques of enforcing terrain-following in the moving simulation objects. In particular, the terrain-following algorithm adopted in the presented work computes an object’s orientations (pitch and roll) based purely on geometric computations. This approach presents good visual accuracy during visualization. Nevertheless, in general, there is no guarantee that an object’s contact points (e.g. tires) all touch the terrain surface at every instant. This is normally unnoticeable during visualization due to the object being in continuous motion. However, in the cases of objects that have large, continuous surface footprints (e.g. backhoe crawlers) instead of discrete, finite contact points such as tires, the adopted geometry-based terrain-following technique can often produce undesirable visual artifacts. Future work can explore the use of collision detection techniques in a dynamics-based framework to design exact methods of computing object-surface contact.

Finally, the time-based velocity profile shape scaling techniques designed in this work assume the profiles to be composed of piecewise-linear segments. This guides the procedure used in scaling the original defined profiles to accommodate communicated activity instance durations. However, typical velocity profiles generally exhibit curvature, particularly in the acceleration and deceleration phases of a moving object. Future work can explore techniques of defining such curved velocity profiles and design methods for heuristically scaling them to accommodate different motion completion times.
7. Summary and Conclusions

The presented research extends the state-of-the-art of scientific 3D visualization of DES modeled processes. The work puts in place the techniques that engineers can use to 1) define and manipulate curved, arbitrarily-shaped trajectories to represent accurate 3D motion paths of virtual simulation objects, 2) compute the precise three-dimensional spatial configuration of virtual simulation objects when they travel on defined motion paths, and 3) instruct virtual simulation objects to follow any arbitrarily-shaped velocity profiles while adhering to fixed motion completion times when traversing along any defined motion path trajectories.

In order to address the problem of describing the accurate, variable-speed motion of simulation objects on realistically shaped motion trajectories, the design/adoPTION and integration of three key technologies is required:

1. A mathematical representation for defining arbitrarily complex curves that can, by manipulating only high-level interaction parameters, be locally and/or globally edited to describe a realistic motion trajectory
2. A geometric basis to guide the computation of a simulation object’s correct 3D orientations as it travels on an arbitrarily uneven virtual terrain surface
3. A computation scheme that externally generates variable velocity profiles for moving simulation objects without wresting their temporo-spatial control from the underlying DES models that author animations

Kochanek-Bartels TCB splines can produce a very general class of cubic interpolating piecewise curves of any length whose shape can be locally or globally edited seamlessly by manipulating a set of high-level numerical parameters. Although such splines can describe a trajectory of any arbitrary 3D shape, the resulting geometric representation of the path precludes the accurate computation of the spatial configuration of objects that traverse that path. Instead, 3D motion path trajectories defined as TCB spline curves parameterized in 2D provide the necessary geometric basis to compute a mobile simulation object’s correct 3D configuration as it travels. This is achieved by superimposing the defined 2D motion trajectories onto the 3D terrain models being used
in the visualization and enforcing geometric terrain-following on the simulation objects as they traverse along the defined paths.

Discrete-event simulation models, by their very nature, are unconcerned about the rate at which activities in a model are performed. They only enforce the time instants at which activity instances start and end. Since the temporo-spatial control of virtual simulation objects cannot be wrested away from DES models, any information that defines an object’s variable velocity profile must originate within an underlying DES model. A computation scheme that allows DES models to define the general shapes of relevant velocity profiles and then heuristically scales those profiles to accommodate communicated activity instance durations performs well in a framework for animating DES models.

Such a technique not only allows simulation objects to be moved with any arbitrarily shaped velocity profiles, but also ensures that their temporo-spatial control rests entirely with the underlying DES models. The visually accurate animation results we obtain prove that this is not only possible, but also very effective in convincingly presenting modeled operations in dynamic 3D virtual worlds.

8. References


Chapter 8
Efficient Interference Detection in Automated 3D Construction Process Visualizations

1. Introduction
The ability to see a 3D animation of construction processes that have been simulated can be of significant help in verifying and validating the discrete-event process models. In addition, visualization of such models can provide valuable insights into subtleties of the simulated construction operations that are otherwise non-quantifiable and presentable. The necessity to effectively communicate modeled construction processes and the resultant evolving constructed facilities is the motivation behind ongoing visualization research efforts at Virginia Tech. These efforts focus on designing automatic, process simulation-driven methods to visualize construction processes and the resulting products in dynamic 3D virtual worlds.

The tangible outcome of this ongoing research is the VITASCOPE visualization system. VITASCOPE is an acronym for VIualizaTion of Simulated Construction OPERations. VITASCOPE is a user-extensible 3D animation language designed specifically for visualizing modeled construction operations in smooth, continuous, dynamic 3D virtual worlds (see Chapter 2 for details). A limited subset of the VITASCOPE language and the corresponding prototype implementation were referred to as the Dynamic Construction Visualizer (DCV) in some prior publications (Kamat and Martinez 2001, 2002).

1.1 Main Contribution
In this paper, we describe an extension to the VITASCOPE visualization system. We present a tool, C-COLLIDE that engineers can use to identify and report any and all undesirable conflicts that can occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources in dynamic 3D construction process visualizations.
Common types of clashes that can occur on real construction sites and that C-COLLIDE can identify beforehand in process visualizations include 1) intersection among physical in-place components (i.e. design interferences), 2) intersection among in-place components, components in transit, and/or pieces of moving equipment during construction (i.e. constructability interferences), 3) craft interferences and accidents e.g. collision between two pieces of equipment operating in the same area, and 4) space intrusions e.g. any resource (worker or equipment) encroaching arbitrarily shaped hazard or protected areas of the jobsite.

We capitalize on advanced documented algorithms for efficient collision detection between arbitrarily moving 3D geometric objects to design mechanisms for interference detection, control, and response in 3D construction process visualizations. C-COLLIDE’s interference detection capabilities dynamically check each motion of VITASCOPE scene objects to determine if any pairs of scene objects interfere undesirably. This provides engineers with a lucid understanding of all object motions and potential interferences in any area of activity on a simulated construction job site.

Previous works in construction have applied collision detection algorithms in various static and dynamic graphical visualization contexts. Applications such as Interference Manager (Bentley 1999) and the Interference Detection Language (IDL) (Dias and Gamito 2001) use collision detection algorithms to identify geometric clashes between permanent facility components and automatically verify the design of a proposed facility. Akinci et al. (2002) have used interference detection algorithms to check for intersections among rectangular prisms (representing spaces required by construction activities) to perform time-space conflict analyses in 4D CAD visualizations. Finally, tools such as Dynamic Animator (Bentley 2000) utilize collision detection algorithms to check for geometric interferences between objects during playback of manually created motion sequences.

Neither of the above applications, however, is real-time. In other words, in all the abovementioned applications, there is no relation between elapsed visualization time and
elapsed wall clock time. Design verification operates on static CAD data and obviously has no notion of time associated with the computation. Although time-space conflict analysis identifies spatio-temporal clashes among construction spaces, there is no relation between simulation time (i.e. activity durations) and elapsed wall clock time i.e. the computation is performed in batch processing mode. In Dynamic Animator, the elapsed visualization time is related to the number of screen updates rather than wall clock time. Thus, in all these applications, efficiency in interference detection computations, although desirable, is not of the essence.

VITASCOPE animations are real-time because elapsed visualization time is always related to the elapsed wall clock time by a distinct viewing ratio. This viewing ratio is strictly maintained regardless of the number of screen updates between subsequent simulation time instants. Thus, to depict smoothly moving scene objects and to avoid jerkiness in the presented graphics, VITASCOPE must always update the screen at an acceptable refresh rate (ideally >= 24 frames per second). The impact of any involved collision detection computations on this frame rate must be minimal.

Efficiency in the adopted collision detection algorithms is therefore of paramount importance in C-COLLIDE. In addition, this efficiency cannot be traded off against accuracy. C-COLLIDE’s main contributions are:

- Highly efficient and accurate methods for automated construction process level interference detection and conflict analyses at interactive rates.
- Integrated framework for performing combined design level, activity level, and process level spatio-temporal interference analyses.

2. Challenges

Scientific construction process visualization involves the faithful representation of real-world construction processes and the resulting evolving products in smooth, dynamic, continuous 3D virtual worlds. The types of undesired collisions and interferences that can occur in such visualizations encompass the entire range of undesired interferences that can occur on real construction sites. In 3D visualizations that describe process-level
construction details, the goal of collision detection and interference analysis is thus, in
general, the identification and reporting of any and all undesirable conflicts that might
occur among static, dynamic, and abstract construction resources on any part of the
virtual jobsite.

Collision detection and interference analysis in virtual environments by definition
requires intensive computation. It is traditionally considered a bottleneck in smooth,
continuous, animated 3D virtual environments (Lin and Gottschalk 1998). The
complexity of the required computation is directly proportional to the complexity of the
3D graphical scene database (i.e. number and types of objects contained). Virtual worlds
describing dynamic construction processes epitomize the description of a complex 3D
scene. Construction is performed in an arbitrarily complex dynamic environment where
equipment and materials operate together, are attached to one another or taken apart, and
the landscape itself changes shape. These actions in turn cause a construction product
facility (e.g. building, bridge, tunnel etc.) to steadily evolve with the passage of time. In
such circumstances, the dynamic 3D graphical database that describes construction
processes and the resulting products in a virtual world can become extremely large and
complex.

A virtual world representing a physical world environment such as a construction site
generally consists of N moving objects and M stationary objects, where both N and M
can be arbitrarily large numbers. Each of the objects can have an arbitrary shape that is
often dynamic and cannot be predetermined. The detection of object to object collisions
in such environments is a computationally intensive task because each of the N moving
objects can theoretically collide with the other moving objects, as well as the stationary
ones. As such, the collision detection detection test has to be performed at each time step (i.e.
frame) in the visualization – an operation whose computation cost increases sharply with
scene complexity. Efficiency in contact determination algorithms is thus paramount in
such environments.
This requirement for extreme efficiency is distinct from requirements of many other applications of collision detection algorithms (e.g. design verification from static CAD data) where the computation time is not at as much premium as is in dynamic animation. In dynamic animation, collision tests must be performed on an arbitrary number of objects at each frame in a fraction of time such that the involved computation has minimal adverse influence on the animation’s frame rate. In addition and notwithstanding the speed requirement, performed collision tests must be spatially accurate and must often determine the exact point(s) of contact between interfering pairs of scene objects. This can be helpful in contemplating remedial actions to avoid the occurring interferences.

In virtual environments representing physical world processes, interference detection computation can almost always be optimized using contextual rules and assumptions (Lin and Gottschalk 1998). For example, the bucket, stick, boom, cabin, and the tracks of a single excavator are represented as a hierarchy of distinct 3D objects that are each capable of being manipulated separately inside the virtual world. However, it would be a waste of computational effort if collisions between pairs of all components on the same excavator (e.g. boom and stick, cabin and tracks etc.) are computed at each frame.

Computation effort would similarly be wasted if CAD objects representing resources guaranteed to be non-interacting are repeatedly tested for intersections. Examples of such resources include excavators operating at different distant loading areas and trades operating on different levels of a building that are a few floors apart. Engineers visualizing construction processes must thus be able to precisely specify and control the pairs of scene objects that must be tested for possible interferences at each frame. This is essential if the requirement of extreme efficiency in collision detection computations is to be achieved.
2.1 Desiderata

In order to satisfy these required criteria (functionality, speed, accuracy, and flexibility) imposed by dynamic 3D virtual construction worlds, mechanisms designed for collision detection and interference analysis must be:

- **General** – A dynamic virtual construction site can consist of several objects of arbitrary shapes, forms, and sizes (from the collision detection algorithm’s viewpoint). The algorithms adopted to perform interference analysis in dynamic construction process visualizations must therefore operate on 3D CAD models of arbitrary shapes, sizes, and forms. In addition, the algorithms must not make any assumptions about object motions, velocities, and accelerations.

- **Efficient** – On a virtual construction site with $N$ mobile and $M$ stationary entities, monitoring $N+M$ objects and checking $\binom{N}{2} + N \cdot M$ pairs of objects for intersection at every time step can become time consuming and inefficient as $N$ and $M$ get large (typical of virtual construction sites). Visualizations describing modeled construction operations must be smooth and continuous and not jerky (Law and Kelton 2000). To achieve interactive rates of interference analysis, then, the total number of pair-wise intersection tests must be reduced before performing exact collision tests on the object pairs that are in the close vicinity of each other. Contextual rules and assumptions are one possible technique. In addition, literature describes several algorithmic optimizations to deal with such large datasets of possibly colliding objects (Cohen et al. 1995).

- **Accurate** – When an undesirable collision or interference occurs between two virtual entities involved in construction, engineers are typically interested in the exact point(s) and/or location(s) of contact. This is especially true if the involved objects are occluded by other objects or if the interference is subtle. Feedback on the exact details of the occurring interference can further be of help to engineers in devising strategies to avoid interference during actual construction. The adopted algorithms must therefore report accurate details of any detected collisions and interferences without overloading the computation pipeline.
• **Dynamic** – A 3D virtual construction site is represented as a scene graph inside the computer (Kamat and Martinez 2002). Since construction operations are performed in an inherently dynamic environment, scene objects (i.e. resources) move, and may appear or disappear at any time during visualization (when they travel to and out of engineers’ visual area of interest). The data structures for collision detection and interference analysis must be able to handle this dynamic behavior. In other words, it must be possible to easily insert and remove scene objects from the collision detection engine’s list of monitored scene objects.

• **Interactive** – Engineers must be able to selectively specify the scene objects that the collision detection engine must monitor and the nature of the feedback that must be generated when interferences are detected.

### 3. Collision Detection Algorithms

Collision detection between 3D geometric models (also known as contact determination or interference detection) has been extensively studied in fields such as geometric modeling, computer graphics, robotics, and computational geometry. Since collision detection is employed for a wide variety of applications, different domain specialized algorithms and methods have been developed for the purpose (Lin and Gottschalk 1998).

#### 3.1 Problem Domain

Determining interferences among multiple complex moving objects in 3D virtual environments has become a very popular research topic in computer graphics in the relatively recent past (Ponamgi et al. 1997). The primary reason for this research emphasis is the need for extreme efficiency in interference detection in such dynamic complex virtual environments. Our discussion of collision detection algorithms is only limited to this latter context where much of the research addresses the familiar N-body problem that signifies the worst case scenario of N unconstrained 3D objects moving arbitrarily in a scene containing possibly M other stationary objects. The non-emergence of a universally popular efficient algorithm is the primary motivator for continued ongoing research in detecting collisions between objects in such dynamic virtual
environments. The collision detection problem in a dynamic 3D virtual world consisting of many moving and stationary objects can be generally separated into three areas:

- Detecting the occurrence of a collision between a pair of scene objects
- Determining the point of contact to some degree of accuracy
- Deciding the nature of the response to the detected collision

The first two steps involve geometric computation and are generally the focus of core research in computer graphics. The last step, i.e. collision response is usually application dependent and often requires a dynamic model.

### 3.2 Relevant Taxonomy

The algorithms to accomplish the first two steps of detecting collisions between 3D geometric objects can be broadly classified into the following two categories based upon their computation scheme:

- **Two-phase methods** – These algorithms divide the contact determination stage into two distinct phases each time a check is desired. The first broad phase quickly culls or eliminates from further computation pairs of scene objects that cannot possibly collide. Approximate geometric techniques, specific rules and context of the pertinent visualization, and/or algorithmic optimization techniques are used for this purpose. Pairs of scene objects that pass the first broad phase are deemed to be possibly colliding. These object pairs are then subjected to accurate detailed collision detection analysis using suitable low-level geometric techniques. This second narrow phase generally requires much more geometric computation.

- **Single-phase methods** – Algorithms in this class do not utilize the first broad object elimination phase. Instead, they directly subject each existing scene object pair to detailed, narrow-phase geometric tests to determine any existing contact. By eliminating the broad phase altogether, such algorithms often prove efficient in cases where most scene objects are generally deemed to be possibly colliding and must be checked in detail anyways.
In addition to computation schemes, collision detection algorithms may also be classified based upon several different criteria such as the input geometry type (e.g. polygonal, non-polygonal, structured, unstructured etc.), types of queries supported (e.g. contact determination, exact separation distance, depth of penetration etc.), and types of consumer visualization environments (e.g. scientific visualizations, games, interactive simulators etc). We refer the reader to Lin and Gottschalk (1998) for a comprehensive and informative survey on these taxonomies, other discussion, and references to several works on core collision detection research.

3.3 Algorithmic Optimizations

All algorithms designed for real-time contact determination among several moving objects in virtual environments generally exploit spatial and temporal coherence among scene objects in their computation. Moving objects in 3D worlds generally tend to occupy the same region of space during subsequent time instants. In other words, the amount by which most objects move each frame is generally less compared to the dimensions of the objects themselves. Several algorithms exploit this characteristic that is common of virtual environments representing physical world processes. Some algorithms require bounds on the motion of the objects (e.g. object velocities or accelerations). Other algorithms such as the ones based on interval arithmetic need a closed form expression of the motion as a function of time. Some algorithms demand no information on the motion but need only the placements of the objects at successive time steps.

4. Technical Approach

We have adopted a two-phase, hierarchical multi-level approach presented in Hudson et al. (1997) to address the problem of collision detection and interference analysis in dynamic 3D construction process visualizations. From a collision detection algorithm’s viewpoint, a typical virtual construction site is composed of N arbitrarily moving and M stationary objects. Each of the N objects can possibly collide with the other N-1 objects as well as with the M stationary objects. However, since activities on a construction site are typically spread both laterally and vertically, not all object pairs can be deemed to be possibly interacting at any time instance. The two-phase approach described in the
previous section is thus more suitable and efficient for our purposes because during each intersection test (one per frame), we can quickly eliminate pairs of scene objects that are not within interacting distance of each other. This reduces the computation load manifold by limiting detailed pair-wise intersection tests to only those object pairs that are very near each other.

Figure 1: Interference Detection Algorithm Architecture

Figure 1 presents the architecture of our adopted interference detection algorithm. After each dynamic scene update, a quick conservative approximate test based on loosely fitting, imaginary bounding volumes known as axis-aligned bounding boxes (AABBs) first finds potentially colliding pairs of objects among the set of existent scene entities. The algorithm adopts a procedure known as an N-body sweep and prune in computer graphics literature (Cohen et al. 1995). After marking pairs of potentially colliding objects, an exact two-level test computes whether the two objects in each marked pair actually collide. In this stage, the algorithm constructs another kind of imaginary
bounding volumes called oriented bounding boxes (OBBs) around each scene object and its geometric primitives. These bounding volumes are then organized into a hierarchical structure called OBBTrees (Gottschalk et al. 1996). Overlap tests on the constructed OBBTrees followed by exact geometric primitive intersection tests are then performed to confirm collisions and to determine the exact point(s) of contact. The following subsections clarify all of the above terminology and elaborate on each of the geometric computation phases of the algorithm. Discussion of the analysis and the response to detected interferences is deferred to the following section.

4.1 Pruning Scene Object Pairs

The sweep and prune part of the algorithm reduces the number of computationally intensive exact pair-wise collision tests by eliminating from the computation pipeline pairs of scene objects that are far apart. This is accomplished by constructing imaginary, loosely fitting 3D bounding volumes to surround (i.e. enclose) each scene object and then determining which pairs of those volumes overlap. Only pairs of objects whose loose-fitting bounding volumes intersect (indicating that the objects are in close vicinity and possibly colliding) are passed on to the next stage of exact contact determination computation.

In the sweep and prune phase, the algorithm we adopt calculates easy-to-compute rectangular bounding volumes called axis-aligned bounding boxes (i.e. AABBs) to enclose each scene object. AABBs are imaginary rectangular prisms whose sides are aligned with the three major coordinate axes irrespective of the enclosed object’s orientation in global space. The orientation of the enclosed object affects the size and the amount of free space inside the enclosing bounding box. As a result, AABBs describe loosely fitting bounding volumes for all but the most ideal enclosed object orientation. Figure 2 illustrates this graphically by projecting in two dimensions.
The constructed bounding boxes are analyzed in 3D space to determine overlapping pairs. A dimension reduction approach is used for the purpose (Cohen et al. 1995). The sweep and prune algorithm begins by projecting each three-dimensional bounding box onto the three coordinate axes (X, Y, and Z). Since the bounding boxes are axis aligned regardless of the enclosed object orientation, their projection onto the coordinate axes describes finite linear intervals. Figure 3 presents this graphically for the X-axis. By determining overlaps among the projected intervals of bounding box pairs, it is possible to deduce whether or not the boxes overlap in 3D space. As is apparent from figure 3, this follows from the simple intuitive result that a pair of AABBs can intersect in 3D if and only if their projected intervals overlap in all three dimensions.

At each time instant (i.e. frame) in a visualization, the sweep and prune algorithm reconstructs the AABBs for each scene object and flags pairs of scene objects whose AABBs intersect. The remaining object pairs are deemed as not colliding and quickly eliminated from the computation pipeline. Only pairs of scene objects that pass the sweep and prune test are passed on to the next detailed collision detection phase that we now describe in the following subsection.

4.2 Exact Pair-wise Collision Tests

The next phase of the algorithm detects exact pair-wise collisions among pairs of scene objects that are flagged as potentially colliding by the preceding approximate test. This is accomplished by building hierarchical representations of another type of imaginary 3D
bounding volumes called oriented bounding boxes (i.e. OBBs) to enclose objects. OBBs are imaginary rectangular prisms aligned at an arbitrary orientation in 3D space. Unlike AABBs that are always aligned with the co-ordinate axes, OBBs can be oriented in any direction such that the resulting volume encloses the bounded object as tightly as possible. Since OBBs are constructed taking an object’s orientation into account, the size and the amount of free space (i.e. the tightness) inside the enclosing bounding box always remain constant. As a result, OBBs describe tight fitting bounding volumes for all enclosed object orientations. Figure 4 illustrates this graphically by projecting a set of OBBs in two dimensions.

Figure 3: Bounding Box Projections
4.2.1 Construction of Bounding Volume Hierarchy

The exact collision test algorithm computes a tree of OBBs (called OBBTree) for every object, with a box containing the entire object as the root and boxes containing only one or a very few primitives of the object as the leaves. The tree construction process, presented graphically in figure 5, has two components. First is the placement of a tight fitting OBB around each (sub)object (represented as a collection of polygons), and second is the grouping of nested OBBs into a tree hierarchy.

Figure 4: Oriented Bounding Boxes

Figure 5: Building the OBBTree
The algorithm approximates the collection of polygons in a (sub)object with an OBB of similar dimensions and orientation. The OBB computation procedure makes use of first and second order statistics summarizing the vertex coordinates. They are the mean and the covariance matrix respectively. The eigenvectors of a symmetric matrix, such as the covariance matrix, are mutually orthogonal. After normalizing them, they are used as a basis. The procedure finds the extremal vertices along each axis of this basis, and sizes the bounding box, oriented with the basis vectors, to enclose those extremal vertices. Since two of the three eigenvectors of the covariance matrix are the axes of maximum and of minimum variance, they tend to align the box with the geometry of a tube or a flat surface patch. The exact formulae and construction details of this procedure are described in detail in Gottschalk et al. (1996).

To construct the OBBTree for a scene object, the algorithm adopts a top-down recursive approach that partitions the primitives in each box into two sub-boxes, based on the location of their centers. The procedure begins with the group of all polygons (i.e. the OBB of the entire object), and recursively subdivides that OBB until all leaf nodes are indivisible. The subdivision rule that is adopted in this procedure splits the longest axis of a box with a plane orthogonal to one of its axes, partitioning the polygons according to which side of the plane their center point lies on. The subdivision coordinate along that axis is chosen to be that of the mean point of the vertices. If the longest axis cannot be subdivided, the second longest axis is chosen. Otherwise, the shortest axis is used. The process continues until the group of polygons cannot be further partitioned along any axis by this criterion (i.e. the group is considered indivisible).

4.2.2 Hierarchical Exact Overlap Test

To check for collision between a pair of objects, the algorithm descends their OBB hierarchies to find any leaf boxes which overlap, and then performs exact intersection tests between the triangles in the overlapping leaves. The root of an OBBTree encloses the entire object itself and its leaves contain one or more primitive constituents. The hierarchy thus defines an intra-object spatial partitioning. Successive levels of OBBs in two OBBTrees must only be tested for intersection if their respective parent OBBs
intersects. Figure 6 presents the pseudo code for traversing the constructed bounding volume hierarchies to detect collisions.

Starting at the root nodes of two given trees
1. Check for intersection between two parent nodes
2. If there is no intersection between two parents
3. Then stop and report “no collision”
4. Else check all children of one node against all children of the other node
5. If there is intersection between any children
6. Then If at leaf nodes
7. Then report “possible collision”
8. Else go to Step 4
9. Else stop and report “no collision”

Figure 6: Traversing Bounding Volume Hierarchies to Detect Collisions

The algorithm confirms whether two scene objects are disjoint or not by projecting their OBBs onto some axis (not necessarily a coordinate axis) in space. Akin to the projection of AABBs onto the coordinate axes, the axial projection of each box on an arbitrary axis forms an interval on that axis. If the intervals don't overlap, then the axis is called a “separating axis” for those boxes. The existence and identification of a separating axis confirms that the tested boxes are disjoint. If the intervals do overlap however, then the boxes may or may not be disjoint and further testing is required. Figure 7 graphically presents these scenarios.

The algorithm we adopt makes use of the separating axis theorem presented in Gottschalk et al. (1996) to check for overlaps between OBBTrees of scene objects. According to the theorem, two convex polytopes in 3-D (OBBs in our case) are disjoint if and only if there exists a separating axis orthogonal to a face of either polytope or orthogonal to an edge from each polytope. Each box has 3 unique face orientations, and 3 unique edge directions. This leads to 15 potential separating axes to test (3 faces from one box, 3 faces from the other box, and 9 pair-wise combinations of edges). If the polytopes are disjoint, then a separating axis exists, and one of the 15 axes mentioned above will be a
separating axis. If the boxes are overlapping, then clearly no separating axis exists. So, testing the 15 given axes is a sufficient test for determining overlap status of two OBBs.

![Image of OBBs and intervals](image)

**Figure 7: Searching for a Separating Axis**

If pairs of leaf OBBs are found to be intersecting during the OBBTree traversal, the algorithm performs deterministic geometric intersection tests between the primitive triangles in the overlapping leaf OBBs. This final computationally demanding test confirms beyond doubt whether the two scene objects actually intersect and if affirmative, also determines the exact point(s) of contact. This last test is performed only if pairs of scene objects pass the earlier tests that are performed in an order of increasing accuracy (and computation cost). By adopting this hierarchical, multi-level procedure, the algorithm is thus able to provide us with exact collision detection capabilities while maximizing computational efficiency.
We now describe our implementation of the adopted algorithms. The implementation is a powerful tool that provides engineers with comprehensive, accurate, and interactive interference detection capabilities in smooth, continuous, dynamic 3D construction process visualizations.

5. C-COLLIDE

Our implementation of efficient interference detection algorithms is a tool that provides engineers with comprehensive and accurate feedback on any and all undesired interferences that occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected areas) construction resources in dynamic 3D construction process visualizations. This tool, called C-COLLIDE, is implemented as an extension (add-on) to the VITASCOPE visualization system.

The C-COLLIDE add-on partly redefines as well as extends the VITASCOPE animation language. The add-on redefines several core VITASCOPE animation language statements. The redefined statements instruct the visualization engine to perform interference detection computations in addition to performing the regular core VITASCOPE computations that define and manipulate objects on virtual construction sites. In addition, the add-on defines several new language statements that provide engineers with precise control over 1) which pairs of scene objects are monitored and tested for potential collisions, and 2) the nature and semantics of the feedback that is generated when interferences among various scene objects are actually detected.

C-COLLIDE also defines special statements to describe arbitrarily shaped abstract construction resources (e.g. hazard or protected spaces) in process visualizations. Such resources need not necessarily have a physical representation (i.e. they might be invisible) but are nevertheless relevant in the context of interference analysis (e.g. hazard or protected space incursions). C-COLLIDE’s parametric text statements can be used together in interesting ways to detect, analyze, and report any occurring interferences in construction process visualizations.
Table 1 presents selected statements C-COLLIDE implements and briefly indicates their usage. In case of redefined (i.e. supplemented) statements that are part of the core VITASCOPE language, the last column indicates their original usage. Statements containing a N/A entry in the last column are original C-COLLIDE statements that have not been previously defined in VITASCOPE. In cases of statements that are redefined, column two indicates the functionality that C-COLLIDE appends to the original statements. For original C-COLLIDE statements, column two describes the complete statement functionality. The detailed usage of all statements is presented in appendix L.

5.1 Object Definition and Initialization

C-COLLIDE redefines several core VITASCOPE statements in order to perform the supplemental computation necessary to initialize the interference detection routines as objects are instantiated and introduced in scenes. The supplemental computation involves converting scene object representations into formats that the interference detection routines can read and operate on. C-COLLIDE’s algorithms require each instantiated scene object (represented by a 3D CAD model) to be an arbitrarily large set of disconnected primitive triangles. These triangles must together describe all the surface and internal features of a scene object.

CAD models used by VITASCOPE to describe the geometric description of simulation objects can be imported from a wide variety of sources in different file formats. These CAD models can be internally described as a set of disconnected triangles but can also be defined in a wide variety of other CAD model representations such as parametric surfaces, constructive solid geometry, implicit surfaces, and polygon sets with topological information. In order to detect interferences between simulation objects then, C-COLLIDE must first convert all input CAD models into the triangulated format that it can read and operate on.
<table>
<thead>
<tr>
<th>Statement</th>
<th>C-Collide Appended/Defined Usage</th>
<th>Original Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS [ClassName] [CADFileName];</td>
<td>Converts the input CAD file into a format interpretable by the interference detection algorithm.</td>
<td>Associates a class of simulation entities with their geometric description contained in a CAD file.</td>
</tr>
<tr>
<td>CREATE [ObjName] [ClassName];</td>
<td>Constructs, associates, and stores the OBBTrees of the created simulation objects.</td>
<td>Creates specific simulation objects by instantiating predefined classes.</td>
</tr>
<tr>
<td>PLACE [ObjName] [AT/ON] [Location];</td>
<td>Notifies the collision detection engine about the presence of new scene objects. This includes any objects attached to the object being placed.</td>
<td>Places simulation objects at particular locations or at the beginning of resource movement paths.</td>
</tr>
<tr>
<td>ATTACH [ChildObjName] [ParentObjName] [AttachPoint];</td>
<td>Notifies the collision detection engine about the presence of new scene objects if the parent objects are already in the scene.</td>
<td>Attaches objects to one another at a specified pivot point</td>
</tr>
<tr>
<td>ABSTRACTOBJECT [ObjName] [ClassName];</td>
<td>Creates an arbitrarily shaped abstract (i.e. invisible) scene object.</td>
<td>N/A</td>
</tr>
<tr>
<td>ACTIVATEOBJECT [ObjName];</td>
<td>Turns on collision tests for all object pairs involving this object.</td>
<td>N/A</td>
</tr>
<tr>
<td>DEACTIVATEOBJECT [ObjName];</td>
<td>Turns off collision tests for all object pairs involving this object.</td>
<td>N/A</td>
</tr>
<tr>
<td>ACTIVATEPAIR [ObjName1] [ObjName2];</td>
<td>Turns on collision tests for a specific pair of objects. Both objects must be active.</td>
<td>N/A</td>
</tr>
<tr>
<td>DEACTIVATEPAIR [ObjName1] [ObjName2];</td>
<td>Turns off collision tests for a specific pair of objects.</td>
<td>N/A</td>
</tr>
<tr>
<td>RESPONSEMODE [INTERACTIVE/SILENT] [LogFileName];</td>
<td>Indicates whether C-COLLIDE must interactively report collisions or record them silently to a disk log file.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

C-COLLIDE provides an internal CAD model converter that triangulates (if necessary) input CAD files into a format suitable for the adopted interference detection algorithm.
This conversion is performed each time a CLASS statement associates a group of simulation entities with their geometric description by supplying a 3D CAD file. While the core VITASCOPE functionality of instantiating, manipulating, and displaying objects in a scene is accomplished using the original input CAD model, C-COLLIDE uses the converted model representation in all its computations.

Figure 8 juxtaposes an original CAD representation of a concrete truck against its converted triangulated representation. VITASCOPE manipulates and displays the original model during visualization. However, C-COLLIDE computations that test for interferences involving any virtual concrete trucks are performed using the triangulated representation. This triangulated representation is obviously not displayed in the virtual world during visualization (i.e. the original model is displayed) and is meant for internal computation only.

![Figure 8: Original and Triangulated CAD Model Representations](image)

The other core statements (CREATE, PLACE, and ATTACH) that C-COLLIDE redefines in effect build, monitor, and maintain an additional, parallel, invisible scene database that is a copy (albeit in a different format) of the scene database that VITASCOPE constructs and manages to display virtual construction processes. While VITASCOPE uses the original scene database for scene manipulation and rendering, C-COLLIDE performs the interference detection computations using its internal scene database copy. Obviously, C-COLLIDE’S database must be updated each time VITASCOPE manipulates the scene.
5.2 Interference Detection Control

C-COLLIDE’s interference detection control statements allow engineers to explicitly and precisely specify contextual rules and assumptions to the collision detection engine. As discussed earlier, this capability can be used prudently to make the interference detection calculations more efficient by eliminating from the computation pipeline pairs of scene objects that are not likely to interact. Examples of such objects include pieces of equipment working on different areas of a jobsite or trades working on different floors of a building under construction.

C-COLLIDE provides two components of statements for interference detection control and contextual optimization. These statements allow engineers to specify which objects should be monitored and tested for collisions. The first per-object component defines statements (ACTIVATEOBJECT, DEACTIVATEOBJECT) that operate on a single scene object and either add or remove from the computation pipeline all possible pairs of scene objects containing the object operated upon. The second pair-wise component defines statements (ACTIVATEPAIR, DEACTIVATEPAIR) that operate on a specific pair of scene objects either turning interference detection among them on or off.

All created C-COLLIDE objects are active by default. In addition, all pairs of scene objects that include the newly created object are also activated initially. C-COLLIDE manages these two control components separately. Thus, for a pair of objects to be tested for intersection or collision, not only must the pair be active, but each of the two objects must be active as well. The following example elucidates the usage of these control statements.

Figure 9 presents an animation snapshot of a bridge construction site. On this job, fresh concrete manufactured at a batch plant (visible on the far left) on the shore was placed in hollow steel jackets to cast the piers of the bridge. Concrete was delivered to the workface at the piers using barges to transport concrete trucks. Concrete from each arriving truck was pumped into a hopper built on the pier’s working platform. The hopper fed a tremie pipe that was lowered into each steel jacket. As the depth of placed concrete
rose, a crane mounted on a floating platform withdrew sections of the tremie pipe from
the jacket and placed them on racks after being cut. This procedure continued until the
entire depth of the jacket was concreted.

Figure 9: Animation Snapshot of a Bridge Construction Jobsite

Figure 10 presents a portion of the animation trace file that uses C-COLLIDE’s control
statements to specify simple contextual rules and assumptions to the collision detection
engine. Of particular interest is the scene object JobSite that describes the terrain
(including the water surface) and the landscape of the jobsite. Most scene objects (i.e.
barges, trucks, floating platforms, batch plant etc.) are obviously in contact with the
surface at all times. Other scene objects such as the crane and the concrete pump operate
on the floating platform and are not anticipated to be in contact with the surface for the
duration of the animated processes.
Since both the terrain and the other scene objects are represented by CAD models, the perennial contact between them is geometric interference from the collision detection algorithm’s viewpoint. From the engineer’s perspective, however, such interaction (e.g. barges hulls touching the water, truck tires touching the terrain) is obviously permissible. The fifth statement thus turns off collision among all pairs of scene objects involving the JobSite object. In addition, the crane mounted on the floating platform is not anticipated to be in close proximity to the batch plant during the duration of the animated operations. The final statement conveys this assumption to the collision detection engine. Such contextual assumptions dramatically reduce the number of scene object pairs that C-COLLIDE must monitor for possible collisions increasing the algorithm’s computational efficiency manifold.

```plaintext
LOADADDON CCollide;

CLASS Terrain Terrain.wrl;
CREATE JobSite Terrain;
PLACE JobSite AT (0,0,0);

DEACTIVATEOBJECT JobSite;

CLASS BatchPlant BatchPlant.wrl;
CREATE BatchPlant1 BatchPlant;
PLACE BatchPlant1 AT '(-152,9,-740)';

CLASS Crane Crane.wrl;
CREATE Crane1 Crane;
PLACE Crane1 AT '(-5,2.9,-15)';

DEACTIVATEPAIR Crane1 BatchPlant1;

//... More contextual rules/assumptions
```

**Figure 10: Specification of Contextual Rules and Assumptions**

### 5.3 Collision Feedback and Response

C-COLLIDE tests for possible interference between all active pairs of scene objects at every time instant (i.e. frame). C-COLLIDE defines a statement (RESPONSEMODE) that allows engineers to specify the nature of the feedback that must be generated when collisions between active scene object pairs are detected. Two response modes –
interactive and silent – are currently defined. In interactive mode, C-COLLIDE pauses a running animation and outputs details about each detected interference. C-COLLIDE then waits for the user to specify whether 1) the interference detected should be ignored, 2) the animation should be aborted, or 3) the scene object pair should be deactivated from further computations.

In silent mode, C-COLLIDE does not interrupt a running animation even if interferences are detected. Instead, details about all collisions that occur during an animation run are time stamped and written to a formatted disk file. Engineers can later analyze the outputted interference log for any collisions that might have occurred during visualization. C-COLLIDE’s silent collision response mode is particularly useful for detecting interferences in long animations that span hours or even days and for detecting collisions that occur rarely (i.e. rare simulation events). In both cases, interactive detection would require hours of visualization even when using a high viewing ratio (ratio of simulation time to wall clock time).

6. Advantages of Add-On Approach

Collision detection and interference analysis is considered to be the most challenging problem in visual simulations. The collision detection algorithms adopted by any visual simulation application usually influence the core components involved in the computation loop. This is evident from the fact that C-COLLIDE redefines (i.e. augments) several core VITASCOPE statements in order to implement collision detection capabilities. VITASCOPE’s ability to allow such core computation components to be implemented as add-ons to the main visualization engine has several distinct advantages.

In C-COLLIDE’s case, the adopted add-on approach allows us to modify any portion of the collision detection routines without having to modify the implementation of VITASCOPE’s main visualization engine. In addition, this approach provides the flexibility that can allow any researcher to implement, if necessary, an entirely different collision detection algorithm for use in VITASCOPE visualizations. This is of significant importance and relevance since fundamental work on efficient collision detection
algorithms is still an active research area in computer graphics. By implementing its collision detection algorithms as an add-on, C-COLLIDE provides engineers the option of easily replacing VITASCOPE’s interference detection capabilities with any other algorithms of their choice.

7. Conclusion and Future Work

The presented work capitalizes on advanced 3D geometric collision detection algorithms to design efficient mechanisms for interference detection, control, and response in dynamic 3D construction process visualizations. The mechanisms defined by C-COLLIDE allow engineers to identify any and all undesirable conflicts that can occur among static, dynamic, and abstract construction resources in construction process visualizations.

C-COLLIDE’s interference detection algorithms only detect the occurrence of “hard” interferences between physical scene components and report the exact point(s) of contact if queried. C-COLLIDE’s algorithms provide no routines to compute and report distances between two scene objects. Such routines can be useful in construction process visualizations as they can allow the detection of “soft” interferences such as minimum clearance violations between any pair of stationary or mobile scene objects. C-COLLIDE does provide a statement (ABSTRACTOBJECT), however, that can be utilized as a workaround to define an invisible, protected envelope around any scene object.

Since research on efficient collision detection algorithms is ongoing in computer graphics, the design of any new algorithms or techniques presents an opportunity to improve upon or replace the collision detection algorithms adopted in C-COLLIDE. Of course, any such improved algorithms must be able to conform to VITASCOPE’s requirements and its add-on interface. From a VITASCOPE user’s perspective, C-COLLIDE provides construction engineer’s with a comprehensive test bed for several studies – such as process or craft level spatial conflict analyses or space usage analyses – that require the capability to detect interferences among virtual resources in dynamic construction process visualizations.
8. References


Chapter 9
Validating Complex Construction Simulation Models Using 3D Visualization

1. Introduction

Discrete-event simulation (DES) is a modeling technique that has been used to analyze and design many construction operations. DES is particularly beneficial for modeling complex dynamic systems that are intractable to other modeling approaches. The state-of-the-art construction simulation systems allow the modeling of complex construction operations in great detail and with utmost flexibility. Notwithstanding, there has been limited use of DES in planning and designing construction operations (Halpin and Martinez 1999, Tucker et al. 1998).

Construction simulation tools typically provide results in the form of numerical or statistical data. However, they do not illustrate the modeled operations graphically in 3D. This poses significant difficulty in communicating the results of simulation models, especially to persons who are not trained in simulation but are nevertheless involved in making decisions. The resulting “Black-Box Effect” is a major impediment in verifying and validating simulation models. Decision makers often do not have the means, the training and/or the time to verify and validate simulation models based solely on the numerical output of simulation models and are thus always skeptical about simulation analyses and have little confidence in the results (Ioannou and Martinez 1996). This lack of credibility is a major deterrent hindering the widespread use of simulation as an operations planning tool in construction.

This paper investigates the efficacy of 3D visualization in verifying and validating discrete-event construction simulation models. The paper illustrates the use of DES in the design of a complex dynamic earthwork operation whose control logic was then verified
and validated using 3D animation. The simulation model was created using STROBOSCOPE and animated using VITASCOPE.

2. Verification and Validation

Discrete-event modeling is an inherently complex activity that is both a science and an art. The modeling of a construction operation requires the description, in the language of the simulation modeling system, of mental plans that are often complex and elaborate. Differences between the mental plan and the operation actually modeled in a first attempt are ubiquitous. Verification is the process by which the model creator looks at what has been actually modeled, compares it to what was intended, and updates the model to accurately reflect the intention.

The developer of the computer simulation model, however, may have misconceptions about how the actual operation will take place in the field. Thus, a model may not be an accurate representation of reality despite proper verification by its developer. Such errors cannot be discovered by verification because the model indeed reflects what the model creator intended. The aim of Validation therefore is to determine whether simulation models accurately represent the real-world system under study. This is typically carried out by consulting people who are intimately familiar with the operations of the actual system, but who are not necessarily proficient in simulation. Figure 1 schematically presents the processes of verification and validation that lead to accreditation of the model and the use of its results in implementing the actual operation at the jobsite.

Simulation models are termed as Credible when the models and their results are accepted as being valid, and are used as an aid in making decisions (Law and Kelton 2000). In the case of both Verification and Validation, the inner workings of a model and its output need to be communicated to others for discussion and input, in a way that is both comprehensive and comprehensible (Oloufa and Ikeda 1997).
Visualizing simulated operations can be an effective means of achieving this (Law and Kelton 2000, Robinson 1997, Henriksen 1998). It is a generally accepted fact that
visually presented information is understood and grasped more easily than any other form of communication. The need to visually communicate simulated operations is more relevant in the context of construction because construction operations analysts (e.g., superintendents) typically do not have the necessary training in simulation to allow them to validate simulation results based on numerical analysis.

Accurate 3D visualization can substantially help to communicate intricacies of construction simulation models. It can provide valuable insight into details of construction operations that are otherwise non-quantifiable and presentable. It has the potential to enable the extraction of knowledgeable information from simulations. Visualization can be of help in verification as well as in validation of simulation models. Volumes of data that take hours to review can be communicated in a few seconds. For instance, many techniques are available to simulation analysts to perform verification (e.g., looking at simulation logs). However, a visualization of what occurred in the simulation model can reveal such errors very quickly. Similarly, communicating the working of simulation models to domain experts through visualization can allow errors in logic to be easily identified and corrected. This is the process of validation, and can be significantly enhanced by animating simulated operations. Through 3D visualization, more people can gain a better understanding of the modeled operations.

The remainder of this paper describes how 3D visualization was used to verify and validate the control logic of a simulation model of a complex earthmoving operation. In addition, the paper highlights how the improvement of the operation was facilitated due to the non-quantifiable and otherwise presentable visual insights provided by accurate 3D animation. The tool used in animating the operation, VITASCOPE, is also briefly described.

3. Earthmoving Operation Case Study

The case study presented here is a combination of two separate operations that took place in the states of Virginia and North Carolina. These two operations have been combined to
make it possible to illustrate two separate but interesting issues of relative complexity in a single exposition.

The presented operation involved moving 975,000 bank m$^3$ of material in 75 workdays (16 work hours each) from two possible sources to a common dumpsite as shown on the plan view in Figure 2. The two sources are located towards the bottom left part of Figure 2 and are labeled MLA (main loading area) and ALA (alternate loading area). The dumpsite is towards the top right part of Figure 2 and is labeled DumpArea. The haul distances from the main and alternate loading areas to the dump area were 1,670 meters and 1,920 meters. Both haul routes shared 1,370 meters and included a narrow segment 470 meters in length.

![Figure 2: Plan View of the Earthwork Operation Jobsite](image)

The narrow portion was not wide enough to allow simultaneous traffic in both directions. Due to the obstruction shown on Figure 2 and other site constraints, it was not feasible to widen the curve. The dump area was 42 meters above the main loading area and 68.5 meters above the alternate loading area. The underfooting in several parts of the haul routes was soft. The maneuvering space at the load and dump areas was limited.
4. Discrete Event Simulation Model

The simulation tool used to design this operation, STROBOSCOPE, is a programmable and extensible general-purpose system that is designed to model complex construction operations with utmost flexibility (Martinez 1996). STROBOSCOPE modeled in detail the transport portion of the operation, including dynamic truck routing strategies and the one lane (but bi-directional) haul road segment.

The primary control logic components, combinations of which were to be tested in the simulation model were:

- Truck Routing Strategy to main and alternate loading areas
- Traffic Management on the narrow one-way segment

In the initial routing strategy, trucks returning after dumping were routed to the main and alternate loading areas with likelihoods of 8 and 4 respectively, indicating a 66.7% probability that a returning hauler would go to the main loading area, and a 33.3% probability that the hauler chose the alternate loading area. This initial truck routing strategy was rather naïve but was the easiest to set up in an initial simulation model. Based on the insights gleaned from visualization, subsequent stages of the operation design explored more sophisticated truck routing strategies by using dynamic formulas to define likelihoods.

The default operating logic for the one-way segment marked for travel in either direction was defined such that a truck arriving to the empty segment established the current direction of travel. This direction was maintained as long as trucks kept arriving at the same end of the segment. Trucks eventually stopped arriving at that end and the segment cleared as the last truck exited. At that point direction of travel reversed if trucks arrived and were waiting at the other end. Otherwise it was again established by the next truck to arrive. In this study, it was necessary to analyze and optimize the transport capacity of the narrow segment, as it was the most constrained control parameter.
5. VITASCOPE

The VITASCOPE visualization system is implemented as a virtual environment application that can process ASCII text files (trace files) written in the VITASCOPE animation language to accurately describe the spatial and temporal characteristics of simulated operations (see Chapter 2 for details).

The trace file driven approach allows its seamless integration with numerous process modeling tools that are capable of generating formatted text output during a simulation run. The required trace file consists of sequential animation command statements such as CREATE, DESTROY, PLACE, and MOVE. In addition, the file also contains statements such as PATH and NONDIRECPATH that define resource movement paths during the animation. The statements in the input file are then processed sequentially to visualize the modeled operations in 3D virtual space.

This is accomplished using 3D CAD models of the involved system resources (e.g. Trucks and Loaders) and other model entities. The result is in essence a “motion picture” of the actual operations being carried out in the virtual environment. This “motion picture” can be replayed at varying speeds depending on the viewer’s preferences. In addition, the system also allows users to jump ahead or back to any point in simulation time which is fairly analogous to being able to instantaneously rewind and fast forward a motion picture tape to a desired location. The user is able to navigate easily in 3D virtual space and hence can position himself/herself at any vantage position he/she desires at any time during the visualization process.

Realistic animations can be created using 3D CAD models from supported data file formats such as .3ds (3D Studio™), .iv (Open Inventor™), and .wrl (VRML). Practically every CAD modeling program can export data files in VRML format. Thus, VITASCOPE is practically independent of any CAD modeling software as well.

VITASCOPE is designed to allow simulation model developers to accurately convey the essence of their simulation models in 3D. Doing so facilitates the verification and
validation of simulated operations, and helps establish credibility. In addition, it provides valuable visual insights into the modeled system that are difficult to be quantified and depicted numerically or by any other form of visualization.

6. Visualization of the Modeled Earthmoving Operation

STROBOSCOPE produces static output in the form of tables and charts. In addition, STROBOSCOPE models can be instrumented to generate animation trace files conforming to the syntax of the VITASCOPE language during simulation runs. The trace files are then processed by the VITASCOPE visualization engine to depict dynamic 3D output in the form of animations.

Simulation models need to be instrumented to generate VITASCOPE animation statements during a simulation run. The following line of code, for example, tells STROBOSCOPE to add two lines of text to the VITASCOPE trace file (named ATF in this case) every time a truck starts to haul.

```
ONSTART Haul PRINT ATF
"TIME %.2f\059
MOVE Truck%.0f HaulRoute %.2f\059\n"
SimTime Haul.Truck.ResNum Haul.Duration;
```

These two lines will be written to the VITASCOPE trace file numerous times, each of which will look similar to:

```
TIME 423.86;
MOVE Truck1 HaulRoute 83.21;
```

The time, truck number, and the duration to haul will of course be different each time. The VITASCOPE trace file will contain other lines of text that will be written out when other parts of the modeled operation take place. Thus, the time-ordered sequence of animation statements written out by all the activities in the model during a simulation run constitutes the trace file necessary to visualize the modeled operations.
The size of the generated trace files depends on the amount of detail modeled and the length of the simulation. The size of typical trace files will vary from a few hundred lines for simple models to several thousand lines for detailed and complex models that simulate operations over long periods of time. The trace file for visualizing 8 hours of the discussed earthwork operation is, for example, 36400 lines long. The STROBOSCOPE model required 44 animation-specific commands to do this. There is no limit on the size of the trace files that can be processed and is therefore not a constraining issue.

In the current visualization, the observer was able to examine the presented operation in a very realistic manner. In addition, the observer could “see” all the characteristics of the terrain such as gradients of the routes, the limited maneuvering spaces at the loading areas, the configuration of the one-way segment, and the limited visibility (due to steep grades) available to truck drivers approaching the junction from the loading areas (point JCT in figure 2). At all times, the observer could “move” to any desired location on the virtual jobsite using keyboard keys to steer. The level of detail at which the operation was visualized comprehensively established the veracity and the validity of the simulation model.

### 6.1 Verification

Differences between the mental plan of the modeler and the operation modeled in a first attempt are ubiquitous. The model of the presented operation was no different. The initial STROBOSCOPE model contained various coding errors. Some of these errors drastically altered the logic of the modeled operation. For instance, the most conspicuous of these errors altered the logic of the model to allow bi-directional traffic through the narrow one-way segment. This meant head-on collision between trucks traversing the segment. Figure 3 depicts an animation snapshot of the scenario.

Another error created discrepancies in the resolution of the right-of-way among trucks arriving at the Y-intersection (Point JCT in Figure 2). This allowed trucks arriving from either direction to traverse the intersection regardless of other present traffic. Figure 4
presents a snapshot of a near miss when an empty truck heading towards the alternate loading area suddenly cut across a loaded truck emerging from the main loading area.

Yet another modeling flaw allowed an empty waiting truck to start taking position under the main excavator even before the loaded truck ahead had cleared the loading area. Figure 5 depicts this scenario wherein the empty truck almost runs into the exiting loaded truck.

Visualization revealed all these errors in a few minutes. It took an equally negligible time to fix the coding errors in the STROBOSCOPE model. Some of these errors could have been detected using traditional verification techniques, albeit with a lot more time investment. For instance, we could have identified after some inspection of the logs that trucks traversed the one-way segment simultaneously in either direction. With a little more effort, we could also have predicted some confusion at the Y-intersection. It would,
however, have been very difficult to identify the discrepancy that caused an empty truck’s early intrusion into the loading area.

![Figure 4: Modeling Flaw - Near Collision at the Y-Intersection](image)

The reason for this can be explained by looking at the portion of the initial stroboscope model presented in Figure 6. The activity LoadTruck was immediately followed by the activity Haul. The activity Maneuver was programmed not to take place while the activity LoadTruck was in progress. At the instant at which LoadTruck ended and Haul commenced, however, a waiting truck would start to Maneuver. This seemed perfectly logical by looking at the model. The slow speed of the exiting loaded trucks and the relative agility of the empty trucks were, however not apparent by looking at the model or its trace. As Figure 5 presents, the faster speed of the empty trucks would almost cause them to hit the rear end of the exiting loaded trucks.
Figure 5: Modeling Flaw - Rear-end Collision at the Main Loading Area

Figure 6: Portion of the Initial STROBOSCOPE Model
This immediately suggested the need for an additional activity, ExitLoadArea, between LoadTruck and Haul that would also block the instantiation of the Maneuver activity. Such subtle errors, although not apparent using traditional verification techniques, are easily noticed by looking at a visualization for a few seconds.

6.2 Validation

The developer of a computer simulation model may be inexperienced or have misconceptions about how the actual operation will take place in the field. Thus, a model may not be an accurate representation of reality despite proper verification. A verified model accurately reflects what the model creator intended. The aim of validation therefore is to determine whether a simulation model accurately represents the real-world operation. This is typically carried out by consulting people who are intimately familiar with the operations of the actual system, but who are not necessarily proficient in simulation.

Due to our background in construction engineering and general knowledge of earthmoving operations, we were competent enough to validate the presented simulation model to a large extent. However, for the purpose of completeness in the investigative exercise, we presented the simulation model and the visualization to an experienced local earthmoving contractor.

The contractor, looking at the visualization of the digging excavator (main loading area) contended that it (the excavator) appeared to be digging too fast for the type of terrain it was operating on. To our surprise, subsequent careful examination of the underlying data used in the simulation model confirmed that the data used to establish the probability distribution came from a different study. The fit of the distribution to the data had been validated through all appropriate statistical methods; the fit was indeed good and proper. It was, however, a fit to data for a different job study. The experienced contractor was able to recognize this immediately by looking at the visualization.
Visualizing the operation that was modeled using bad data provided a tangible opportunity for the domain expert to notice this discrepancy during visualization. It would have been very difficult indeed for the contractor, who was not proficient in simulation, to notice this flaw had he been told that the probability distribution for the loading time of the main excavator was a Normal with a mean of 48.5 seconds and a standard deviation of 7.25 seconds. Visualization made all the difference.

### 6.3 Insightful Visual Details

In addition, visualization provided several non-quantifiable and otherwise presentable details that were critical in making decisions. The basic problem with the narrow one-way segment was that loaded trucks, traveling uphill, were very slow. They arrived at the curve at such an interval that they entered when a previous truck was almost exiting. The direction of travel was thus maintained in the loaded direction for very long periods, during which empty trucks arrived and bunched at the other end. When empty trucks entered the curve, however, they traversed and cleared it very quickly.

The dynamic output produced by the VITASCOPE provided a much better picture of the truck bunching and additionally revealed strategies that could be used to improve the operation. Figure 7 shows a snapshot of the animation with 5 empty trucks bunched up waiting to enter the big one-way segment; one loaded truck about to enter the curve before another loaded truck finishes traversing it; and one loaded truck heading towards the dump area. The slow speed of the loaded trucks and the fast speed of the empty trucks as they traverse the curve cannot be seen on the snapshot. Only the animation can convey that information.

Figures 8 and 9 show snapshots of the animation a while later, when the trucks that had bunched up have arrived almost together to the main and alternate loading areas. A few trucks are out of view in the snapshot but will arrive soon to the loading areas. The visualization provided clear indications that the entry of loaded trucks to the segment had to be controlled so that empty trucks could traverse it in smaller bunches.
Figure 7: Bunched up Trucks Waiting to Enter the Narrow Segment

Figure 8: Bunched up Trucks at the Main Loading Area
The visualization also revealed the ineffectiveness of the probabilistic (but random) truck routing policy; at times trucks were routed to an excavator that was busy and had a long queue of trucks waiting to be served even though the other excavator was free. Visualizing the operation clearly indicated that the percentages naively used in the preliminary design were not a bad choice, although the routing method itself was.

![Figure 9: Bunched up Trucks at the Alternate Loading Area](image)

#### 7. Conclusion

The purpose of using simulation to model construction operations is to test and obtain insights into the consequences of using various construction alternatives. The results of simulation are expected to help the planner in making the most advantageous decisions. It is of utmost importance, however, that simulation models be credible if they are to be accepted and used as a decision-making aid.

The presented research investigated the effectiveness of dynamic 3D visualization in the verification and validation of discrete-event construction simulation models. The paper
demonstrated, with the help of a case study that visualizing simulated operations in 3D can be an effective means of accurately communicating modeled operations, especially to decision makers who are not necessarily proficient in simulation. This is of significant help in verifying and validating simulation models thus establishing their credibility.

The paper also demonstrated that the dynamic visual output provided by 3D visualization can provide several subjective details about the operations that can be of immense help in decision-making. In addition, the reader was briefly introduced to research being conducted at Virginia Tech to enable smooth dynamic animation of discrete-event construction simulations.

8. References

Chapter 10
Conclusion

In the domain of operations design and analysis, the ability to see a 3D animation of processes that have been simulated allows for three very important things:

1. The developer of the simulation model can make sure that there are no errors in the coding (Verification).
2. The experts, field personnel, and decision makers can discover differences between the way they understand the operation and the way the model developer understands it (Validation).
3. The model can be communicated effectively which, coupled with verification and validation, makes it “credible” and thus used in making decisions.

This research successfully investigated methods to accurately describe and portray the performance of modeled construction processes in smooth, continuous, dynamic 3D virtual worlds. The tangible product of the work is VITASCOPE, an open, loosely-coupled, software-authorable, straight-line, parametric animation description language that can generally describe the performance of simulated construction operations in 3D. The language is implemented as a scalable and extensible tool that can process sequential animation-describing statements to graphically portray modeled construction processes with spatial and chronological accuracy.

In addition to helping accredit discrete-event construction simulation models, VITASCOPE animations can be potentially exploited in many other ways in construction practice and education. Using available CAD models of infrastructure and the resources, it is possible to re-create in a virtual world what happened in the past (from trace-driven simulation) or what may happen in the future (by showing what was simulated by a simulation model). These visualizations can be very realistic, with accurate depictions of construction sites, infrastructure, equipment, and atmospheric conditions (e.g. rain, snow). Historical (from past data) and predicted (from simulations) animations can be in
compressed or expanded time. A 20 second incident can be studied in very slow motion.
General operations, in contrast, can be animated in fast motion so that several hours of
operations are viewed in a few minutes.

VITASCOPE’s first forte is its open and loosely coupled 3D animation scheme. The
language can be used to animate processes modeled with any simulation tool capable of
generating formatted text output during simulation runs. VITASCOPE is specifically
designed to animate modeled construction processes and supplement advanced
construction simulation tools such as STROBOSCOPE (Martinez 1996). However,
VITASCOPE can also animate processes modeled in other simulation languages such as
GPSS (Schriber 1995), SLX (Henriksen 1998), Extend (Krahls 2002), Slam (O'Reilly
1994), Siman (Pegden et al. 1995), and Simscript II.5 (Russell 1993); and programming
languages such as C (Kernighan and Ritchie 1988) and C++ (Stroustrup 2000).
VITASCOPE’s independence from animation-authoring simulation tools is, by design, a
radical departure from other schemes of animating simulations in 3D.

In particular, 3D animation tools from certain manufacturing modeling systems (e.g.
AutoMod, Quest) are tightly coupled to their simulation engines. In other words, one
cannot use their animation methods without also using their simulation tools. This is
unacceptable because 1) manufacturing simulations tools are not a natural choice for
modeling construction processes (Martinez and Ioannou 1999) and/or animating them
(Tucker et al. 1998); and 2) simulation practitioners invest significant time and effort in
becoming proficient with particular simulation tools of choice, and are typically reluctant
to learn and use an entirely different, perhaps unsuitable simulation tool solely for 3D
animation purposes. Vendors of tightly-coupled, animation-integrated simulation tools
often maintain that their approach is the only way to add animation to a simulation
(Henriksen 1998). Success in designing and implementing VITASCOPE has proved that
that is not the case.

VITASCOPE’s second distinctive feature is its extensibility. The designed language is
implemented in an extensible and scalable framework that defines an add-on interface to
the core 3D animation statements. This flexible add-on interface allows others to extend VITASCOPE’s animation description language by designing and implementing custom 3D animation statements and functions. VITASCOPE add-ons can be designed for animating specific complex simulation models, or they can be general extensions that become part of the VITASCOPE language and can subsequently be used for animating any modeled processes. Others can exploit VITASCOPE’s extensibility (i.e. extend the language) without having to understand or modify the implementation (i.e. source code) of VITASCOPE’s core animation statements or any other prior add-ons. Extensibility of this nature is imperative to allow the state-of-the-art to incrementally evolve through the collective efforts and skills of others. The efficacy of several, non-trivial VITASCOPE extensions (i.e. add-ons) designed in this research prove that the approach is not only possible, but also very effective.

1. Fundamental Capabilities Provided by VITASCOPE to Visualize Simulated Construction Processes

VITASCOPE provides essential capabilities that enable it to animate simulated construction operations in smooth, continuous, dynamic, 3D virtual worlds. These capabilities are outlined in the following subsections in terms of what is necessary to accurately describe and portray a complex construction operation in a visually convincing manner that prompts instant credibility.

1.1 Concatenation of Elementary Geometric Transformations

An animation description language’s statements must be able to generally describe all the motions construction resources (e.g. equipment and labor) undergo as they perform communicated processes. Communication with virtual simulation objects (e.g. a piece of equipment) can only be achieved in a computer interpretable vocabulary. A virtual simulation object (e.g. a backhoe) cannot be directly told to perform a basic construction task (e.g. dig dirt). In fact, such an object cannot be directly told to perform even elemental motions (e.g. swing cabin, lift bucket). In order to communicate instructions to such virtual resources in computer interpretable vocabulary, elemental motions involved
in performing construction must be further broken down into geometric transformations such as rotations and translations.

Just as each basic task (e.g. dig dirt) is comprised of a set of elemental motions (e.g. swing cabin, lift bucket), every elemental motion can be dismantled into a set of geometric transformations (e.g. rotate bucket, rotate stick, rotate boom). The rigid motions of all construction equipment and craftsmen can ultimately be broken down into concatenated rotations, translations, and/or other geometric transformations. This is the only level at which instructions to perform construction tasks can be directly communicated to virtual pieces of equipment and craftsmen.

### 1.2 Accurate Resource Motion Trajectories

The shape of a trajectory that a virtual simulation object (i.e. construction resource) must travel on during performance of a communicated construction task is arbitrary. The features of a resource motion path depend on the spatial configuration of the process being modeled and the pertinent construction resource; and cannot be defined or stored a-priori. The trajectories on which resources move during the animation of a simulated construction process must thus be defined by a simulation model itself using parametric-text based statements that are both simple (so that a simulation model can author them) and powerful (so that they can flexibly describe and manipulate a 3D motion trajectory of any complexity). Such defined motion trajectories must maintain their spatial integrity in a dynamic construction environment where the shape of the terrain under the defined paths can itself change shape (i.e. deform) over the course of construction. In addition, the paths must provide a geometric basis for determining the correct spatial orientations of resources that traverse them.

### 1.3 Variable Speed Resource Motion

In authoring 3D animations of modeled processes, discrete-event simulation models can only provide information (to the animation methods) about when activity instances begin and end. The models can provide no guidance on the rate at which the tasks in communicated activity instances are performed. Such information is however required to
describe the realistic, uneven velocity profiles with which construction resources move in reality. Since such information must be formulated externally (i.e. cannot be extracted from simulation models), it is imperative that the temporal integrity of the underlying simulation models is not violated i.e. any externally computed temporal variables must perfectly coincide with an underlying simulation model’s event times (activity start and end times).

### 1.4 Automated Landscape Generation

Digital elevation and aerial imagery data is readily available for the entire United States and several other parts of the world. However, this data is archived in several different digital formats at varying levels of detail. In order to automatically generate virtual jobsite terrain databases (i.e. 3D CAD models) from archived digital data, it must be possible to 1) parse (i.e. read) and interpret streams of elevation and imagery data in all commonly available formats, and 2) convert interpreted elevation and imagery data into a standard internal representation that can be visually portrayed in a 3D animated world, and can also be locally modified as needed to describe actions such as terrain deformation.

### 1.5 Dynamic Construction Jobsite Terrains

The resultant locally deformed shape and appearance of a virtual terrain in response to an animated construction task such as digging dirt cannot be determined a-priori by the underlying simulation model that authors the animation, or by the animation engine itself. Each deformation a virtual terrain must undergo in response to an animated construction task (e.g. digging) depends on 1) the type, size, and configuration of the involved piece of virtual equipment (e.g. backhoe), and 2) the amplitude of the motion of its components (e.g. boom, stick, bucket) in the particular animated instance of that task.

For instance, a model that communicates simulated earthmoving operations using parametric animation statements can indicate the location (as a 3D coordinate) where a virtual piece of equipment (e.g. backhoe) must “dig” in each loading pass. However, the exact trajectory that a piece of equipment’s digging implement (e.g. bucket edge) will
follow in each virtual digging stroke cannot be determined beforehand by either the authoring process (i.e. simulation model) or the animation engine. The calculations that determine the shape of an evolving terrain in response to animated construction processes must thus be performed during animation in real-time and is solely the responsibility of the animation engine.

1.6 Fluid Construction Materials

Geometric transformation based semantics can describe any rigid motion that resources undergo as they perform communicated construction tasks. However, volumes of fluid construction materials such as concrete, dirt, gravel, mortar, sand, slurry, and water naturally do not have fixed deterministic shapes and forms, and cannot be described by concatenating rigid geometric transformations. Such materials flow under the influence of prevailing natural (e.g. gravity) and imparted (e.g. pump pressure) forces until physical equilibrium is established. Computational fluid dynamics literature provides classic models such as the Navier-Stokes equations to describe the motion and behavior of flowing liquids. Such models although highly accurate, require very intensive computation to solve.

An animation description language designed to allow simulation models to communicate construction processes involving fluid construction materials demands that the methods (i.e. statements) themselves be simple enough that their syntax and the values sought by their parameters are both within a model’s authoring capabilities. The animation methods must thus incorporate rich semantics so that dynamic arbitrary volumes of ubiquitous fluid construction materials can be described with minimal inter-process communication.

1.7 Multiply Articulated Construction Equipment

When elemental sub-tasks (e.g. lower boom, scoop dirt, swing loaded, etc.) involved in performing basic construction tasks (e.g. dig dirt) are not explicitly modeled (which is typically the case), discrete event simulation models do not encapsulate and therefore cannot communicate information about when those elemental sub-tasks begin or end. This information is however critical to 3D animation because geometric transformation
Based elemental tasks are the necessary building blocks for describing the motions of the multiply-articulated construction resources (e.g. backhoes, cranes) that perform the simulated construction processes virtually on the computer.

An animation description language for describing modeled construction operations must thus realize “smart” pieces of virtual, articulated construction equipment that can be instantiated and manipulated using simple, parametric-text statements in a higher-level, contextual, construction task terminology. A simulation model can then instruct such articulated equipment pieces to perform basic construction tasks (e.g. dig dirt) and relegate to the animation methods the task of computing the multiple elementary geometric transformations that a resource’s components (e.g. boom, stick, bucket) must undergo to visually depict the communicated construction task.

1.8 Interference Detection and Reporting

In addition to helping verify and validate the logic of discrete-event simulation models, 3D animation must accurately validate whether a modeled (and animated) construction operation is physically executable. The kinds of undesired collisions and interferences that can occur in an animation of a modeled construction operation encompass the entire range of undesired conflicts that can occur on real construction sites. These include potential clashes among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources. Due to the large number of potentially interfering simulation objects (i.e. construction resources) active on a typical animated construction site, efficiency is interference detection computations is of paramount importance.

1.9 Quantitative Run-Time Evaluation

During a 3D animation of a simulated construction operation, the dynamic display of pertinent quantitative information can be useful and at times critical in conveying critical details of the operation being visualized. The dynamic quantitative information that can be graphically portrayed during a 3D animation of a simulated operation includes the numerous simulation run-time statistics maintained by a running process model. The
ability to juxtapose dynamic displays of quantitative, numerical simulated operation data alongside 3D view ports during visualization affords significant incremental utility from 3D animation.

2. Contributions

This research established the knowledge that allows accurate, smooth, continuous, dynamic 3D visualization of construction operations modeled using discrete-event simulation. The research contributes to construction operations design by improving the verification, validation, and communication of discrete-event simulation models of construction operations; which in turn makes models more credible and thus used in making decisions. The work contributes to the computing infrastructure for research by enabling other investigators to pursue avenues of discovery that rely on the capability of visualizing a simulated construction operation in 3D.

This work also contributes to the computing infrastructure for construction education. By addressing the innate human ability to process graphic information, educators can, in a very short time, impart young engineers with the understanding and decision-making skills that would otherwise take years of risky field experience. Related contributions are afforded to the performance of actual operations in the field, by allowing proper communication of the work to be performed prior to its execution.

In addition, the contributions of this work are the basis for advances that can enable real-time, immersive virtual construction environments to be commonplace. In these virtual environments, objects under the control of simulation models will be aware of, and react to, humans and human controlled machines. Discoveries leading to such environments are directly dependent on, or are greatly facilitated by the knowledge that enables the 3D visualization of discrete-event operations simulation, if the latter is based on a loosely coupled, general-purpose methodology such as VITASCOPE’s.

Similar benefits accrue in other civil engineering disciplines such as transportation (especially aviation) in addition to other domains such as shipbuilding, aerospace,
manufacturing, and the service industries wherein the necessity to effectively communicate simulations is as acute as in construction. Success in this research also provides visibility for our work in construction to diverse domains where simulation is commonly used as an operations analysis tool.

The following list summarizes some of VITASCOPE’s specific technical contributions to the state-of-the-art:

- An open, loosely-coupled, animation language-based 3D visualization scheme to describe the accurate performance of construction operations in smooth, continuous, dynamic virtual worlds.
- Simple, expressive, geometric transformation-based, parametric animation statements that can describe all rigid elementary motions resources undergo as they perform construction work.
- A portable, virtual environment framework that parses, interprets, and processes sequential animation instructions; and recreates a dynamic, smooth, continuous visual representation of the communicated construction operations using references to 3D CAD models of the involved construction resources.
- Mechanisms that allow the core 3D animation description language to be seamlessly extended by others via a flexible extension (add-on) interface.
- Techniques that textually communicate and accurately portray volumes of ubiquitous fluid construction materials such as concrete, mortar, water, and slurry in animated virtual construction worlds.
- “Smart” pieces of virtual construction equipment that can be instantiated and manipulated in animated construction worlds using simple text statements in a higher-level, contextual, construction work-like terminology.
- Automated techniques to define 3D virtual construction jobsite terrain databases (i.e. CAD models) by combining readily available digital topographical (e.g. USGS Digital Elevation Maps) and aerial imagery (e.g. National Aerial Photography Program) data.
- Methods to depict dynamic, deformable terrain in animations of simulated construction processes such as digging and dumping dirt.
• Simple, parametric-text, simulation model-authorable statements to define and manipulate curved trajectories of arbitrary shape and length to represent 3D resource motion paths.

• Geometric techniques to accurately describe the three-dimensional spatial configuration of virtual construction resources as they travel on defined motion trajectories over uneven construction jobsite terrain.

• A unique computation scheme to portray variable-speed motion of simulation objects (i.e. construction resources), using only a one-time, simulation model-authorable, parametric-text definition of the desired velocity profile shape and per-instance activity timing information (i.e. start time and duration).

• A text statement-controlled dynamic charting tool to graphically animate simulation statistics alongside 3D view ports during animation of simulated construction processes.

• A comprehensive interference detection framework to identify and report undesirable conflicts and/or collisions that can occur among static (e.g. structure in-place, idle equipment), dynamic (e.g. active machines and workers), and abstract (e.g. hazard or protected spaces) construction resources in animations of simulated construction operations.

3. Implementation and Validation

In some areas of research, the implementation of an idea is separate from the idea itself. For example, a new method for the solution of a linear programming problem could be sound, precise, and effective. It could be validated on paper without the need for a corresponding computer program, or it could be validated with a poorly written computer program that is only capable of solving a few small problems. In some areas of scientific computing-oriented research, however, this is not the case.

3D visualization is exclusively a computer-based activity. A 3D animation description language that is not well implemented cannot be used to validate the ideas behind it. The concepts behind the designed language need to be exercised by animating simulated construction operations whose performance is well understood, and that in turn will allow
the validation of the animation language and the implementation itself. The design of the VITASCOPE animation description language and its implementation were thus inextricably interrelated and were carried out in parallel.

In order to validate VITASCOPE and its add-ons, we created several discrete-event simulation models of construction processes and instrumented them to generate animation traces during simulation runs. The generated traces were then post-processed in VITASCOPE’s virtual environment application to investigate the efficacy of the designed language in animating modeled construction processes. Trace file segments and snapshots of several such animations (e.g., earthmoving, block masonry, concrete delivery and placement, steel erection, etc.) were presented in the preceding manuscripts of this dissertation.

In addition to those examples, a capstone simulation-driven visualization of operations involved in the construction of a bridge was developed to investigate the combined effectiveness of VITASCOPE’s core and extended animation language statements. In particular, we simulated and animated the construction of a five-span, cast-in-place, balanced-cantilever, segmental concrete bridge. The specific construction processes that were modeled and animated include: 1) Excavation of pier foundations, 2) Placing concrete pier footings, 3) Casting pier shafts using modular climbing forms, 4) Casting massive pier tables, and 5) Casting the balanced-cantilever superstructure segments (box girders) using form travelers.

Figure 1 presents snapshots of these animations. The construction processes themselves are described in detail in (Lucko and de la Garza 2003). The visualization of bridge construction, by virtue of the complex involved processes, exercised the statements of the entire VITASCOPE language and all its add-ons in a single exposition. This was particularly useful in validating critical aspects of the designed animation description language.
(a) Excavating Foundations   (b) Placing Concrete Footings

(c) Casting Pier Shafts   (d) Casting Pier Tables

(e) Casting Decks (Initial Segments)   (f) Casting Decks (Final Segments)

Figure 1: Animated Segmental Concrete Bridge Construction Processes

VITASOPE’s validation exercise itself consisted of the following four steps:

1. Validation of the language’s Simplicity.
2. Validation of the language’s Sufficiency (Semantic Richness).
3. Validation of the Portability of the language and its implementation.
4. Validation of the language’s Extensibility and its implementation’s Scalability.
3.1 Validating Simplicity

Evaluation of VITASCOPE’s simplicity was an essential step in the conducted validation exercise. Simplicity, in this context, meant that the constructs (i.e. statements) of the designed animation description language and the information sought by the statements’ parameters were both within the authoring capabilities of discrete-event simulation models. Discrete-event simulation tools are generally limited in the content and format of output that running models can generate because the models encapsulate operational information only to the extent necessary to come up with meaningful quantitative results.

VITASCOPE’s simplicity was validated by post-processing animation trace files generated by discrete-event simulation models and examining the extent of operational information that the models can communicate in the syntax of the designed language. The visually accurate results portrayed by processing the animation trace files verified that:

- VITASCOPE’s syntax was software-authorable; and a running discrete-event simulation model could easily generate, format, and output parametric-text statements in its language.
- Operational information sought by the numerical parameters of VITASCOPE’s animation statements can be extracted from a running simulation model. In addition, intelligent assumptions can compensate for the lack of any non-extractable operational data.
- A running simulation model can generate an animation of construction processes of any length since the sequential trace file that describes the animation can be arbitrarily long.

3.2 Validating Sufficiency

Validating VITASCOPE’s sufficiency (i.e. semantic richness) involved confirming the expressiveness of its succinct, parametric, animation statements. In particular, the designed language statements were evaluated to check whether they can collectively encapsulate all operational information required to describe a smooth, continuous, construction operation in 3D. Sufficiency and simplicity were two conflicting objectives
in VITASCOPE’s design. While simplicity strives to capture details with minimum requirements on information input, semantic richness suggested that the language statements be designed to encapsulate detailed information about described construction operations. These two objectives (sufficiency and simplicity) were carefully considered and balanced in designing the VITASCOPE language.

In addition, VITASCOPE’s sufficiency was validated alongside its simplicity by examining the degree to which generated animations were faithful to the underlying modeled (and communicated) construction processes. The quality and detail of animations achieved by processing the generated animation trace files confirmed that:

- Operational information encapsulated in succinct, sequential VITASCOPE language statements can generally capture the description of a dynamic, continuous construction operation.
- The animations of construction operations achieved by processing sequential VITASCOPE language statements are absolutely faithful in temporal and spatial accuracy to the operational information communicated by the underlying simulation models.

### 3.3 Validating Portability

Another important step in VITASCOPE’s validation exercise was confirming the portability of the designed animation description language and the software tool that implements it. Portability in this context refers to the ability of the designed language and its implementation to be compatible with multiple computing platforms that range from high-end graphics workstations to common desktop and laptop computers. VITASCOPE implements its own interpreter that reads sequentially recorded parametric-text animation statements, organizes the information captured therein, and calls appropriate computer graphics functions/routines to manipulate 3D CAD models of pertinent resources. The software tool that implements the VITASCOPE animation language relies on the OpenGL Optimizer (Silicon Graphics 1998) Scene Graph Application Programming Interface (API) as the graphical backend to perform all low-level graphics computations required to actually display a construction operation in 3D.
The OpenGL Optimizer Scene Graph API was chosen after carefully considering and comparing the features of several other popular APIs. The Optimizer API is compatible with SGI IRIX and Microsoft Windows platforms. Between them, these computing platforms represent the majority of high-end graphics workstations, as well as common business desktops and laptops. VITASCOPE’s portability was thus ascertained by implementing the interpreter software tool on both computing platforms currently supported by the Optimizer API. Comparison of the visualizations achieved by processing animation trace files on both the platforms established that:

- VITASCOPE’s efficacy in animating modeled construction processes is independent of any computing platform-specific issues. The quality of the achieved animations is only influenced by the raw computing power of the native computer and not on the platform it operates on.

- The tool that implements the VITASCOPE language is tightly coupled with the Optimizer API. Thus, VITASCOPE can only be ported on platforms that support the Optimizer API (currently IRIX and Windows).

### 3.4 Validating Extensibility and Scalability

Evaluation of VITASCOPE’s extensibility and scalability comprised the final important step in the validation exercise. Validating extensibility in this context meant confirming that the designed VITASCOPE animation description language could be seamlessly extended by others. Verifying scalability, on the other hand, involved measuring the ability to retain animation performance and quality as the language is extended. In terms of the computing context, scalability is the ability to not only function well in a rescaled situation, but to actually take full advantage of it.

In particular, VITASCOPE would be scalable if performance levels in a particular computing context are maintained as more features (e.g. statements) are added. Similarly, from the viewpoint of the computing context itself, the language and its implementation would be scalable if they could be ported to a more powerful computer and took full advantage of the better system in improving animation performance and/or quality. The validation of VITASCOPE’s extensibility and scalability were thus mutually interrelated.
Several non-trivial extensions to the core animation description language were designed and implemented in this research using VITASCOPE’s flexible add-on interface. All the data, variables, language statements, and functions that describe such extensions are physically contained in separate libraries (DLLs on Windows and DSOs on IRIX) that call back (as necessary) into the core VITASCOPE visualization engine via the defined API. Experience with designing the presented language extensions (add-ons) and examination of several animations (particularly the visualization of bridge construction processes) described by core and extension statements on different computing platforms confirmed that:

- Others can design extensions to the VITASCOPE animation description language without understanding and/or modifying the implementation (i.e. source code) of the existing language set.
- New designed statements can be used in animations as though they were part of the original language, i.e. the end-user of the language perceives no difference in the usage of core and extended language statements.
- Visualization performance is unaffected by the size of the animation description language set, i.e. an animation’s frame rate is not affected adversely as additional extensions to the language are designed and implemented.
- VITASCOPE’s implementation takes conspicuous advantage of a host computer’s raw computing power. On any given configuration, the animation frame rate displayed is the maximum that can be achieved on the given computer.

4. Directions for Future Research

This research investigated methods that enable the core technologies required to graphically portray modeled construction processes in dynamic, smooth, and continuous 3D virtual worlds. The work, while significantly advancing the state-of-the-art, has revealed several other interesting research issues that future work can address. Several such initiatives were presented at the end of most preceding manuscripts of this dissertation. In addition, the following subsections describe specific broader research opportunities for holistically exploiting and/or extending VITASCOPE.
4.1 Integration of Simulation and CAD Modeling Tools

Visualization of construction operations, by definition, requires the creation, organization, and maintenance of a database of 3D CAD models of the constructed facility components (i.e. the construction product) and the resources – materials, equipment, temporary structures etc. – that are involved in the processes that build it. In animating construction processes that assemble tangible facilities (e.g. buildings, plants, bridges), the ability to exploit spatial information embedded in 3D CAD product models from the design stage can significantly streamline the process of animating the construction operations involved in building the facility. This requires exploration of methods that facilitate inter-process communication between simulation modeling tools and 3D CAD modeling systems. Success in such work will allow simulation models to automatically extract product model features from 3D CAD drawings of designed facilities and communicate them to the animation methods.

Automatically extractable information that describes the final spatial configuration of constructed components can significantly enhance simulation models’ capability in communicating the construction processes that assemble those components. In addition, extracted product characteristics can be exploited in describing the numerical parameters of the simulation models themselves. Future research initiatives should explore, in particular, the role of interoperability standards such as Industry Foundation Classes in facilitating the desired integration between simulation and CAD modeling tools (Bazjanac and Crawley 1997). The work can subsequently lead to the design of an integrated simulation-driven, animated construction operations workspace that not only models and animates construction processes but can also communicate a wide range of non-CAD project data.

4.2 Special-Purpose Animated Construction Simulators

VITASCOPE has been designed to allow general-purpose simulation languages and tools to author animations of modeled processes. However, VITASCOPE can also be seamlessly integrated as a 3D animation component in any special-purpose simulation
tool (i.e. simulator). This is possible due its flexible, adaptive design, and the open, loosely-coupled animation scheme it supports. Advanced simulation languages such as STROBOSCOPE often demand a level of training that is beyond that which can be found in typical current practitioners.

Special-purpose simulators, on the other hand, bring the benefits of discrete-event simulation to users with little or no training by providing an environment that is very close to the problem and geared towards a narrow domain (e.g. earthmoving, crane operations, concrete placement). Integrating automated 3D animation capabilities in such simulators can significantly enhance their efficacy and appeal, thus facilitating their widespread deployment in construction practice. From the research standpoint, the emphasis while designing such animation-integrated construction simulators should be the study of strategic issues in the specific construction domains and not on programming the simulators.

### 4.3 Concurrent Animation of Field Construction Processes

An open, loosely-coupled visualization scheme such as VITASCOPE’s suggests that animations need not necessarily be generated by discrete-event simulation models. An animation of a dynamic construction operation can be authored by any external operation-describing process capable of generating formatted text output in VITASCOPE’s syntax. The API defined by VITASCOPE’s extensible framework allows the execution of a straight-line, animation-describing program that is concurrently being generated. For instance, a real-time data stream (e.g. from GPS sensors mounted on equipment pieces) could, after due intermediate processing, drive an animated, 3D virtual construction world that replicates a jobsite’s current status.

The ability to remotely visualize the accurate locations and orientations of field resources, while being able to freely navigate in the virtual (possibly immersive) 3D construction world can afford significant, real-time decision-making capabilities to practitioners. It is envisioned that the specific work required to implement such a capability would involve the design of techniques to process the communicated data
streams, and convert them to the appropriate VITASCOPE language syntax before feeding them to the visualization engine. The work may also require the identification of any latency that might exist in the data communication process and appropriate methods to alleviate any resulting adverse artifacts.

4.4 Interactive, Simulation-Driven Virtual Construction Environments

Separation between the animation of an operation and the process that generates it opens another interesting research opportunity. Future work can explore methods to enable virtual, immersive, construction worlds that allow interactive participation in a running simulation. The possibilities of interacting with and controlling a simulated operation via animated virtual worlds have the potential to add significant value beyond passive visualization as far as operations design, operator training, and learning is concerned. In these virtual environments, objects under the control of simulation models will be aware of, and react to, humans and human-controlled machines.

The critical challenge in accomplishing this work is the design of the interaction interface between users and running simulation models. In addition, the visualization engine must communicate bi-directionally and at high speed with simulation models running in another process and perhaps in another machine. It is envisioned that the specific work required to design such technologies would involve 1) Exploration of relationships between hardware (e.g. joysticks), communication methods, and the latencies that will exist between program generation and its visualization; 2) Investigation of methods to minimize the latencies so that they are imperceptible, i.e. exploration of methods to minimize the time lapse between event generation and its visualization; and 3) Design of methods that will effectively, and with minimum latency, allow the visualization engine to communicate events generated by virtual reality gear, to the process that is generating the straight-line program being visualized. This will be essential to enable interactive participation in a running simulation.
5. References


Appendices
Appendix A
List of Core VITASCOPE Language Statements

This appendix lists all the statements available in the core VITASCOPE 3D animation description language. The statements are grouped according to their functionality. Details of these statements and examples of their usage are described in appendix B.

System Statements

TIME <EventTimeValue>;
TIMEJUMP <TimeValue>;
TIMESKIP <TimeUnitsToSkip>;
VIEWRATIO <ViewRatioValue>;
LOADADDON <AddOnName>;
SCHEDULE <AnimationStatement> <ExecutionTime>;
END;
/ <Comment>

Scene Construction Statements

CLASS <ClassName> <CADFileName>;
ORIENTCLASS <ClassName> <AboutAxis> <RotationAmount>;
CREATE <ObjName> <ClassName>;
DESTROY <ObjName>;
ATTACH <ChildObjName> <ParentObjName> <AttachPoint>;
ATTACHNOSCALE <ChildObjName> <ParentObjName> <AttachPoint>;
DETACH <ChildObjName>;
CHANGECLASS <ObjName> <ClassName>;
PLACE <ObjName> AT <PlacePoint>;
PLACE <ObjName> ON <PathName>;
HORIZORIENT <ObjName> <RotationValue>;
VERTORIENT <ObjName> <RotationValue>;

290
SIDEORIENT <ObjName> <RotationValue>;
PATH <PathName> <Points>;
NONDIRECPATH <PathName> <Points>;

**Property-Setting Statements**

SET CLASS <ClassName> RGP <Value>;
SET CLASS <ClassName> FORECLEARANCE <Value>;
SET CLASS <ClassName> AFTCLEARANCE <Value>;
SET OBJECT <ObjName> RGP <Value>;
SET OBJECT <ObjName> FORECLEARANCE <Value>;
SET OBJECT <ObjName> AFTCLEARANCE <Value>;
OBJECTSTAT <ObjStatName> <InitialString>;
ATTACHSTAT <ObjStatName> <ParentObjName> <AttachPoint>;
UPDATEOBJSTAT <ObjStatName> <NewString>;
OPERATIONSTAT <OperationStatName> <InitialString>;
UPDATEOPERSTAT <OperationStatName> <NewString>;

**Dynamic Statements**

MOVE <ObjName> <PathName> <TravelDuration>;
MOVESPEED <ObjName> <PathName> <TravelSpeed>;
SLIDE <ObjName> <TranslationValue> <TravelDuration>;
TGTSLIDE <ObjName> <TargetPosition> <TravelDuration>;
ROTATE <ObjName> HOR <RotationAmount> <RotateDuration>;
TGTROTATE <ObjName> HOR <TargetRotation> <RotateDuration>;
ROTATE <ObjName> VERT <RotationAmount> <RotateDuration>;
TGTROTATE <ObjName> VERT <TargetRotation> <RotateDuration>;
ROTATE <ObjName> SIDE <RotationAmount> <RotateDuration>;
TGTROTATE <ObjName> SIDE <TargetRotation> <RotateDuration>;
SCALE <ObjName> <ScaleFactor> <ScaleDuration>;
TGTSCALE <ObjName> <TargetScaleFactor> <ScaleDuration>;
View Manipulation Statements

VIEWPOINT <Name> <Position> <RotationAxis> <RotationAmt>;
ATTACHCAMERATO <ChannelNum> <ObjName> <AttachPoint>;
DETACHCAMERA <ChannelNum>;;
Appendix B
VITASCOPE Language Reference

This appendix describes the animation statements available in the core VITASCOPE 3D animation description language. The syntax and semantics of each statement are described with examples of their usage.

How VITASCOPE’s Animation Clock Works
VITASCOPE’s animation clock generally matches the clock of the simulation models that author animations. Whenever a running simulation model writes a statement to an animation trace file that includes a time value or activity duration, it typically uses the same time units that are used in the model. This can be changed, however, by having the simulation model scale all time values before they are written to a trace file. VITASCOPE measures time in floating point animated time units. One time unit can equal whatever duration is most suitable for the animation (e.g. a microsecond, a second, a minute, an hour, a day) as long as it matches (or uniformly scales) the time unit in the simulation model that is driving the animation.

The TIME Statement
The TIME statement is the primary time-related VITASCOPE animation statement.

Syntax: TIME <EventTimeValue>;
Example: TIME 25.12;

The TIME statement waits for the animation clock to reach the new event time value specified as the argument. VITASCOPE then executes the animation statements that follow the statement until another TIME statement is reached. When a TIME statement is encountered in a trace file, VITASCOPE initially verifies that EventTimeValue is greater than or equal to the current animated time. If not, the animation terminates with an error. After ascertaining that the TIME statement specifies a future animation time value,
VITASCOPE suspends the reading of any more lines from the trace file until the animation time specified by the TIME statement has been reached or exceeded. When that happens, VITASCOPE reads and processes the next line(s) in the trace file until another TIME statement is encountered. Statements are read and processed in this manner until the end of the trace file is reached or the viewer interrupts the animation. The reading and processing of the trace file statements is practically instantaneous. All the while, VITASCOPE continues to display the animation as it progresses.

**The TIMEJUMP Statement**

The TIMEJUMP statement performs a fast forward or rewind to a desired animation time. When a TIMEJUMP statement is encountered in a trace file, VITASCOPE suspends the animation and quickly reprocesses the trace file from the beginning until it reaches a TIME statement that is greater than or equal to the specified TimeValue. VITASCOPE then refreshes the virtual scene with everything up-to-date i.e. as it would be at that TimeValue if the trace file were to be processed normally without jumping time.

**Syntax:**

```
TIMEJUMP <TimeValue>;
```

**Example:**

```
TIMEJUMP 2525.75;
```

**The TIMESKIP Statement**

The TIMESKIP statement skips over idle time during animation by instantly advancing the animation clock with the indicated amount of time units. When a TIMESKIP statement is encountered in a trace file, VITASCOPE’s clock is immediately advanced by the indicated amount of animation time units as long as there are no intermediate activities recorded in the trace file. If there are intermediate activities recorded in a trace file, animation time is skipped to the start of the first obstructing activity. The TIMESKIP statement is for skipping over idle times and should not be confused with TIMEJUMP.

**Syntax:**

```
TIMESKIP <TimeUnitsToSkip>;
```

**Example:**

```
TIMESKIP 200;
```
The VIEWRATIO Statement

The VIEWRATIO statement sets the speed at which the animation is viewed.

Syntax: VIEWRATIO <ViewRatioValue>;
Example: VIEWRATIO 6;

VITASCOPE animations can run at any desired animation speed. The animation speed, also known as the viewing ratio, represents the number of animated time units per second of viewing time. For instance, if the simulation model (and the animation trace) uses seconds as a unit of time, and the viewing ratio is changed to 6, then the VITASCOPE animation will run at a rate of six animated seconds per viewing second. Consequently, a modeled activity requiring one minute for completion in reality would be accomplished in 10 (i.e. 60/6) seconds in the animation. A viewing ratio of less than 1 can also be used to animate processes in slow motion.

Creating and Assembling Scene Objects

A VITASCOPE object is a dynamic scene entity that can be manipulated inside the virtual world. All simulation objects (e.g. construction resources – pieces of equipment, materials etc.) can be represented as VITASCOPE objects.

The CLASS Statement

The CLASS statement identifies the CAD file that contains the geometric representation for a particular class (i.e. type) of VITASCOPE objects. The path of the specified CAD file can be local, relative, or absolute. It is generally recommended that CAD files be contained in the same directory as the animation trace file. The format of the CAD files must be either VRML (*.wrl) or native Cosmo3D/OpenGl Optimizer CSB (*.csb). Most CAD modeling tools (e.g. 3dsmax, AutoCAD, MicroStation etc.) can export created 3D models in VRML format thus making their use in class definition straightforward.
Syntax:     CLASS <ClassName> <CADFileName>;
Example:    CLASS A30C VolvoA30C.wrl;
Example:    CLASS A30C 'C:\Models\VolvoA30C.wrl';

**The ORIENTCLASS Statement**

The ORIENTCLASS statement rotates the CAD model of a defined class about an indicated coordinate axis (X, Y, or Z) in order to align it appropriately with its local coordinate frame. This statement is only necessary if a class’ CAD model is inappropriately oriented in its local space, in which case the statement must be executed on a defined class before any objects are instantiated from it.

Syntax:     ORIENTCLASS <ClassName> <AboutAxis> <RotationAmount>;
Example:    ORIENTCLASS A30C X 90;
Example:    ORIENTCLASS A30C Y 180;
Example:    ORIENTCLASS A30C Z -90;

In particular, VITASCOPE expects a class’ CAD model to be aligned with its local X axis and resting on the X-Z plane. This desired orientation is graphically portrayed in the following figure.
The ORIENTCLASS statement can be used to transform inappropriately oriented CAD models to this desired configuration. Alternatively, the CAD model may be correctly oriented by editing it inside the native CAD modeling program. The latter approach is often more straightforward and is thus recommended over the ORIENTCLASS statement. In general, the ORIENTCLASS statement must only be used if access to a CAD drawing’s native editing tool is unavailable.

**The CREATE Statement**

The CREATE statement creates specific VITASCOPE scene objects by instantiating predefined classes. Several objects of the same type can be created from a class. For instance, a fleet of trucks (of the same model) can be created by instantiating several objects from the class that references that truck model’s CAD drawing.

Syntax: CREATE <ObjectName> <ClassName>;

Example: CREATE Truck1 A30C;
CREATE Truck2 A30C;

**The DESTROY Statement**

The DESTROY statement complements the CREATE statement and destroys an existing (i.e. created) VITASCOPE scene object and removes it from the scene permanently. A destroyed object cannot be manipulated without recreating it again.

Syntax: DESTROY <ObjectName>;

Example: DESTROY Truck1;

**The ATTACH Statement**

The ATTACH statement allows one VITASCOPE object to be attached to another at a specified attachment point. Once an object is attached to another, all manipulations (e.g. rotation, scaling) applied to the parent object transfer to the child object. However the reverse is not true, i.e. a child object can be manipulated independently without affecting
the parent object. The hierarchy of attachment can be as deep as required, i.e. one object could be attached to another, which in turn could be attached to a third object, and so on. In addition, more the one child objects can be attached to a single parent object. However, a child object can (at any given time) be attached to only one parent object.

The ATTACH statement is particularly useful in assembling pieces of articulated construction equipment (e.g. backhoes); where there is a general need to manipulate each component (e.g. boom, stick, bucket, etc.) independently in addition to having the whole assembly behave as a coherent unit. In a backhoe’s case, the bucket must be attached to the stick, the stick to the boom, the boom to the cabin, and the cabin to the crawlers, in order to define the articulated machine.

Syntax: ATTACH <ChildObjName> <ParentObjName> <AttachPoint>;

Example: ATTACH Cabin1 Crawlers1 (0,1,0);
Example: ATTACH Boom1 Cabin1 (3,1.4,0);
Example: ATTACH Shape8 Hook1 (0,-0.5,0);

The ATTACHNOSCALE Statement

The ATTACHNOSCALE statement is very similar to the ATTACH statement. The only difference is that when a child object is attached to a parent object using the ATTACHNOSCALE statement, any scaling applied to the parent object does not propagate down to the child object. This statement is particularly useful in situations where it is necessary to change the size of an object to portray an animated task while leaving the size of any attached children unchanged. A perfect example of this is a hook object of a crane that is attached to that crane’s cable object. The lowering and raising of a virtual crane’s hook is achieved by scaling the cable object in the vertical direction to increase or decrease its length. These significant changes in the size of the cable object should obviously not propagate to the child hook object. This can be achieved by using the ATTACHNOSCALE statement to attach the hook to the cable.
Syntax: \texttt{ATTACHNOSCALE \textless ChildObjName\textgreater \textless ParentObjName\textgreater <AttachPoint>;} \\
Example: \texttt{ATTACHNOSCALE Boom1 Tower1 (0,35,0);} \\
Example: \texttt{ATTACHNOSCALE Hook1 Cable1 (0,-1,0);} \\

\textbf{The DETACH Statement} \\
The DETACH statement complements the ATTACH and ATTACHNOSCALE statements. In general, any child object that has been previously attached to another object can be disconnected from that parent object using the DETACH statement. \\

Syntax: \texttt{DETACH \textless ChildObjectName\textgreater;} \\
Example: \texttt{DETACH Shape8;} \\

\textbf{The CHANGECLASS Statement} \\
As its name implies, the CHANGECLASS statement allows the class of a created object to be changed at animation runtime. For instance, we could define two classes (i.e. two CAD drawings) for a particular truck model. One of the CAD drawings could describe an empty truck while the other could describe a truck full of dirt (i.e. a loaded truck). A particular truck object could be instantiated with the empty truck class. Then, during animation, when a loader loads that truck with dirt, the class of that truck object could be instantaneously changed to the loaded truck class. The process could be reversed when the truck dumps the dirt at a dumpsite during animation. \\

Syntax: \texttt{CHANGECLASS \textless ObjectName\textgreater \textless NewClassName\textgreater;} \\
Example: \texttt{CHANGECLASS Truck1 LoadedA30C;} \\

\textbf{Defining Motion Paths} \\
VITASCOPE paths are 3D trajectories over which created scene objects can be moved. A VITASCOPE path consists of one or more, straight line segments that define an ordered
set. A path is defined by specifying a set of 3D coordinates. Since a path must have at least 1 segment, a minimum of 2 coordinates are required to define a path. VITASCOPE paths define basic, piecewise-linear motion trajectories. The PathFinder add-on for VITASCOPE allows the definition of complex, arbitrarily-shaped, curved motion path trajectories. Those statements are described in Appendix K.

**The PATH Statement**

The PATH statement defines a VITASCOPE path by constructing connected linear segments that pass through the indicated 3D coordinates. Paths defined by the PATH statement are directional. In other words, objects that are moved on trajectories defined by the PATH statement automatically point in the direction of motion. In addition, VITASCOPE paths are accumulating i.e. VITASCOPE will temporarily stop a traveling object before the end of a path if it is blocked by another object. When the blocking object at the end of the path moves away (e.g. travels on a connected subsequent path), the stopped object resumes its motion until it reaches the end of the path or is blocked again. If that was not the case, then multiple objects accumulating at the end of a path would come to rest “within” each other, which never occurs with physical objects.

Syntax: \[ \text{PATH <PathName> <Points>;} \]

Example: \[ \text{PATH LoadToDump '}(10,5,10)' '}(10,10,-30)' '}(10,10,-130)' '}(90,5,-130)' \]

**The NONDIRECPATH Statement**

The NONDIRECPATH statement defines a VITASCOPE path by constructing connected linear segments that pass through the indicated 3D coordinates. Paths defined by the NONDIRECPATH statement are identical to those defined by the PATH statement except that they are non-directional. In other words, objects that are moved on trajectories defined by the NONDIRECPATH statement slide along the path instead of turning around on corners i.e. moving objects do not turn to face the direction of motion. Instead, objects retain their original configuration that was current when motion started.
Syntax: NONDIRECPATH <PathName> <Points>;
Example: NONDIRECPATH StorePallet '(10,0,10)' '(2,0,-30)';

Setting Dynamic Properties for Classes and Objects
VITASCOPE provides several statements to set the dynamic properties of classes and/or objects. These properties include the specification of the Rear Guide Point (RGP), the Fore Clearance, and the After Clearance. These properties affect how objects move on motion paths. An object’s RGP offset defines an optional rear guide point for that object. The defined RGP behaves as a second point of attachment (in addition to the object’s local origin) when the object moves along directional VITASCOPE paths. The RGP is always behind an object’s local origin; and always lies on a path (along with the local origin) when that object moves. This is graphically presented in the following figure.

When simulation objects accumulate one behind the other on VITASCOPE’s accumulating paths, they are prevented from bumping/running into each other if the objects’ have been previously assigned “clearance” values. VITASCOPE objects can be attributed with two clearance values, specified in linear units. These are the Fore Clearance and the After Clearance. In general, a moving object is stopped n units
(measured linearly along the path) behind the local origin of the obstructing object ahead of it. The value of n is equal to the sum of the leading object’s after clearance and the trailing object’s fore clearance. This is graphically presented in the following figure.

When objects having the same values of clearances move along a path, the allocation of an object’s total clearance into fore and after clearance does not make any difference so long as the total clearance is comfortably larger than the length of the object. For instance, in the figure above, it is irrelevant whether the total clearance is $7+4 = 11$ or $5+6 = 11$, as long as 11 is comfortably greater than the object’s length. The only time at which the allocation of the total clearance matters is when dissimilar VITASCOPE objects (e.g. models of different trucks) accumulate on a path.

**The SET CLASS…RGP Statement**

The SET CLASS…RGP statement sets the RGP offset for the specified class. Existing objects of the particular class are unaffected. All new objects instantiated from the class, however, will inherit the class’ new RGP offset.

**Syntax:**

```
SET CLASS <ClassName> RGP <Value>;
```

**Example:**

```
SET CLASS CAT777D RGP 5.1;
```

**The SET CLASS…FORECLEARANCE Statement**

The SET CLASS…FORECLEARANCE statement sets the fore clearance for the specified class. Existing objects of the particular class are unaffected. All new objects instantiated from the class, however, will inherit the class’ new fore clearance value.
Syntax: \[\text{SET CLASS} \ <\text{ClassName}> \ \text{FORECLEARANCE} \ <\text{Value}>;\]
Example: \[\text{SET CLASS CAT777D FORECLEARANCE 4.0;}\]

**The SET CLASS…AFTCLEARANCE Statement**

The SET CLASS…AFTCLEARANCE statement sets the after clearance for the specified class. Existing objects of the particular class are unaffected. All new objects instantiated from the class, however, will inherit the class’ new after clearance value.

Syntax: \[\text{SET CLASS} \ <\text{ClassName}> \ \text{AFTCLEARANCE} \ <\text{Value}>;\]
Example: \[\text{SET CLASS CAT777D AFTCLEARANCE 7.0;}\]

**The SET OBJECT…RGP Statement**

The SET OBJECT…RGP statement sets the RGP offset for the specified object. This new value overrides the RGP offset inherited by the object from its class at the time it was created.

Syntax: \[\text{SET OBJECT} \ <\text{ObjectName}> \ \text{RGP} \ <\text{Value}>;\]
Example: \[\text{SET OBJECT Truck3 RGP 5.4;}\]

**The SET OBJECT…FORECLEARANCE Statement**

The SET OBJECT…FORECLEARANCE statement sets the fore clearance for the specified object. This new value overrides the fore clearance value inherited by the object from its class at the time it was created.

Syntax: \[\text{SET OBJECT} \ <\text{ObjectName}> \ \text{FORECLEARANCE} \ <\text{Value}>;\]
Example: \[\text{SET OBJECT Truck3 FORECLEARANCE 3.8;}\]
The SET OBJECT...AFTCLEARANCE Statement

The SET OBJECT...AFTCLEARANCE statement sets the after clearance for the specified object. This new value overrides the after clearance value inherited by the object from its class at the time it was created.

Syntax: SET OBJECT <ObjectName> AFTCLEARANCE <Value>;
Example: SET OBJECT Truck3 AFTCLEARANCE 6.8;

Placing Objects in the Scene

VITASCOPE objects created using the CREATE statement must be placed in the scene in order to be visible and before they can be manipulated (e.g. moved on paths). The PLACE statement allows a created VITASCOPE object to be introduced in the virtual world.

The PLACE…AT Statement

The PLACE…AT statement is used to place a created object at a desired 3D position.

Syntax: PLACE <ObjectName> AT <PlacePoint>;
Example: PLACE Truck1 AT '(10,0,10)';

The PLACE…ON…[AT] Statement

The PLACE…ON statement is used to place a created object at the beginning of a defined motion path. The optional [AT] statement extension can be used to place the object at any desired downstream distance along the path.

Syntax: PLACE <ObjectName> ON <PathName> [AT] <Distance>;
Example: PLACE Truck1 ON LoadToDump;
Example: PLACE Truck2 ON LoadToDump AT 15;
The HORIZORIENT Statement
The HORIZORIENT statement is used to explicitly change the horizontal rotation (i.e. yaw) of an object after it has been placed. This is necessary if the object’s default yaw (0 degrees) at the time of placement is not appropriate.

Syntax:          HORIZORIENT <ObjectName> <RotationValue>;
Example:         HORIZORIENT Shape8 180;

The VERTORIENT Statement
The VERTORIENT statement is used to explicitly change the vertical rotation (i.e. pitch) of an object after it has been placed. This is necessary if the object’s default pitch (0 degrees) at the time of placement is not appropriate.

Syntax:          VERTORIENT <ObjectName> <RotationValue>;
Example:         VERTORIENT Column9 90;

The SIDEORIENT Statement
The SIDEORIENT statement is used to explicitly change the side rotation (i.e. roll) of an object after it has been placed. This is necessary if the object’s default roll (0 degrees) at the time of placement is not appropriate.

Syntax:          SIDEORIENT <ObjectName> <RotationValue>;
Example:         SIDEORIENT Crawlers1 32;

Moving Objects in the Scene
VITASCOPE objects can move in guided or unguided motion. Guided motion always takes place along a defined path. Such motion is possible when the simulation objects move on fixed, well-described paths that can be pre-defined. Unguided motion, on the other hand, occurs between two 3D points and is useful in defining random motion that
cannot be defined a-priori. In VITASCOPE, the MOVE and MOVESPEED statements describe guided motion whereas the SLIDE and TGTSLIDE statements support unguided object motion.

The MOVE Statement
A VITASCOPE object that has been placed in the scene can be moved along any defined motion path. During placement, an object should obviously be placed on the first path it will be moved on i.e. at the point it will start its first motion from. If not, then the object will instantaneously jump to the beginning of a path when motion first starts.

In the case of the MOVE statement, the speed of motion is automatically calculated from the specified duration and the internally calculated path length. An object is smoothly moved on the indicated path at this constant average speed. The PathFinder add-on for VITASCOPE allows objects to be moved on paths with arbitrary velocity profiles (instead on constant average speeds as done by the MOVE statement). Those statements are described in Appendix K.

Syntax: \[\text{MOVE} \ <\text{ObjectName}> \ <\text{PathName}> \ <\text{TravelDuration}>;\]
Example: \[\text{MOVE Truck1 LoadToDump 225;}\]

The MOVESPEED Statement
The MOVESPEED statement moves a VITASCOPE object on an indicated path with the explicitly specified constant motion speed. The duration of the motion is implicitly computed from the indicated speed and the path’s internally calculated length.

Syntax: \[\text{MOVESPEED} \ <\text{ObjectName}> \ <\text{PathName}> \ <\text{TravelSpeed}>;\]
Example: \[\text{MOVESPEED Truck1 LoadToDump 10;}\]

The SLIDE Statement
The SLIDE statement causes an object to move in a straight line between its current position and the point defined by adding the indicated translation value to that position. The time required to accomplish this motion is indicated by the specified travel duration.
Syntax: SLIDE <ObjectName> <TranslationValue> <TravelDuration>;
Example: SLIDE Pallet21 '(5,0,0)' 4.43;

The TGTSIDE Statement
The TGTSIDE statement causes an object to move in a straight line to the point specified as the target position. The object thus moves in a straight line from its current position to the explicitly specified target 3D position. The time required to accomplish this motion is indicated by the specified travel duration.

Syntax: TGTSIDE <ObjectName> <TargetPosition> <TravelDuration>;
Example: TGTSIDE Pallet21 '(55,0,18)' 4.43;

Advanced Object Manipulation
VITASCOPE’s advanced object manipulation statements allow simulation objects to be rotated and scaled to describe their accurate dynamic behavior.

The ROTATE Statement
The ROTATE statement causes an object to rotate in the specified plane of rotation by the indicated rotation amount specified in degrees. The rotation occurs about the object’s local origin and is accomplished in the indicated time duration.

Syntax: ROTATE <ObjectName> <RotationPlane> <RotationAmount> <RotateDuration>;
Example: ROTATE Cabin1 HOR 45 12.20;
Example: ROTATE Boom1 VERT -50 15.10;
Example: ROTATE TBMHead1 SIDE 360 120;
In particular, rotation in the horizontal plane changes an object’s yaw, rotation in the vertical plane changes the pitch, and rotation in the side plane manipulates an object’s roll. This is graphically portrayed in the figure below.

In general, clockwise rotation is negative and anticlockwise rotation is positive. For instance, a pair of similarly aligned objects rotated in any plane by 45 degrees and -315 degrees respectively would end up in the same configuration, albeit rotating in opposite directions. The ROTATE statement is particularly useful in describing the motion of articulated pieces of construction equipment such as backhoes. In a backhoe’s case, for instance, the cabin is rotated in the horizontal plane, and the boom, stick, and bucket are rotated in the vertical plane to describe its accurate motion.

The TGTROTATE Statement

The TGTROTATE statement is very similar to the ROTATE statement. The only difference is that in the TGTROTATE statement, an absolute target rotation value (i.e. heading) is explicitly specified instead of an incremental rotation amount that is specified as a ROTATE statement’s argument. For instance, for an object whose rotation (in any plane) is 45 degrees, a ROTATE statement with 45 degrees incremental rotation is equivalent to a TGTROTATE statement with 90 degrees target rotation. The TGTROTATE statement is often useful in cases when the desired configuration of an object is known, but its current configuration is indeterminate.
Syntax: TGTROTATE <ObjectName> <RotationPlane>
       <TargetRotation> <RotateDuration>;

Example:   TGTROTATE Cabin1 HOR 90 12.20;
Example:   TGTROTATE Boom1 VERT 20 15.10;
Example:   TGTROTATE TBMHead1 SIDE 180 60;

**The SCALE Statement**

The SCALE statement allows the size of an object to be expanded or shrunk in any required axial direction. The desired change in size is achieved in the indicated time duration. This statement is particularly useful in creating objects with different sizes. For instance, 2 trucks of different sizes (but similar shape) could be created from the same class. The trucks, after instantiation (i.e. creation), could be scaled appropriately to portray their relative sizes. Scaling is also useful in describing motion of several common pieces of construction equipment. For instance, the portrayal of a crane lowering and raising its hook can be achieved by scaling the crane’s cable (to which the hook is attached) appropriately in the vertical (Y) direction.

Syntax:   SCALE <ObjectName> <ScaleFactor> <ScaleDuration>;
Example:  SCALE Cable1 (0,35,0) 23.2;

**The TGTSCALE Statement**

The TGTSCALE statement is very similar to the SCALE statement. The only difference is that instead of incremental axial scaling values, the desired target scales are explicitly indicated. For instance, for an object whose current scale is (1,1,1), a SCALE statement with (1,1,1) incremental scaling factors is equivalent to a TGTSCALE statement with (2,2,2) as the desired target scale. The TGTSCALE statement is often useful in cases when the desired configuration of an object is known, but its current configuration is indeterminate. An example of this is the case when a crane hook’s desired position is known, but it current height cannot be determined (i.e. accessed).
Syntax: TGTSCALE <ObjectName> <TargetScaleFactor> <ScaleDuration>;
Example: TGTSCALE Cable1 (1,40,1) 23.2;

Displaying Dynamic Numerical Statistics in Animations

VITASCOPE provides several statements to display numerical statistics in animation viewports in the form of dynamic text strings. In particular, two types of statistics are allowed; they are Object Statistics and Operation Statistics. Object statistics are dynamic strings of text that can be attached to any VITASCOPE object. Once attached, the dynamic text strings travel with the objects when the objects move. At all times, the strings face the viewer so that they can be read (regardless of the objects’ rotations and/or the user’s viewpoint). Object statistics are particularly useful to display the dynamic numerical properties of the objects’ they are attached to.

Operation statistics, on the other hand, are dynamic text strings that appear sequentially in another statistics-only viewport (i.e. window) during animation. Alternatively, the active operations statistics strings can float over the depicted 3D graphics in animation viewport(s). These text strings are more suitable to describing the dynamic numerical statistics that apply to the animated operation as a whole.

The OBJECTSTAT Statement

The OBJECTSTAT statement instantiates a dynamic object statistic string. The initial content of the string is set to the indicated InitialString.

Syntax: OBJECTSTAT <ObjectStatName> <InitialString>;
Example: OBJECTSTAT TruckFuel1 'Fuel Remaining: 52.1g';

The ATTACHSTAT Statement

The ATTACHSTAT statement attaches an instantiated object statistic string to a created VITASCOPE scene object at the specified attachment point.
Syntax: ATTACHSTAT  <ObjectStatName> <ParentObjName>  
                <AttachPoint>;
Example: ATTACHSTAT TruckFuel1 Truck1 (0,5,0);

The UPDATEOBJSTAT Statement
The UPDATEOBJSTAT statement updates the text string contained in an existing object statistic.

Syntax: UPDATEOBJSTAT <ObjectStatName> <NewString>;
Example: UPDATEOBJSTAT TruckFuel1 'Fuel Remaining: 45.3g';

The OPERATIONSTAT Statement
The OPERATIONSTAT statement instantiates a dynamic operation statistic string. The initial content of the string is set to the indicated InitialString.

Syntax: OPERATIONSTAT <OperStatName> <InitialString>;
Example: OPERATIONSTAT ConcVol 'Vol. Conc. Placed: 42M3';

The UPDATEOPERSTAT Statement
The UPDATEOPERSTAT statement updates the text string contained in an existing operation statistic.

Syntax: UPDATEOPERSTAT <OperStatName> <NewString>;

Special System Statements
The VITASCOPE language consists of several other special system statements. These statements are described in the following subsections.
The LOADADDON Statement

The LOADADDON statement dynamically loads VITASCOPE add-on (extension) modules. The syntax of this VITASCOPE statement is straightforward and consists of the name of the add-on module to be loaded preceded by the LOADADDON keyword as indicated below:

Syntax: LOADADDON <AddOnName>;
Example: LOADADDON KineMach;

When VITASCOPE encounters such a statement in a trace file, it attempts to locate the physical computer module that contains the add-on indicated in the argument. In particular, VITASCOPE searches the host computer for a dynamically linked library (DLL) named AddOnName.dll (e.g. KineMach.dll) on MS Windows or for a dynamically shared object (DSO) named libAddOnName.so (e.g. libKineMach.so) on SGI IRIX machines. This scheme conforms to the physical module naming conventions of each computing platform.

The SCHEDULE Statement

The SCHEDULE statement is used to schedule execution of any other VITASCOPE statements at future animation times. When VITASCOPE encounters a SCHEDULE statement, it reads and stores the complete statement that the command wants to schedule for future execution and also records the future execution time. Then, when that animation time is reached or exceeded, the scheduled statement(s) are executed as though they were encountered in the trace file at that time.

Syntax: SCHEDULE <AnimationStatement> <ExecutionTime>;
Example: SCHEDULE 'CREATE Truck1 CAT777D' 25;
The trace file segment,

```
TIME 0;
SCHEDULE 'CREATE Truck1 CAT777D' 25;
```

is thus functionally equivalent to,

```
TIME 25;
CREATE Truck1 CAT777D;
```

**The END Statement**
The END statement terminates all processing of an animation trace file.

Syntax: `END;`
Example: `END;`

**The “/” Operator**
The “/” operator is used to include comments in VITACSOPE animation trace files. In particular, VITASCOPE ignores all lines that begin with “/” during trace file processing.

Syntax: `/ <Comment>`
Example: `/ This is a comment in VITASCOPE`

**Controlling the User’s Viewpoint**
The VITACSOPE language provides several statements that help control the user’s viewpoint during animation. These statements complement the graphical user interface presented by the VITASCOPE virtual environment (VE) application. In particular, the user can control the camera using the keyboard and the mouse during animation in the VITASCOPE VE application. Those facilities are described in Appendix C.
The VIEWPOINT Statement

The VIEWPOINT statement is used to define vantage viewpoints from which to view the animation. The statement describes a viewpoint in 3 parameters: the first is the position of the camera; the second is the axis of rotation about which the camera is oriented; and the final parameter is the amount of orientation (i.e. rotation) itself. Any number of viewpoints can be defined in a trace file. When VITASCOPE encounters a VIEWPOINT statement, it adds the viewpoint to a stored circular array of viewpoints.

During animation, users can cycle through such predefined viewpoints using the F11 and F12 keys in any 3D viewport. In order to identify the parameters of vantage viewpoints, users can navigate to desired locations (using the mouse and/or keyboard) and then press the “P” key to print the camera’s current configuration to the output console. That information can then be used to define viewpoints.

Syntax: \[ \text{VIEWPOINT} \quad \text{<Name>} \quad \text{<Position>} \quad \text{<RotationAxis>} \quad \text{<RotationAmt>}; \]

Example: \[ \text{VIEWPOINT V1 '}(150,136,145)' '}(0,-1,0)' 0.25; \]

The ATTACHCAMERATO Statement

The ATTACHCAMERATO statement is used to mount the camera of a particular 3D viewport on an existing VITASCOPE object at a specified attachment point. Viewports are numbered starting at 0, i.e. the first 3D window created is channel number 0, the second is channel number 1, and so on. Once attached to an object, the camera automatically travels with that object and can be manipulated in that object’s local space. This for instance allows the user to ride in the cabin of a particular truck object in an animation and look around the virtual scene from the perspective of a real truck’s driver.

Syntax: \[ \text{ATTACHCAMERATO} \quad \text{<ChannelNum>} \quad \text{<ObjectName>} \quad \text{<AttachPoint>}; \]

Example: \[ \text{ATTACHCAMERATO 1 Truck1 (0,2,0);} \]
The DETACHCAMERA Statement

The DETACHCAMERA statement complements the ATTACHCAMERATO statement. The statement is used to detach the camera of a particular viewport from any object it might be attached to.

Syntax: DETACHCAMERA <ChannelNum>;
Example: DETACHCAMERA 1;
Appendix C

VITASCOPE Virtual Environment Application Reference

This appendix describes the VITASCOPE Virtual Environment (VE) application that processes animation trace files written in the VITASCOPE animation language and graphically portrays the communicated processes by instantiating and manipulating 3D CAD models of simulation objects (i.e. resources). The VITASCOPE VE application is implemented as a kernel that can serve several clients. The simplest of such clients – a command line driver – has also been implemented to complement the kernel. The kernel is implemented as a dynamic link library (Vitascope.dll) on MS Windows and as a dynamically shared object (libVitascope.so) on SGI IRIX. The command line driver is an executable named VitascopeCLD.exe on both platforms.

Valid animation trace files must have an extension VTF (e.g. Trace1.VTF). A trace file is submitted for processing by invoking the command line driver with the file name as the first argument. Optional arguments to the command line driver include the number and type of viewports (i.e. windows) to open for viewing the animated scene. The VITASCOPE VE application implements a multi-channel environment, i.e. the animated scene can be viewed simultaneously in multiple viewports each having independent navigation capability. This allows users, for instance, to view the same animated process from two different viewpoints in two separate windows at the same time.

VITASCOPE implements 3 types of viewports: 1) scene viewports, which are general animation windows that allow users to freely navigate in the 3D virtual environment; 2) plan viewports, which present users with an overhead plan view of the animated scene; and 3) statistic viewports, which exclusively display any instantiated operation statistics text strings. The operations statistics strings can also be overlaid in the other two types of viewports (i.e. scene and plan).
The general structure of the command line call is:

\[\text{VitascopeCLD.exe <filename> [-scene <num>] [-plan <num>] [-stats <num>]}\]

Examples of typical command line calls include:

\[\text{VitascopeCLD.exe Trace1.VTF}\]
\[\text{VitascopeCLD.exe Trace1.VTF -scene 2 -plan 1 -stats 1}\]
\[\text{VitascopeCLD.exe Trace1.VTF -scene 3 -plan 1}\]

In the first call, a single scene viewport is created by default. Each of the other 2 calls presented above create 4 windows of the types specified for viewing the animation.

**Performing User Interaction Tasks**

A user can interact with an animation by using the keyboard or the mouse in any of the instantiated animation windows. The command line driver conducts all required input and output through the command window. The following tables present the key and/or mouse commands for the supported user interaction tasks. Some of the tasks require interaction with the command window since user input is required to continue.

<table>
<thead>
<tr>
<th>User Interaction Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key</strong></td>
</tr>
<tr>
<td>Ctrl + S</td>
</tr>
<tr>
<td>Ctrl + P</td>
</tr>
<tr>
<td>Ctrl + V</td>
</tr>
<tr>
<td>Ctrl + T</td>
</tr>
</tbody>
</table>
Selecting/Picking Scene Objects

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + Any Mouse Button</td>
<td>Toggle the display of the selected object’s property string. The property string is displayed alongside the object and travels with the object if it moves.</td>
</tr>
<tr>
<td>Shift + Any Mouse Button</td>
<td>Displays the last selected object’s property string in the upper left corner of the window. Clicking in a blank area with the shift key depressed clears the display.</td>
</tr>
</tbody>
</table>

Navigating in the 3D Virtual Environment

A user can freely navigate in any 3D viewport instantiated by the VITASCOPE VE application. In plan viewports, navigation is limited to panning, and zooming in and out; the overhead plan view (i.e. looking straight down) is maintained. In scene viewports, there are no restrictions whatsoever on the user’s movements. Statistics viewports merely display text strings and require no navigation capabilities. A user can navigate using the keyboard, the mouse, or a combination of the two. The implemented navigation techniques are very intuitive and game-like. The following tables outline the mouse and keyboard navigation techniques implemented by the VITASCOPE VE application. Several other runtime key commands are also enumerated in the second table.

### Mouse Controls

<table>
<thead>
<tr>
<th>In Walk/Fly Navigation Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Button</td>
<td>Move forward/backward, Turn left/right</td>
</tr>
<tr>
<td>Middle Button</td>
<td>Look up/down/left/right</td>
</tr>
<tr>
<td>Right Button</td>
<td>Pan</td>
</tr>
</tbody>
</table>
**In Examine Mode**

<table>
<thead>
<tr>
<th>Button</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Button</td>
<td>Spin object</td>
</tr>
<tr>
<td>Middle Button</td>
<td>Look up/down/left/right</td>
</tr>
<tr>
<td>Right Button</td>
<td>Pan</td>
</tr>
</tbody>
</table>

**Runtime Key Commands**

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Travel faster</td>
</tr>
<tr>
<td>9</td>
<td>Travel slower</td>
</tr>
<tr>
<td>r</td>
<td>Reset and force viewpoint to see whole scene</td>
</tr>
<tr>
<td>n</td>
<td>Toggle navigation mode (WALK/EXAMINE)</td>
</tr>
<tr>
<td>p</td>
<td>Print viewport camera configuration</td>
</tr>
<tr>
<td>s</td>
<td>Toggle operation statistics</td>
</tr>
<tr>
<td>Shift + s</td>
<td>Toggle rendering performance statistics</td>
</tr>
<tr>
<td>F11</td>
<td>Select previous viewpoint</td>
</tr>
<tr>
<td>F12</td>
<td>Select next viewpoint</td>
</tr>
<tr>
<td>[</td>
<td>Close Field of View (FOV) by 5 degrees</td>
</tr>
<tr>
<td>]</td>
<td>Open Field of View (FOV) by 5 degrees</td>
</tr>
<tr>
<td>Escape</td>
<td>Exit</td>
</tr>
<tr>
<td>Up Arrow</td>
<td>Move forward</td>
</tr>
<tr>
<td>Down Arrow</td>
<td>Move backward</td>
</tr>
<tr>
<td>Left Arrow</td>
<td>Look left</td>
</tr>
<tr>
<td>Right Arrow</td>
<td>Look right</td>
</tr>
<tr>
<td>Ctrl + Arrow Keys</td>
<td>Pan</td>
</tr>
<tr>
<td>Page Up</td>
<td>Look up</td>
</tr>
<tr>
<td>Page Down</td>
<td>Look down</td>
</tr>
<tr>
<td>Ctrl + Page Up</td>
<td>Move up</td>
</tr>
<tr>
<td>Ctrl + Page Down</td>
<td>Move down</td>
</tr>
<tr>
<td>Home</td>
<td>Restore upward position</td>
</tr>
</tbody>
</table>
Appendix D
Segment of a VITASCOPE Animation Trace File

/ Define all motion paths
PATH ConcreteTruckBackUpToUnLoad '(-0.74,3,-44.2)'
  '(2.64,3,-36.1)';

... Define all other trajectories

TIME 0;
/ Position and orient camera
VIEWPOINT View1 '(-47.83, 16.74, 112.48)’ ‘(0,1,0)’ 1.51;

/ Define all object classes (i.e. CAD files)
/ The mobile crane
CLASS CraneCrawler Crawler.wrl;
CLASS CraneBoom Boom.wrl;
CLASS CraneCabin Cabin.wrl;
CLASS CraneCable Cable.wrl;
CLASS CraneHook Hook.wrl;
... Define all other classes

/ Create, assemble, and place objects in scene
CREATE Crawler1 CraneCrawler;
CREATE Cabin1 CraneCabin;
ATTACH Cabin1 Crawler1 (0,2,0);
CREATE Boom1 CraneBoom;
ATTACH Boom1 Cabin1 (0,0,0);
CREATE Cable1 CraneCable;
ATTACH Cable1 Boom1 (0,55,0);
CREATE Hook1 CraneHook;
ATTACHNOSCALE Hook1 Cable1 (0,-1,0);
PLACE Crawler1 AT (-5,2.9,-15);
... Create, assemble, and place all other objects

... Static and dynamic time stamped events
TIME 8914.30;
TGTROTATE Cabin1 HOR 270.00 26.57;
TGTROTATE Cable1 VERT 10.47 26.57;
TGTROTATE Boom1 VERT -10.47 26.57;
TIME 8940.87;
TGTSCALE Cable1 (1,44.48,1) 25.52;
TGTSLIDE Hook1 (0,-44.48,0) 25.52;
TIME 8994.52;
ATTACH TremieFunnel Hook1 (0,-0.6,0);
TIME 8994.52;
SCALE Cable1 (0,-5.00,0) 26.86;
SLIDE Hook1 (0,5.00,0) 26.86;
TIME 9021.38;
TGTROTATE Cabin1 HOR 296.57 29.37;
TGTROTATE Cable1 VERT 11.73 29.37;
TGTROTATE Boom1 VERT -11.73 29.37;
TIME 9027.62;
MOVE Truck2 TruckEntersBarge 50.60;
TIME 9050.75;
TGTSCALE Cable1 (1,46.05,1) 26.66;
TGTSLIDE Hook1 (0,-46.05,0) 26.66;
TIME 9078.22;
ATTACH Truck2 Barge1 (16.92,3,-6.42);
TIME 9078.22;
MOVE Barge1 BargeBacksUp 109.29;
TIME 9104.79;
DETACH TremieFunnel;
PLACE TremieFunnel AT (0,12.15,-5);
TIME 9114.79;
SCALE Cable1 (0,-5.00,0) 25.87;
SLIDE Hook1 (0,5.00,0) 25.87;
TIME 9140.65;
TGTROTATE Cabin1 HOR 270.00 26.97;
TGTROTATE Cable1 VERT 10.47 26.97;
TGTROTATE Boom1 VERT -10.47 26.97;
TIME 9187.51;
MOVE Barge1 BargeTravelsToPier 583.19;
TIME 9187.51;
MOVE Barge2 EmptyBargeDocksAtShore 197.63;
TIME 9194.73;
TGTSCALE Cable1 (1,44.48,1) 27.68;
TGTSLIDE Hook1 (0,-44.48,0) 27.68;
TIME 9250.87;
ATTACH TremiePipe2 Hook1 (0,-0.6,0);
TIME 9250.87;
SCALE Cable1 (0,-6.66,0) 28.56;
SLIDE Hook1 (0,6.66,0) 28.56;
TIME 9308.00;
SCALE Cable1 (0,-6.34,0) 13.53;
SLIDE Hook1 (0,6.34,0) 13.53;
TIME 9308.00;
TGTROTATE Cabin1 HOR 257.07 27.06;
TGTROTATE Cable1 VERT 16.53 27.06;
TGTROTATE Boom1 VERT -16.53 27.06;
TIME 9335.06;
TGTSCALE Cable1 (1,38.27,1) 25.94;
TGTSLIDE Hook1 (0,-38.27,0) 25.94;
TIME 9385.14;
DETACH Truck3;
TIME 9385.14;
MOVE Truck3 ConcreteTruckBackUpToExit 5.63;
TIME 9388.80;
DETACH TremiePipe2;
PLACE TremiePipe2 AT (-8.5,18.81,0.25)
TIME 9390.77;
MOVE Truck3 ConcreteTruckExitBarge 40.85;
TIME 9398.80;
SCALE Cable1 (0,-5.00,0) 28.80;
SLIDE Hook1 (0,5.00,0) 28.80;
TIME 9427.60;
TGTROTATE Cabin1 HOR 270.00 29.49;
TGTROTATE Cable1 VERT 10.47 29.49;
TGTROTATE Boom1 VERT -10.47 29.49;
TIME 9431.61;
DETACH Truck4;

... More static and dynamic events
Appendix E
VITASCOPE Add-On Interface Reference

This appendix presents the VITASCOPE add-on interface and supplements the information presented in chapter 3. The function prototypes and constant definitions for use by C and C++ add-ons are presented. The add-on interface is defined in the header file Vitascope.h, which is directly reproduced below. Example source code for a complete working add-on designed using VITASCOPE’s add-on interface is presented in Appendix F.

Vitascope.h

ativoscope.h

Function prototypes and constant definitions to be used by C and C++ add-ons for the Vitascopy Visualization System.

ifndef __VITASCOPE_H__
define __VITASCOPE_H__

ifdef defined(WIN32)
  ifdef VITASCOPE_DLLBUILD //Building DLL, export symbols
    define DCV_DLLEXPORT __declspec(dllexport)
  define EXPIMP_TEMPLATE
  else // Not building DLL, import symbols
    define DCV_DLLIMPORT _declspec(dllexport)
  define EXPIMP TEMPLATE extern
  endif // VITASCOPE_DLLBUILD
else
  define DCV_DLLIMPORT
endif // WIN32

define DCV_UNDEFINED 0 // No valid return value
define DCV_DOMAIN 1 // Argument domain error
define DCV_SING 2 // Argument singularity
define DCV_OVERFLOW 3 // Overflow range error
define DCV_UNDERFLOW 4 // Underflow range error
```c
#define DCV_TLOSS 5 // Total loss of precision
#define DCV_PLOSS 6 // Partial loss of precision

#define MAX_STATEMENT_LENGTH 1024*4

// VITASCOPE header files
#include "dcvObjectClass.h"
#include "dcvObject.h"
#include "dcvPath.h"
#include "dcvViewer.h"

typedef int (*FPExtStatement)(const char* szExpression,
                             void* pData1,
                             void* pData2);

typedef int (*FPUpdateFunc)(void* puserData,
                            bool bIsJumpingTime);

typedef int (*FPDrawFunc)(csDrawAction* pcsDrawAction,
                        int nChannelNum,
                        void* puserData);

typedef int (*FPExtKeyHandler)(int nKey,
                        int nMod,
                        void* puserData);

// VITASCOPE kernel entry point. This and the next two
// functions should only be used by new client interfaces
// and not by general add-ons.
DCV_DLLEXPORT int dcvEntryPoint(const char* szInputFileName,
                        int nSceneViewers,
                        int nPlanViewers,
                        int nStatViewers);

DCV_DLLEXPORT int dcvStartAnimation();

DCV_DLLEXPORT int dcvStopAnimation();

// Register add-on (re)defined statement. The function
// returns the fully qualified name of the last
// registered statement (if any) with the same keyword. The
// form of the returned string is AddOnName::StatementName.
// In case the last registered statement is part of the
// core VITASCOPE language, the returned string is of the
// form VITA::StatementName. If no prior statement with the
// same keyword exists, a blank string is returned.
```
DCV_DLLEXPORT const char* dcvRegStatement(const char* szAlias,
    FPExtStatement pFunc, 
    void* pD1, void* pD2);

    // Register add-on update function, if any.
DCV_DLLEXPORT int dcvRegUpdateFunc(FPUpdateFunc pFunc,
    void* pUserData=NULL);

    // Register add-on draw function, if any.
DCV_DLLEXPORT int dcvRegDrawFunc(FPDrawFunc pFunc,
    void* pUserData=NULL);

    // Register add-on defined and/or instantiated viewers, 
    // if any.
DCV_DLLEXPORT int dcvRegViewer(dcvViewer* pAddOnCreatedViewer);

    // Retrieve a pointer to an existing viewer by supplying 
    // its channel number.
DCV_DLLEXPORT dcvViewer* dcvGetViewer(int nChannelNumber);

    // Register a new, add-on defined keyboard command.
DCV_DLLEXPORT int dcvRegKey(int key,
    FPExtKeyHandler pHandler,
    const char* szHelpMessage = "No Help Message",
    void* pUserData = NULL);

    // Retrieve current simulation time.
DCV_DLLEXPORT const double& dcvGetCurSimTime();

    // Execute any existing animation statement from within 
    // the add-on module.
DCV_DLLEXPORT int dcvExecuteStatement(const char* 
    szFullStatement);

    // Schedule any existing animation statement for execution 
    // at a future animation time from within the add-on 
    // module.
DCV_DLLEXPORT int dcvScheduleStatementExecution(const 
    double dEventTriggerTime, 
    const char* szFullStatement);

    // Retrieve a pointer to an existing VITASCOPE class by 
    // supplying its name.
DCV_DLLEXPORT dcvObjectClass* dcvGetObjectClass(const char* szClassName);

// Register a new, add-on defined VITASCOPE class.
DCV_DLLEXPORT int dcvRegObjectClass(const char* szNewClsName,
                                    dcvObjectClass* pNewObjCls);

// Retrieve a pointer to an existing VITASCOPE object by
// supplying its name.
DCV_DLLEXPORT dcvObject* dcvGetObject(const char* szObjectName);

// Register a new, add-on defined VITASCOPE object.
DCV_DLLEXPORT int dcvRegObject(const char* szNewObjName,
                               dcvObject* pNewObj);

// Retrieve a pointer to an existing VITASCOPE path by
// supplying its name.
DCV_DLLEXPORT dcvPath* dcvGetPath(const char* szPathName);

// Register a new, add-on defined VITASCOPE path.
DCV_DLLEXPORT int dcvRegPath(const char* szNewPathName,
                             dcvPath* pNewPath);

// Print a message to the standard output console.
DCV_DLLEXPORT void dcvPrintToStdOutput(const char* szText);

// Print a message to the standard error console.
DCV_DLLEXPORT void dcvPrintToStdError(const char* szText);

// Throw a computation error.
DCV_DLLEXPORT void dcvMathError(const char* szFunction,
                                 int nErrorType);

// Convert a text string to a double numerical value.
DCV_DLLEXPORT double dcvConvertStringToDouble(const char* szString);

// Convert double numerical value to a text string.
DCV_DLLEXPORT const char* dcvConvertDoubleToString(double dValue);

// Extract an object of class csVec3f (3D vector) from a
// text string of the form '(x,y,z)'.
DCV_DLLEXPORT csVec3f dcvGetcsVec3fFromToken(const char* szToken);
// Retrieve/extract the next token from an argument string.
#if defined(__cplusplus)
DCV_DLLEXPORT char* dcvExtractArgument(char*& szRemainingArguments);
#else
DCV_DLLEXPORT char* dcvExtractArgument(char** const szRemainingArguments);
#endif
#endif //__VITASCOPE_H__
Appendix F

KineMach Add-On for VITASCOPE Source Code

This appendix presents example source code for a complete working add-on designed using VITASCOPE’s add-on interface. In particular, portions of the source code that implements the KineMach add-on for VITASCOPE are provided. KineMach implements a generic tower crane, crawler-mounted crane, backhoe, and dumptruck. This appendix presents the complete implementation of the KineMach crawler-mounted lattice boom crane.

KineMach Class Hierarchy

The following diagram presents the KineMach class hierarchy in the Booch OOD notation.
KineMach Source Files

KineMachAddOn.cpp

// Vitascope KineMach Add-On
// By Vineet R. Kamat and Julio C. Martinez
// Copyright (C) 2002, Virginia Polytechnic Institute and
// State University

#include "Vitascope.h"
#include "dcvFunctors.h"
#include "dcvTowerCrane.h"
#include "dcvCrawlerCrane.h"
#include "dcvDumpTruck.h"
#include "dcvBackHoe.h"

// Working buffer for string manipulation
char szScratch[MAX_STATEMENT_LENGTH];

// Map to hold pointers to instantiated machines
std::map<std::string, dcvKineMachMachine*> glb_mpMachineNameToObjPtr;

// List to hold instances of dcvFunctors
std::list<dcvFunctor*> glb_ltdcvFunctors;

// The Add-On update function. This will be called each
// time Vitascope updates its scene objects.
int UpdateKineMachAddOn(void* puserData,
                         bool bIsJumpingTime);

// The function used to expose the methods of
int InvokeMethod(const char* szArguments,
                  void* p1, void* p2);

// Functions declarations for statements defined in this
// add-On.
int KMProcessTOWERCRANE(const char* szArguments,
                         void* p1, void* p2);
int KMProcessCRAWLERCRANE(const char* szArguments,
                          void* p1, void* p2);
int KMProcessDUMPTRUCK(const char* szArguments,
                        void* p1, void* p2);
int KMProcessBACKHOE(const char* szArguments, 
    void* p1, void* p2);

// Strings to hold the last active (if any) fully qualified 
// command names for statements defined in this add-on. In 
// case the statements are original (i.e. defined for the 
// first time in this add-on), then these strings are empty 
// (i.e. Vitascopc returns empty strings)
std::string glb_szPrevTowerCraneStatement;
std::string glb_szPrevCrawlerCraneStatement;
std::string glb_szPrevDumpTruckStatement;
std::string glb_szPrevBackHoeStatement;

// The Add-On initialization function. This must be 
// declared extern "C" so that the symbol name is not 
// mangled by the IRIX C++ compiler. On Win32, the name can 
// be exported in a DEF file.
extern "C" int VitaAddOnInit(const char* szTraceFileName) 
{
    cerr << endl << "Vitascope KineMach AddOn Version 
    1, 2, 0, 0" << endl;
    cerr << "By Vineet R. Kamat and Julio C. Martinez" 
        << endl;

    // Register the add-on’s update function.
    dcvRegUpdateFunc(&UpdateKineMachAddOn, NULL);

    // Register the add-on defined statements
    glb_szPrevTowerCraneStatement =
        dcvRegStatement("TOWERCRANE", 
                      &KMProcessTOWERCRANE, NULL, NULL);

    glb_szPrevCrawlerCraneStatement =
        dcvRegStatement("CRAWLERCRANE", 
                       &KMProcessCRAWLERCRANE, NULL, NULL);

    glb_szPrevDumpTruckStatement =
        dcvRegStatement("DUMPTRUCK", 
                        &KMProcessDUMPTRUCK, NULL, NULL);

    glb_szPrevBackHoeStatement =
        dcvRegStatement("BACKHOE", 
                        &KMProcessBACKHOE, NULL, NULL);

    return true;
}
The Add-On clean-up function. This must be declared extern "C" so that the symbol name is not mangled by the IRIX C++ compiler. On Win32, the name can be exported in a DEF file.

extern "C" int VitaAddOnExit(const char* szTraceFileName)
{
    // Delete all the KineMach objects created on the heap.
    std::map<std::string, dcvKineMachMachine*>::iterator
    myMapIterator;

    myMapIterator = glb_mpMachineNameToObjPtr.begin();
    while(myMapIterator != glb_mpMachineNameToObjPtr.end())
    {
        delete (*myMapIterator).second;
        glb_mpMachineNameToObjPtr.erase(myMapIterator++);
    }

    // Delete all the functors created on the heap while exporting methods
    std::list<dcvFunctor*>::iterator myListIterator;

    myListIterator = glb_ltdcvFunctors.begin();
    while(myListIterator != glb_ltdcvFunctors.end())
    {
        delete (*myListIterator);
        glb_ltdcvFunctors.erase(myListIterator++);
    }

    cerr << endl << "KineMach Add-On unloaded."
    return true;
}

int
UpdateKineMachAddOn(void* puserData, bool bIsJumpingTime)
{
    // Call the update method of each instantiated machine.

    std::map<std::string, dcvKineMachMachine*>::iterator
    myMapIterator;

    myMapIterator = glb_mpMachineNameToObjPtr.begin();
while (myMapIterator != glb_mpMachineNameToObjPtr.end())
{
    (*myMapIterator).second->
        UpdateMachineMotionEffects(bIsJumpingTime);

    myMapIterator++;
}

return true;

}

bool
KMMachineExists(std::string szMachineName)
{
    std::map<std::string, dcvKineMachMachine*>::iterator
        myMapIterator;

    myMapIterator =
        glb_mpMachineNameToObjPtr.find(szMachineName);

    if(myMapIterator != glb_mpMachineNameToObjPtr.end())
    {
        return true;
    }

    return false;
}

int
KMProcessTOWERCRANE(const char* szArguments,
    void* p1, void* p2)
{
    // TOWERCRANE <name> <height> <radius>;

    strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);

    char* szWork=szScratch;

    // Assert that there is no other statement by that
    // name already registered.

    if(glb_szPrevTowerCraneStatement.length())
    {
        char szErrorMessage[256];
sprintf(szErrorMessage, "KineMach cannot override the TOWERCRANE statement."
    " Please resolve the add-on ambiguity.");

    throw(szErrorMessage);
}

//Create a STL list of pointers to character strings. //Each string will hold one argument.
std::list<std::string> rgszTokens;

//Separate and store all the arguments in the argument string
while(strlen(szWork))
    rgszTokens.push_back(dcvExtractArgument(szWork));

int nArgsInList = rgszTokens.size();

if(3 != nArgsInList)
{
    throw("Incorrect number of arguments");
}

std::list<std::string>::iterator myListIterator =
    rgszTokens.begin();

std::string szNewCraneName = *myListIterator;
std::string szCraneHeight = *(++myListIterator);
std::string szCraneRadius = *(++myListIterator);

if(KMMachineExists(szNewCraneName))
{
    char szErrorMessage[128];
    sprintf(szErrorMessage, "Machine %s already exists. Attempted redefinition",
        szNewCraneName.c_str());

    cerr << szErrorMessage << endl;
    throw(szErrorMessage);
}

double dCraneHeight = atof(szCraneHeight.c_str());
double dCraneRadius = atof(szCraneRadius.c_str());
if(dCraneHeight <= 0 || dCraneRadius <= 0)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The Tower height or the
    Boom radius cannot be negative.");
    throw(szErrorMessage);
}

// Create a new tower crane object and store it in
// the map
dcvTowerCrane* pdcvTowerCrane = new
dcvTowerCrane(szNewCraneName, dCraneHeight,
dCraneRadius);

glb_mpMachineNameToObjPtr[szNewCraneName] =
pdcvTowerCrane;

// Register the methods for this object. Functors
// make this easy.

// Build the names of the predefined actions by
// appending the operation to the name of the Object.
// We will use functors to export the methods.
char szPredefVarName[128];
dcvAnyClassFunctor<dcvTowerCrane>*pdcvTowerCraneFunctor;

sprintf(szPredefVarName,"%s.PlaceAt",
    szNewCraneName.c_str());
pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::PlaceAt);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.HoistBy",
    szNewCraneName.c_str());
pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::HoistBy);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.LowerBy",
    szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::LowerBy);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.HookAt",
    szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::HookAt);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.SwingBy",
    szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::SwingBy);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.SwingToward",
    szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::SwingToward);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.TrolleyTravelBy",
        szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::TrolleyTravelBy);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.TrolleyTravelTo",
        szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::TrolleyTravelTo);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.PutThatThere",
        szNewCraneName.c_str());

pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::PutThatThere);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvTowerCraneFunctor, NULL);
sprintf(szPredefVarName, "%s.PutThisThere",
    szNewCraneName.c_str());

    pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::PutThisThere);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.ChangeHeight",
    szNewCraneName.c_str());

    pdcvTowerCraneFunctor = new
dcvAnyClassFunctor<dcvTowerCrane>(pdcvTowerCrane,
dcvTowerCrane::ChangeHeight);

glb_ltdcvFunctors.push_back(pdcvTowerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvTowerCraneFunctor, NULL);

    return true;
}

int KMProcessCRAWLERCRANE(const char* szArguments, void* p1,
    void* p2)
{
    // CRAWLERCRANE <name> <crawler width> <boom length>;

    strncpy(szScratch, szArguments, MAX_STATEMENT_LENGTH);
    char* szWork = szScratch;

    // Assert that there is no other statement by that
    // name.
    if (glb_szPrevCrawlerCraneStatement.length())
    {
        char szErrorMessage[256];
        sprintf(szErrorMessage, "KineMach cannot override
            the CRAWLERCRANE statement.
            "Please resolve the
            add-on ambiguity."});
throw(szErrorMessage);
}

//Create a STL list of pointers to character strings. //Each string will hold one argument.
std::list<std::string> rgszTokens;

//Separate and store all the arguments in //the argument string
while(strlen(szWork))
    rgszTokens.push_back(dcvExtractArgument(szWork));

int nArgsInList = rgszTokens.size();

if(3 != nArgsInList)
{
    throw("Incorrect number of arguments.");
}

std::list<std::string>::iterator myListIterator =
    rgszTokens.begin();

std::string szNewCraneName = *myListIterator;
std::string szCrawlerWidth = *(++myListIterator);
std::string szBoomLength = *(++myListIterator);

if(KMMachineExists(szNewCraneName))
{
    char szErrorMessage[128];
    sprintf(szErrorMessage, "Machine %s already
    exists. Attempted redefinition",
    szNewCraneName.c_str());

    throw(szErrorMessage);
}

double dCrawlerWidth = atof(szCrawlerWidth.c_str());
double dBoomLength = atof(szBoomLength.c_str());

if(dCrawlerWidth <= 0 || dBoomLength <= 0)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The Crawler width or the
    Boom length cannot be negative.");
    throw(szErrorMessage);
}
// Create a new Crawler crane object and store
// it in the map
dcvCrawlerCrane* pdcvCrawlerCrane = new
dcvCrawlerCrane(szNewCraneName, dCrawlerWidth,
dBoomLength);

glb_mpMachineNameToObjPtr[szNewCraneName] =
pdcvCrawlerCrane;

// Register the methods for this object. Functors
// make this easy.

// Build the names of the predefined actions
// by appending the operation to the name of the
// Object. We will use functors to export the methods.
char szPredefVarName[128];
dcvAnyClassFunctor<dcvCrawlerCrane>
    *pdcvCrawlerCraneFunctor;
sprintf(szPredefVarName,"%s.PlaceAt",
    szNewCraneName.c_str());
pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane,
dcvCrawlerCrane::PlaceAt);

    glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvCrawlerCraneFunctor, NULL);

    sprintf(szPredefVarName,"%s.PlaceOn",
    szNewCraneName.c_str());
pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane,
dcvCrawlerCrane::PlaceOn);

    glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvCrawlerCraneFunctor, NULL);
sprintf(szPredefVarName, "%s.HoistBy", szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::HoistBy);

glob_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.LowerBy", szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::LowerBy);

glob_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.HookAt", szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::HookAt);

glob_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.SwingBy", szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::SwingBy);

glob_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName,"%s.SwingToward",
    szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane,
    dcvCrawlerCrane::SwingToward);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.LowerBoom",
    szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane,
    dcvCrawlerCrane::LowerBoom);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.RaiseBoom",
    szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane,
    dcvCrawlerCrane::RaiseBoom);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%sBoomAngle",
    szNewCraneName.c_str());

pdcvCrawlerCraneFunctor = new
dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::BoomAngle);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.BoomOver", szNewCraneName.c_str());
pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::BoomOver);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.PutThatThere", szNewCraneName.c_str());
pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::PutThatThere);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);

sprintf(szPredefVarName, "%s.PutThisThere", szNewCraneName.c_str());
pdcvCrawlerCraneFunctor = new dcvAnyClassFunctor<dcvCrawlerCrane>(pdcvCrawlerCrane, dcvCrawlerCrane::PutThisThere);

glb_ltdcvFunctors.push_back(pdcvCrawlerCraneFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvCrawlerCraneFunctor, NULL);
int KMProcessDUMPTRUCK(const char* szArguments,
    void* p1, void* p2)
{
    // DUMPTRUCK <name> <wheel base>;
strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);
char* szWork=szScratch;

// Assert that there is no other statement
// by that name.
if(glb_szPrevDumpTruckStatement.length())
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "KineMach cannot override
    the DUMPTRUCK statement."
            " Please resolve the
    add-on ambiguity.");

    throw(szErrorMessage);
}

//Create a STL list of pointers to character strings.
//Each string will hold one argument.
std::list<std::string> rgszTokens;

//Separate and store all the arguments in the
//argument string
while(strlen(szWork))
    rgszTokens.push_back(dcvExtractArgument(szWork));

int nArgsInList = rgszTokens.size();

if(2 != nArgsInList)
{
    throw("Incorrect number of arguments.");
}

std::list<std::string>::iterator myListIterator =
    rgszTokens.begin();

std::string szNewTruckName = *myListIterator;
std::string szWheelBase = *(++myListIterator);

if(KMMachineExists(szNewTruckName))
{
    char szErrorMessage[128];
    sprintf(szErrorMessage, "Machine %s already
    exists. Attempted redefinition",
            szNewTruckName.c_str());

    throw(szErrorMessage);
}
double dWheelBase = atof(szWheelBase.c_str());

if(dWheelBase <= 0) {
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The Dumptruck's wheel base cannot be negative.");
    throw(szErrorMessage);
}

// Create a new Crawler crane object and store it in the map
dcvDumpTruck* pdcvDumpTruck = new dcvDumpTruck(szNewTruckName, dWheelBase);

glb_mpMachineNameToObjPtr[szNewTruckName] = pdcvDumpTruck;

// Register the methods for this object. Functors make this easy.
// Build the names of the predefined actions by appending the operation to the name of the Object.
// We will use functors to export the methods.
char szPredefVarName[128];
dcvAnyClassFunctor<dcvDumpTruck>*pdcvDumpTruckFunctor;

sprintf(szPredefVarName,"%s.PlaceAt",
    szNewTruckName.c_str());

pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
    dcvDumpTruck::PlaceAt);

glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName,"%s.PlaceOn",
    szNewTruckName.c_str());

pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
    dcvDumpTruck::PlaceOn);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName,"%s.Travel",
    szNewTruckName.c_str());
pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
dcvDumpTruck::Travel);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName,"%s.Backup",
    szNewTruckName.c_str());
pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
dcvDumpTruck::Backup);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName,"%s.Dump",
    szNewTruckName.c_str());
pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
dcvDumpTruck::Dump);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName,"%s.DumpBedAngle",
    szNewTruckName.c_str());
pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
dcvDumpTruck::DumpBedAngle);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
( void*)pdcvDumpTruckFunctor, NULL);

sprintf(szPredefVarName, "%s.Destroy",
    szNewTruckName.c_str());
pdcvDumpTruckFunctor = new
dcvAnyClassFunctor<dcvDumpTruck>(pdcvDumpTruck,
    dcvDumpTruck::Destroy);
glb_ltdcvFunctors.push_back(pdcvDumpTruckFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
    ( void*)pdcvDumpTruckFunctor, NULL);

    return true;
}

int KMProcessBACKHOE(const char* szArguments, void* p1, void* p2)
{
    // BACKHOE <name> <crawler width>;
    strncpy(szScratch, szArguments, MAX_STATEMENT_LENGTH);
    char* szWork = szScratch;

    // Assert that there is no other statement
    // by that name.
    if (glb_szPrevBackHoeStatement.length())
    {
        char szErrorMessage[256];
        sprintf(szErrorMessage, "KineMach cannot override
            the BACKHOE statement."
            " Please resolve the
            add-on ambiguity."");

        throw(szErrorMessage);
    }

    //Create a STL list of pointers to character strings.
    //Each string will hold one argument.
    std::list<std::string> rgszTokens;

    //Separate and store all the arguments in the
    // argument string
    while (strlen(szWork))
        rgszTokens.push_back(dcvExtractArgument(szWork));

    int nArgsInList = rgszTokens.size();
if(2 != nArgsInList)
{
    throw("Incorrect number of arguments");
}

std::list<std::string>::iterator myListIterator =
    rgszTokens.begin();

std::string szNewBackHoeName = *myListIterator;
std::string szCrawlerWidth = *(++myListIterator);

if(KMMachineExists(szNewBackHoeName))
{
    char szErrorMessage[128];
    sprintf(szErrorMessage, "Machine %s already 
exists. Attempted redefinition", 
    szNewBackHoeName.c_str());

    throw(szErrorMessage);
}

double dCrawlerWidth = atof(szCrawlerWidth.c_str());

if(dCrawlerWidth <= 0)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The Crawler width cannot be negative.");
    throw(szErrorMessage);
}

// Create a new BackHoe object and store it in the map
dcvBackHoe* pdcvBackHoe = new
dcvBackHoe(szNewBackHoeName, dCrawlerWidth);

glb_mpMachineNameToObjPtr[szNewBackHoeName] =
    pdcvBackHoe;

// Register the methods for this object. Functors 
// make this easy.

// Build the names of the predefined actions by 
// appending the operation to the name of the Object. 
// We will use functors to export the methods.
char szPredefVarName[128];
dcvAnyClassFunctor<dcvBackHoe>
    *pdcvBackHoeFunctor;
sprintf(szPredefVarName, "%s.PlaceAt",
        szNewBackHoeName.c_str());

    pdcvBackHoeFunctor = new
        dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::PlaceAt);

    glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

    dcvRegStatement(szPredefVarName, &InvokeMethod,
        (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.PlaceOn",
        szNewBackHoeName.c_str());

    pdcvBackHoeFunctor = new
        dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::PlaceOn);

    glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

    dcvRegStatement(szPredefVarName, &InvokeMethod,
        (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.BoomAngle",
        szNewBackHoeName.c_str());

    pdcvBackHoeFunctor = new
        dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::BoomAngle);

    glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

    dcvRegStatement(szPredefVarName, &InvokeMethod,
        (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.StickAngle",
        szNewBackHoeName.c_str());

    pdcvBackHoeFunctor = new
        dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::StickAngle);
glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.BucketAngle",
        szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::BucketAngle);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.SwingBy",
        szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::SwingBy);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.SwingToward",
        szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
        dcvBackHoe::SwingToward);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.Travel",
        szNewBackHoeName.c_str());
pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
dcvBackHoe::Travel);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.Backup",
        szNewBackHoeName.c_str());
pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
dcvBackHoe::Backup);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.BucketAt",
        szNewBackHoeName.c_str());
pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
dcvBackHoe::BucketAt);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName, "%s.Dig",
        szNewBackHoeName.c_str());
pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
dcvBackHoe::Dig);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);
dcvRegStatement(szPredefVarName, &InvokeMethod,
(void*)pdcvBackHoeFunctor, NULL);
sprintf(szPredefVarName,"%s.DigAt",
    szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
    dcvBackHoe::DigAt);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.Dump",
    szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
    dcvBackHoe::Dump);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.DumpAt",
    szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
    dcvBackHoe::DumpAt);

glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod,
    (void*)pdcvBackHoeFunctor, NULL);

sprintf(szPredefVarName,"%s.DigHereDumpThere",
    szNewBackHoeName.c_str());

pdcvBackHoeFunctor = new
dcvAnyClassFunctor<dcvBackHoe>(pdcvBackHoe,
    dcvBackHoe::DigHereDumpThere);
glb_ltdcvFunctors.push_back(pdcvBackHoeFunctor);

dcvRegStatement(szPredefVarName, &InvokeMethod, 
(void*)pdcvBackHoeFunctor, NULL);

return true;
}

// The actual function that is exported to Vitascope to
// provide access to the methods in dcvKineMachMachine and
// derived objects.
int InvokeMethod(const char* szArguments, void* p1, void* p2)
{
    strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);
    char* szWork=szScratch;

    //Create a STL list of character strings.
    //Each string will hold one argument.
    std::list<std::string> rgszTokens;

    //Separate and store all the arguments in the
    //argument string
    while(strlen(szWork))
        rgszTokens.push_back(dcvExtractArgument(szWork));

    //The functor encapsulating the particular method
    //was disguised as the first void pointer. The second
    //void pointer is not used since the functor
    //encapsulates both the pertinent object and its
    //pertinent member function.
    dcvFunctor* pdcvFunctor = (dcvFunctor*)p1;

    //Simple call to the functor's execution function.
    return pdcvFunctor->Call(rgszTokens);
}

dcvFunctors.h
// dcvFunctors.h: Interface and implementation
// for the dcvFunctor class.

#ifndef __DCVFUNCTORS_H__
#define __DCVFUNCTORS_H__
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// Standard header files
#include <string>
#include <list>

// Abstract base class
class dcvFunctor
{
public:
    // Two possible functions to call member function.
    // Virtual causes derived classes will use a pointer
    // to an object and a pointer to a member function
    // to make the function call.

    // Call using operator
    virtual int operator()(std::list<std::string> szArgList) = 0;

    // Call using function
    virtual int Call(std::list<std::string> szArgList) = 0;
};

// Derived template class
template <class dcvAnyClass> class dcvAnyClassFunctor :
    public dcvFunctor
{
private:
    // Pointer to member function
    int (dcvAnyClass::*pt2MemFunc)(std::list<std::string> szArgList);

    // Pointer to object
    dcvAnyClass* pt2Object;

public:
    // Constructor - takes pointer to an object and
    // pointer to a member and stores them in two private
    // variables
    dcvAnyClassFunctor(dcvAnyClass* _pt2Object,
                       int (dcvAnyClass::*_pt2MemFunc)(std::list<std::string> szArgList))
    { pt2Object = _pt2Object; pt2MemFunc = _pt2MemFunc; };

354
// Override operator "()"
virtual int operator() (std::list<std::string> szArgList)
    {return (*pt2Object.*pt2MemFunc)(szArgList);};

// Override function "Call"
virtual int Call (std::list<std::string> szArgList)
    {return (*pt2Object.*pt2MemFunc)(szArgList);};
};

#endif // __DCVFUNCTORS_H__

dcvKineMachMachine.h

// dcvKineMachMachine.h: Interface for the
// dcvKineMachMachine class.

#ifndef __DCVKINEMACHMACHINE_H__
#define __DCVKINEMACHMACHINE_H__

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Vitascope.h"

class dcvKineMachMachine
{
public:
    dcvKineMachMachine (std::string szName);
    virtual ~dcvKineMachMachine();
    virtual int UpdateMachineMotionEffects (bool bIsJumpingTime){return true;}

protected:
    std::string m_szName;
    int CheckNumArguments (int nReqArgs, const std::list<std::string>& szArgList);
    csVec2f GetcsVec2fFromToken (const char* szToken, bool& bPlaceIn3D);
    csMatrix4f GetLocNonScaledTransformMatrix (dcvObject* pdcvObject);
    csMatrix4f GetGlbNonScaledTransformMatrix (dcvObject* pdcvObject);
};

#endif // __DCVKINEMACHMACHINE_H__
# include "dcvKineMachMachine.h"

dcvKineMachMachine::dcvKineMachMachine(std::string szName)
{
    m_szName = szName;
}

dcvKineMachMachine::~dcvKineMachMachine(){ }

int
dcvKineMachMachine::CheckNumArguments(int nReqArgs,
    const std::list<std::string>& szArgList)
{
  // Check if the correct number of arguments
  // are supplied.
  int nArgsInList = szArgList.size();
  
  if(nArgsInList != nReqArgs)
  {
    char szErrorMessage[128];
    sprintf(szErrorMessage, "Incorrect number of
    arguments passed.");
    throw(szErrorMessage);
  }

  return true;
}

csVec2f
dcvKineMachMachine::GetcsVec2fFromToken(const char* szToken,
    bool& bPlaceIn3D)
{
    float fPathPoint[3];
    char rdgchDelimit[] = " [\}
    char* szDigit;
    int k = 0;

    // While there are useful elements in szToken,
    // separate each of them into x,y
    k =0;
szDigit =
    strtok(const_cast<char*>(szToken), rgchDelimit);

while(szDigit != NULL)
{
    fPathPoint[k] = float(atof(szDigit));
    k++;

    szDigit = strtok(NULL, rgchDelimit);
}

if(3 == k)
{
    bPlaceIn3D = true;
    return csVec2f(fPathPoint[0], fPathPoint[1]);
}

if(2 != k)
{
    throw("Syntax error in defining coordinate. (X,Y)  
            or (X,Y,Z) expected.");
}

return csVec2f(fPathPoint[0], fPathPoint[1]);

}
pdcvObject->GetVertOrntTransform()->
    getMatrix(WorkingMat);

LocalMat.postMult(WorkingMat);
pdcvObject->GetObjectRoot()->getMatrix(WorkingMat);
LocalMat.postMult(WorkingMat);
return LocalMat;
}

csMatrix4f
dcvKineMachMachine::GetGlbNonScaledTransformMatrix
    (dcvObject* pdcvObject)
{
    // This function returns the matrix ThisTransMat as
    // the product of the matrix stack of all matrices in
    // the path from the root of the scene to this object.
    // Scaling matrices along the way are ignored since we
    // need each object's global transformation matrix
    // that must not include scaling factors

csMatrix4f ThisTransMat =
    GetLocNonScaledTransformMatrix(pdcvObject);

dcvObject* pParentObject = pdcvObject->
    GetMyParentObj();

    // As high as the hierarchy goes.
while(pParentObject != NULL)
{
    ThisTransMat.postMult
    (GetLocNonScaledTransformMatrix(pParentObject));

    pParentObject = pParentObject->GetMyParentObj();
}
return ThisTransMat;
}

dcvSlewingMachine.h
// dcvSlewingMachine.h: Interface for the
// dcvSlewingMachine class.
#ifndef __DCVSLEWINGMACHINE_H__
#define __DCVSLEWINGMACHINE_H__

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "dcvKineMachMachine.h"

class dcvSlewingMachine : public dcvKineMachMachine
{
public:
    dcvSlewingMachine(std::string szName);
    virtual ~dcvSlewingMachine();

    double GetHeading(double dXSrc, double dZSrc,
                       double dXDst, double dZDst);

    double GetLength(double dXSrc, double dZSrc,
                      double dXDst, double dZDst);
};

#endif // __DCVSLEWINGMACHINE_H__

dcvSlewingMachine.cpp

// dcvSlewingMachine.cpp: Implementation of the
dcvSlewingMachine class.

#include "dcvSlewingMachine.h"

dcvSlewingMachine::dcvSlewingMachine(std::string szName) : dcvKineMachMachine(szName)
{
}

dcvSlewingMachine::~dcvSlewingMachine(){ }

double
dcvSlewingMachine::GetHeading(double dXSrc, double dZSrc,
                                double dXDst, double dZDst)
{
    double x1 = dXSrc;
    double z1 = dZSrc;
    double x2 = dXDst;
    double z2 = dZDst;
double d_LengthOfSegment=0;
double d_CWAngle;

//Calculate length of the segment first
double DelX = fabs(x1-x2);
double DelZ = fabs(z1-z2);

d_LengthOfSegment = sqrt(pow(DelX,2)+pow(DelZ,2));

if(d_LengthOfSegment<=0)
{
    return 0;
}
else //Calculate the CW angle with +X axis
{
    //Determine the Quadrant of the Out Vector
    if(z2<=z1 && x2<=x1)//Quad 1
    {
        d_CWAngle =
        (M_PI-asin(DelZ/d_LengthOfSegment))
        *180/M_PI;
    }
    else if (z2>=z1 && x2<=x1)//Quad 2
    {
        d_CWAngle =
        (M_PI+asin(DelZ/d_LengthOfSegment))
        *180/M_PI;
    }
    else if (z2>=z1 && x2>=x1)//Quad 3
    {
        d_CWAngle =
        (2*M_PI-asin(DelZ/d_LengthOfSegment))
        *180/M_PI;
    }
    else if (z2<=z1 && x2>=x1)//Quad 4
    {
        d_CWAngle =
        (asin(DelZ/d_LengthOfSegment))
        *180/M_PI;
    }
    if(d_CWAngle>=360)d_CWAngle-=360;
// Return the Rotation in Cosmo3D's Terminology
return d_CWAngle;
}

double
dcvSlewingMachine::GetLength(double dXSrc, double dZSrc,
    double dXDst, double dZDst)
{
    double x1 = dXSrc;
    double z1 = dZSrc;
    double x2 = dXDst;
    double z2 = dZDst;
    double d_LengthOfSegment=0;

    // Calculate length of the segment first
    double DelX = fabs(x1-x2);
    double DelZ = fabs(z1-z2);
    d_LengthOfSegment = sqrt(pow(DelX,2)+pow(DelZ,2));
    return d_LengthOfSegment;
}

dcvCrawlerCrane.h

// dcvCrawlerCrane.h: Interface for the
// dcvCrawlerCrane class.

#ifndef __DCVCRAWLERCRANE_H__
#define __DCVCRAWLERCRANE_H__

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "dcvSlewingMachine.h"

class dcvCrawlerCrane : public dcvSlewingMachine
{
public:
    dcvCrawlerCrane(std::string szName,
        double dCrawlerWidth,
        double dBoomLength);

    virtual ~dcvCrawlerCrane();
}
// Exported Method Functions
int PlaceOn(std::list<std::string> szArgList);
int PlaceAt(std::list<std::string> szArgList);
int HoistBy(std::list<std::string> szArgList);
int LowerBy(std::list<std::string> szArgList);
int HookAt(std::list<std::string> szArgList);
int SwingBy(std::list<std::string> szArgList);
int SwingToward(std::list<std::string> szArgList);
int LowerBoom(std::list<std::string> szArgList);
int RaiseBoom(std::list<std::string> szArgList);
int BoomOver(std::list<std::string> szArgList);
int BoomAngle(std::list<std::string> szArgList);
int PutThatThere(std::list<std::string> szArgList);
int PutThisThere(std::list<std::string> szArgList);
int Travel(std::list<std::string> szArgList);
int Backup(std::list<std::string> szArgList);
int TetherLoad(std::list<std::string> szArgList);

private:
    double    m_dCrawlerWidth;
    double    m_dBoomLength;
    double    m_dBoomAttachHeight;
    double    m_dBoomAngle;
    double    m_dCableDropLength;
    double    m_dRGPDistance;

    bool     m_bHasJustBackedUp;

    std::string   m_szCrawlerName;
    std::string   m_szCabinName;
    std::string   m_szBoomName;
    std::string   m_szCableName;
    std::string   m_szHookName;
    std::string   m_szImagObjName;
    std::string   m_szScratch;
};

#endif // __DCVCRAWLERCRANE_H__

dcvCrawlerCrane.cpp

// dcvCrawlerCrane.cpp: Implementation of the
dcvCrawlerCrane class.

#include "dcvCrawlerCrane.h"
dcvCrawlerCrane::dcvCrawlerCrane(std::string szName, 
   double dCrawlerWidth, 
   double dBoomLength)
   : dcvSlewingMachine(szName)
{
   m_dCrawlerWidth = dCrawlerWidth;
   m_dBoomLength = dBoomLength;

   // Get the directory where Vitascope is installed so 
   // that the data directory can be located. The 
   // required CAD models are located in the data 
   // subdirectory
   std::string szVitaInstDir;

   // Get the value of the VITASCOPE_INSTALL_DIR 
   // environment variable
   szVitaInstDir = getenv("VITASCOPE_INSTALL_DIR");

   if(!szVitaInstDir.length())
   {
      char szErrorMessage[256];

      sprintf(szErrorMessage, "The environment variable 
               VITASCOPE_INSTALL_DIR has not been set." 
               " KineMach cannot locate 
               the required components.");

      throw(szErrorMessage);
   }

   #if defined(WIN32) // Windows
   std::string szDirSepSlash = "\\";
   #else // IRIX
   std::string szDirSepSlash = "/";
   #endif

   std::string szDataDirName = szVitaInstDir + 
   szDirSepSlash + "data";

   // Define classes when the first tower crane is 
   // instantiated. Construct the fully qualified 
   // filenames of the component csb files.
   char szScratch2[MAX_STATEMENT_LENGTH];
// Define classes when the first crawler crane is
// instantiated
if(NULL ==
dcvGetObjectClass("_KineMach_CrawlerCrane_Crawler_")
{
    sprintf(szScratch2, "CLASS
    _KineMach_CrawlerCrane_Crawler_
        '%s%s_CC Craw_.csb'",
        szDataDirName.c_str(),
        szDirSepSlash.c_str());

dcvExecuteStatement(szScratch2);

    sprintf(szScratch2, "CLASS
    _KineMach_CrawlerCrane_Cabin_
        '%s%s_CC Cabin_.csb'",
        szDataDirName.c_str(),
        szDirSepSlash.c_str());

dcvExecuteStatement(szScratch2);

    sprintf(szScratch2, "CLASS
    _KineMach_CrawlerCrane_Boom_
        '%s%s_CC Boom_.csb'",
        szDataDirName.c_str(),
        szDirSepSlash.c_str());

dcvExecuteStatement(szScratch2);

    sprintf(szScratch2, "CLASS
    _KineMach_CrawlerCrane_Cable_
        '%s%s_CC Cable_.csb'",
        szDataDirName.c_str(),
        szDirSepSlash.c_str());

dcvExecuteStatement(szScratch2);

    sprintf(szScratch2, "CLASS
    _KineMach_CrawlerCrane_Hook_
        '%s%s_CC Hook_.csb'",
        szDataDirName.c_str(),
        szDirSepSlash.c_str());

dcvExecuteStatement(szScratch2);
dcvExecuteStatement(szScratch2);
}

// Build the names of this crawler crane's components
m_szCrawlerName = "_" + szName +
"_KineMach_CrawlerCrane_Crawler_";
m_szCabinName = "_" + szName +
"_KineMach_CrawlerCrane_Cabin_";
m_szBoomName = "_" + szName +
"_KineMach_CrawlerCrane_Boom_";
m_szCableName = "_" + szName +
"_KineMach_CrawlerCrane_Cable_";
m_szHookName = "_" + szName +
"_KineMach_CrawlerCrane_Hook_"
;

// Create components of this crawler crane
m_szScratch = "CREATE " + m_szCrawlerName +
"_KineMach_CrawlerCrane_Crawler_"
;
dcvExecuteStatement(m_szScratch.c_str());

// The length of the crawler tracks is 9m and their
// width is 7m in the CAD model of the crawler tracks.
// We must scale or shrink the tracks and the cabin to
// get them to the desired size.
double dCrawlerAndCabinTgtScaleFac =
    m_dCrawlerWidth/7;

m_dRGPDistance = dCrawlerAndCabinTgtScaleFac*5;
sprintf(szScratch2, "SET OBJECT %s RGP %f",
    m_szCrawlerName.c_str(), m_dRGPDistance);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "SET OBJECT %s FORECLEARANCE %f",
    m_szCrawlerName.c_str(),
    4*dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "SET OBJECT %s AFTCLEARANCE %f",
    m_szCrawlerName.c_str(),
    7*dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

m_szScratch = "CREATE " + m_szCabinName + "_KineMach_CrawlerCrane_Cabin_";
dcvExecuteStatement(m_szScratch.c_str());

m_szScratch = "CREATE " + m_szBoomName + "_KineMach_CrawlerCrane_Boom_";
dcvExecuteStatement(m_szScratch.c_str());

m_szScratch = "CREATE " + m_szCableName + "_KineMach_CrawlerCrane_Cable_";
dcvExecuteStatement(m_szScratch.c_str());

m_szScratch = "CREATE " + m_szHookName + "_KineMach_CrawlerCrane_Hook_";
dcvExecuteStatement(m_szScratch.c_str());

// Scale the created components as required and then assemble the components

sprintf(szScratch2, "TGTSCALE %s '(%f,%f,%f)' 0", m_szCrawlerName.c_str(), dCrawlerAndCabinTgtScaleFac, dCrawlerAndCabinTgtScaleFac, dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTSCALE %s '(%f,%f,%f)' 0", m_szCabinName.c_str(), dCrawlerAndCabinTgtScaleFac, dCrawlerAndCabinTgtScaleFac, dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

// The length of the boom is 55m in the CAD model. We must scale or shrink the boom to get it to the desired length.
double dBoomLengthScaleFac = m_dBoomLength/55;
sprintf(szScratch2, "TGTSCALE %s '(%f,%f,%f)' 0",
    m_szBoomName.c_str(),
    dCrawlerAndCabinTgtScaleFac,
    dBoomLengthScaleFac,
    dCrawlerAndCabinTgtScaleFac);

dcvExecuteStatement(szScratch2);

// Assemble the components now
m_dBoomAttachHeight = 2*dCrawlerAndCabinTgtScaleFac;

sprintf(szScratch2, "ATTACHNOSCALE %s %s '(0,%f,0)'",
    m_szCabinName.c_str(),
    m_szCrawlerName.c_str(),
    m_dBoomAttachHeight);

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "ATTACHNOSCALE %s %s '(0,0,0)'",
    m_szBoomName.c_str(), m_szCabinName.c_str());

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "ATTACHNOSCALE %s %s '(0,%f,0)'",
    m_szCableName.c_str(), m_szBoomName.c_str(),
    55*dBoomLengthScaleFac);

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "ATTACHNOSCALE %s %s '(0,-1,0)'",
    m_szHookName.c_str(), m_szCableName.c_str());

dcvExecuteStatement(szScratch2);

// Set the initial boom angle to be 80 degrees
m_dBoomAngle = 80;

sprintf(szScratch2, "TGTROTATE %s VERT %f 0",
    m_szCableName.c_str(), 90-m_dBoomAngle);

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTROTATE %s VERT -%f 0",
    m_szBoomName.c_str(), 90-m_dBoomAngle);

dcvExecuteStatement(szScratch2);
// Set the initial cable drop to 5m
m_dCableDropLength = 5;

sprintf(szScratch2, "TGTSCALE %s (1,5,1) 0",
    m_szCableName.c_str());
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTSLIDE %s (0,-5,0) 0",
    m_szHookName.c_str());
dcvExecuteStatement(szScratch2);

// Build a new geometryless imaginary dcvObject and
// attach it to the backhoe at its RGP. This object
// will be used when the backhoe backs up.
// Construct a hard to guess name for this object
m_szImagObjName = "__" + m_szName +
    "_Imag_Obj_At_RGP__";

// VITA_INVISIBLE is a special class defined by
// VITASCOPE. All objects created with this class
// are not visible in the animation.
sprintf(szScratch2, "CREATE %s VITA_INVISIBLE",
    m_szImagObjName.c_str());
dcvExecuteStatement(szScratch2);

// Set this object's properties to be the same as the
// crane.
sprintf(szScratch2, "SET OBJECT %s RGP %f",
    m_szImagObjName.c_str(),
    m_dRGPDistance);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "SET OBJECT %s FORECLEARANCE %f",
    m_szImagObjName.c_str(),
    4*dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "SET OBJECT %s AFTCLEARANCE %f",
    m_szImagObjName.c_str(),
    7*dCrawlerAndCabinTgtScaleFac);
dcvExecuteStatement(szScratch2);

// Turn it around to face the opposite direction.
sprintf(szScratch2, "HORIZORIENT %s 180",
    m_szImagObjName.c_str());
dcvExecuteStatement(szScratch2);

// Now attach this object to this truck at its RGP
sprintf(szScratch2, "ATTACHNOSCALE %s %s '(-%f,0,0)')",
    m_szImagObjName.c_str(),
    m_szCrawlerName.c_str(),
    m_dRGPDistance);
dcvExecuteStatement(szScratch2);

m_bHasJustBackedUp = false;
}

dcvCrawlerCrane::~dcvCrawlerCrane(){ }

int dcvCrawlerCrane::PlaceOn(std::list<std::string> szArgList)
{
    // CraneName.PlaceOn <path name>

    // Check the number of arguments
    CheckNumArguments(1, szArgList);

    std::string szPlacePath = szArgList.front();

    // Construct the PLACE statement to be executed.
    m_szScratch = "PLACE " + m_szCrawlerName + " ON " +
        szPlacePath;
    dcvExecuteStatement(m_szScratch.c_str());

    return true;
}

int dcvCrawlerCrane::PlaceAt(std::list<std::string> szArgList)
{
    // CraneName.PlaceAt <location>
    // Check the number of arguments
    CheckNumArguments(1, szArgList);
std::string szPlaceParam = szArgList.front();

// Construct the PLACE statement to be executed
m_szScratch = "PLACE " + m_szCrawlerName + " AT " +
              szPlaceParam;
dcvExecuteStatement(m_szScratch.c_str());

return true;
}

int
dcvCrawlerCrane::HoistBy(std::list<std::string> szArgList)
{
    // CraneName.HoistBy <amount> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    std::string szHoistAmt = szArgList.front();
    szArgList.pop_front();
    std::string szHoistTime = szArgList.front();

    // Check if the requested hoist is within the crane's
    // range.
    double dHoistAmt = atof(szHoistAmt.c_str());
    m_dCableDropLength -= dHoistAmt;

    if(m_dCableDropLength < 1)
    {
        char szErrorMessage[256];
        sprintf(szErrorMessage, "The requested hoist is
                  beyond %s's reach.",
                m_szName.c_str());
        throw(szErrorMessage);
    }

    // Construct and execute the required Vitascope
    // statements.

    char szScratch2[MAX_STATEMENT_LENGTH];

    sprintf(szScratch2, "SCALE %s '(0,-%f,0)' %s",
            m_szCableName.c_str(), dHoistAmt,
            szHoistTime.c_str());
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "SLIDE %s '(0,%f,0)' %s",
    m_szHookName.c_str(), dHoistAmt,
    szHoistTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::LowerBy(std::list<std::string> szArgList)
{
    // CraneName.LowerBy <amount> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    std::string szLowerAmt = szArgList.front();
    szArgList.pop_front();
    std::string szLowerTime = szArgList.front();

    // No need to check if the requested lower is within
    // the crane's range. Current assumption is that the
    // cable length is infinite.
    double dLowerAmt = atof(szLowerAmt.c_str());
    m_dCableDropLength += dLowerAmt;

    // Construct and execute the required Vitascope
    // statements
    char szScratch2[MAX_STATEMENT_LENGTH];

    sprintf(szScratch2, "SCALE %s '(0,%s,0)' %s",
        m_szCableName.c_str(), dLowerAmt,
        szLowerTime.c_str());

dcvExecuteStatement(szScratch2);

    sprintf(szScratch2, "SLIDE %s '(0,-%s,0)' %s",
        m_szHookName.c_str(), dLowerAmt,
        szLowerTime.c_str());

dcvExecuteStatement(szScratch2);
return true;
}

int
dcvCrawlerCrane::HookAt(std::list<std::string> szArgList)
{
  // CraneName.HookAt <elevation> <time>

  // Check the number of arguments
  CheckNumArguments(2, szArgList);

  std::string szElevation = szArgList.front();
  szArgList.pop_front();
  std::string szOperTime = szArgList.front();

  // We need to adjust for the hook and cable's initial
  // position before executing target statements. In
  // addition, since this is a crawler crane, we need to
  // use the current boom angle and the sizes of the
  // crawler, cab, and boom length in our calculations.

  double dElevation = atof(szElevation.c_str());

  // In default position, the hook is 1.777m below the
  // top of the pulley. So the maximum elevation of the
  // hook is given by (dBoomTipYPos - 1.777), so the
  // target elevation cannot be higher than that. We
  // need to check that here. The dBoomTipYPos in turn
  // depends on the component sizes and the current boom
  // angle.
  dcvObject* pCCCrawler =
    dcvGetObject(m_szCrawlerName.c_str());
  double dCrawlYpos = (pCCCrawler->GetCurPosn())[1];
  double dBoomTipYPos = dCrawlYpos + m_dBoomAttachHeight
                        + m_dBoomLength * 
                        sin(CS_DEG2RAD(m_dBoomAngle));
  double dMaxPermElev = dBoomTipYPos - 1.777;
  if(dElevation > dMaxPermElev)
  {
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The hook's target
            elevation is beyond %s's reach.",
            m_szName.c_str());
throw(szErrorMessage);
}

// Then compute how much the cable must drop so that
// the hook reaches the desired elevation. For now,
// there are no lower bounds. The hook could be
// lowered into a pit of infinite depth assuming
// that the cable length is infinite. We need a method
// that allows us to set the max length of the cable.

// When the scale of the cable is (1,1,1), the hook is
// 1.777m below the boom tip. The cable is 1m in
// length at this scale and the hook is 0.777m deep
// from its origin.
double dTgtScaleAndSlideParam = dBoomTipYPos -
                        dElevation - 0.777;

m_dCableDropLength = dTgtScaleAndSlideParam;

// Construct and execute the required Vitascope
// statements
char szScratch2[MAX_STATEMENT_LENGTH];
sprintf(szScratch2, "TGTSCALE %s '(1,%f,1)' %s",
            m_szCableName.c_str(),
            dTgtScaleAndSlideParam, szOperTime.c_str());
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTSLIDE %s '(0,-%f,0)' %s",
            m_szHookName.c_str(),
            dTgtScaleAndSlideParam, szOperTime.c_str());
dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::SwingBy(std::list<std::string> szArgList)
{
    // CraneName.SwingBy   <amount> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

std::string szSwingAmt = szArgList.front();
szArgList.pop_front();
std::string szOperTime = szArgList.front();

// Construct and execute the required Vitoscope
// statements

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "ROTATE %s HOR %s %s",
        m_szCabinName.c_str(), szSwingAmt.c_str(),
        szOperTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::SwingToward(std::list<std::string>
szArgList)
{
    // CraneName.SwingToward <vector OR object> <time>
    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    std::string szObjOrVec = szArgList.front();
    szArgList.pop_front();
    std::string szOperTime = szArgList.front();

    // First check if there is a object by that name.
    // If there is, then get its position vector. If not,
    // assume a vector has been passed in and get a
    // ceVec3f.

    csVec3f vTarget;

dcvObject* pdcvObject =
    dcvGetObject(szObjOrVec.c_str());

    if(pdcvObject) // There is an object by that name.
    {
        vTarget = pdcvObject->GetCurPosn();
    }
else // No object, vector passed in.
{
    vTarget =
        dcvGetCsVec3fFromToken(szObjOrVec.c_str());
}

// Now compute the amount of rotation necessary to
// align the cab and boom with the target location.

dcvObject* pCCCrawler =
    dcvGetObject(m_szCrawlerName.c_str());

csVec3f vCurCranePos = pCCCrawler->GetCurPosn();

double dTgtHeading = GetHeading(vCurCranePos[0],
    vCurCranePos[2],
    vTarget[0],
    vTarget[2]);

// Subtract the Crawler's rotation from the target
// since crawler is already rotated
dTgtHeading = dTgtHeading -
    pCCCrawler->GetCurHorRotn();

// Construct and execute the required Vitascope
// statements

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "TGTROTATE %s HOR %f %s",
    m_szCabinName.c_str(),
    dTgtHeading, szOperTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::LowerBoom(std::list<std::string>
    szArgList)
{
    // CraneName.LowerBoom <amount> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    return true;
}
std::string szLowerAmt = szArgList.front();
szArgList.pop_front();
std::string szOperTime = szArgList.front();

double dLowerAmt = atof(szLowerAmt.c_str());

m_dBoomAngle -= dLowerAmt;

if (m_dBoomAngle < 10 || m_dBoomAngle > 80)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The requested boom angle is beyond %s's reach.\n", m_szName.c_str());
    throw(szErrorMessage);
}

// Construct and execute the required Vitascope statements

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "TGTROTATE %s VERT %f %s", 
    m_szCableName.c_str(), 90-m_dBoomAngle, 
    szOperTime.c_str());

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTROTATE %s VERT -%f %s", 
    m_szBoomName.c_str(), 90-m_dBoomAngle, 
    szOperTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int dcvCrawlerCrane::RaiseBoom(std::list<std::string> szArgList)
{
    // CraneName.RaiseBoom <amount> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    return true;
}
std::string szRaiseAmt = szArgList.front();
szArgList.pop_front();
std::string szOperTime = szArgList.front();

double dRaiseAmt = atof(szRaiseAmt.c_str());

m_dBoomAngle += dRaiseAmt;

if(m_dBoomAngle < 10 || m_dBoomAngle > 80)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The requested boom angle
    is beyond %s's reach.", m_szName.c_str());
    throw(szErrorMessage);
}

// Construct and execute the required Vitascope
// statements

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "TGTROTATE %s VERT %f %s",
    m_szCableName.c_str(), 90-m_dBoomAngle, szOperTime.c_str());

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTROTATE %s VERT -%f %s",
    m_szBoomName.c_str(), 90-m_dBoomAngle, szOperTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::BoomAngle(std::list<std::string>
    szArgList)
{
    // CraneName.BoomAngle  <target boom angle> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);
std::string szTgtBoomAngle = szArgList.front();
szArgList.pop_front();
std::string szOperTime = szArgList.front();

double dTgtBoomAngle = atof(szTgtBoomAngle.c_str());
m_dBoomAngle = dTgtBoomAngle;

if(m_dBoomAngle < 10 || m_dBoomAngle > 80)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The requested boom angle is beyond %s's reach.",
            m_szName.c_str());
    throw(szErrorMessage);
}

// Construct and execute the required Vitascope statements
char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "TGTROTATE %s VERT %f %s",
        m_szCableName.c_str(), 90-m_dBoomAngle,
        szOperTime.c_str());
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTROTATE %s VERT -%f %s",
        m_szBoomName.c_str(), 90-m_dBoomAngle,
        szOperTime.c_str());
dcvExecuteStatement(szScratch2);

return true;
}

int dcvCrawlerCrane::BoomOver(std::list<std::string> szArgList)
{
    // CraneName.BoomOver <vector OR object> <time>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

    return true;
}
std::string szObjOrVec = szArgList.front();
szArgList.pop_front();
std::string szOperTime = szArgList.front();

// First check if there is an object by that name.
// If there is, then get its position vector. If not,
// assume a vector has been passed in and get a
// csVec3f.

csVec3f vTarget;
dcvObject* pdcvObject =
dcvGetObject(szObjOrVec.c_str());

if(pdcvObject) // There is an object by that name.
{
    vTarget = pdcvObject->GetCurPosn();
}
else // No object, vector passed in.
{
    vTarget =
dcvGetcsVec3fFromToken(szObjOrVec.c_str());
}

// Now compute the amount of boom rotation necessary
// to align the boom tip with the target location.

dcvObject* pCCCrawler =
dcvGetObject(m_szCrawlerName.c_str());

csVec3f vCurCranePos =  pCCCrawler->GetCurPosn();

double dTgtDistance = GetLength(vCurCranePos[0],
    vCurCranePos[2],
    vTarget[0],
    vTarget[2]);

// Check if the target is within reach
double dMinDist = m_dBoomLength * cos(CS_DEG2RAD(80));
double dMaxDist = m_dBoomLength * cos(CS_DEG2RAD(10));

if(dTgtDistance > dMaxDist || dTgtDistance < dMinDist)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "The boom's target
    position is beyond %s's reach.",
        m_szName.c_str());
}
throw(szErrorMessage);
}

// Compute the angle the boom has to be at to be over
// the target radius
m_dBoomAngle =
    CS_RAD2DEG(acos(dTgtDistance/m_dBoomLength));

// Construct and execute the required Vitascope
// statements

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "TGTROTATE %s VERT %f %s",
    m_szCableName.c_str(), 90-m_dBoomAngle,
    szOperTime.c_str());

dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "TGTROTATE %s VERT -%f %s",
    m_szBoomName.c_str(), 90-m_dBoomAngle,
    szOperTime.c_str());

dcvExecuteStatement(szScratch2);

return true;
}

int
dcvCrawlerCrane::PutThatThere(std::list<std::string>
    szArgList)
{
    // CraneName.PutThatThere <object> <objheight>
    //       <pivotpt> <targetpt> <horornt> <vertornt>
    //       <clearance> <time>;

    // ASSUMPTIONS:
    // The pivot point is the place on an object where the
    // hook will be attached to it. This point is in the
    // object's local space.
    // Any change in vertical orientation occurs during
    // the hoist
    // Any adjustment to horizontal orientation occurs
    // during lowering
    // After detaching, object is placed and oriented
    // globally

    //
// Clearance is the amount by which the hook must
// lower its load to reach the object at its target.

// This function essentially concatenates and executes
// the following basic Vitascope language statements
// in one shot. Of course, the argument values are
// different each time
// TGTROTATE Cabin HOR 270.00 26.57;
// TGTROTATE Cable VERT 10.47 26.57;
// TGTROTATE Boom VERT -10.47 26.57;
// TIME 8940.87;
// TGTSCALE Cable (1,44.48,1) 25.52;
// TGTSIDE Hook (0,-44.48,0) 25.52;
// TIME 8994.52;
// ATTACH Shape Hook (0,-0.6,0);
// TIME 8994.52;
// SCALE Cable (0,-5.00,0) 26.86;
// SLIDE Hook (0,5.00,0) 26.86;
// TIME 9021.38;
// TGTROTATE Cabin HOR 296.57 29.37;
// TGTROTATE Cable VERT 11.73 29.37;
// TGTROTATE Boom VERT -11.73 29.37;
// TIME 9050.75;
// TGTSCALE Cable (1,46.05,1) 26.66;
// TGTSIDE Hook (0,-46.05,0) 26.66;
// TIME 9104.79;
// DETACH Shape;

// Check the number of arguments
CheckNumArguments(8, szArgList);

std::list<std::string>::iterator myListIterator =
    szArgList.begin();

std::string szObject = *myListIterator;
std::string szObjHeight = *(++myListIterator);
std::string szPivotPt = *(++myListIterator);
std::string szTargetPt = *(++myListIterator);
std::string szTgtHorRot = *(++myListIterator);
std::string szTgtVertRot = *(++myListIterator);
std::string szVertClearance = *(++myListIterator);
std::string szOperTime = *(++myListIterator);

dcvObject* pdcvObject =
    dcvGetObject(szObject.c_str());

double dObjHeight = atof(szObjHeight.c_str());
csVec3f vSource = pdcvObject->GetCurPosn();
csVec3f vPivot =
    dcvGetcsVec3fFromToken(szPivotPt.c_str());
csVec3f vHookSrcTgt = vSource + vPivot;

dcvObject* pdcvTgtObject =
    dcvGetObject(szTargetPt.c_str());
csVec3f vDest;

if(pdcvTgtObject) // There is an object by that name.
{
    // Get the target object's global position,
    // it might be attached
    csMatrix4f ColMajorObjTransMat;

    ColMajorObjTransMat =
        GetGlbNonScaledTransformMatrix(pdcvTgtObject);

    vDest.set(0, ColMajorObjTransMat.get(3,0));
    vDest.set(1, ColMajorObjTransMat.get(3,1));
    vDest.set(2, ColMajorObjTransMat.get(3,2));
}
else // No object, vector passed in.
{
    vDest =
        dcvGetcsVec3fFromToken(szTargetPt.c_str());
}

csVec3f vHookDestTgt = vDest + vPivot;
double dSrcHorRot = pdcvObject->GetCurHorRotn();
// Convert to positive heading
while(dSrcHorRot < 0) dSrcHorRot += 360;
double dSrcVertRot = pdcvObject->GetCurVertRotn();
// Convert to positive heading
while(dSrcVertRot < 0) dSrcVertRot += 360;
double dDestHorRot = atof(szTgtHorRot.c_str());
// Convert to positive heading
while(dDestHorRot < 0) dDestHorRot += 360;
double dDestVertRot = atof(szTgtVertRot.c_str());

// Convert to positive heading
while(dDestVertRot < 0) dDestVertRot += 360;

double dVertClear = atof(szVertClearance.c_str());
double dOperTime = atof(szOperTime.c_str());

// Check that operation time is positive
if(dOperTime <= 0)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "A crawler crane lift
    must have a positive duration.");
    throw(szErrorMessage);
}

// Subtasks to be performed or scheduled and
// associated time percentages
// 1. Lift boom to be at target distance + Lift empty
// hook to safety - 5
// 2. Swing cab to align with load source + adjust
// boom - 15
// 3. Lower hook till it is at the required position -
// 10
// 4. Tether sling to load - 10
// 5. Hoist hook to safety - 15
// 6. Swing cab to align with target + adjust boom -
// 20
// 7. Lower hook till it is at the required target -
// 15
// 8. Detach the load - 10

char szScratch2[MAX_STATEMENT_LENGTH];
double dTaskTime = dcvGetCurSimTime();

// Compute the safe elevation for the hook. vDest[1]
// is the elevation of the target position.
double dSafeHookElev = vDest[1] + dVertClear;

// 1. Lift boom to be at target distance + Lift empty
// hook to safety - 5

sprintf(szScratch2, "%s.BoomOver (%f,%f,%f) %f",
    m_szName.c_str(), vHookDestTgt[0],
    vHookDestTgt[1], vHookDestTgt[2],
    0.05*dOperTime);
dcvExecuteStatement(szScratch2);

sprintf(szScratch2, "%s.HookAt %f %f",
    m_szName.c_str(), dSafeHookElev, 0.05*dOperTime);

dcvExecuteStatement(szScratch2);

// Required for scheduling future events
dTaskTime += 0.05*dOperTime;

// 2. Swing cab to align with load source + adjust
// boom - 15

sprintf(szScratch2, "%s.SwingToward %s %f",
    m_szName.c_str(), szObject.c_str(),
    0.15*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "%s.BoomOver %s %f",
    m_szName.c_str(), szObject.c_str(),
    0.15*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.15*dOperTime;

// 3. Lower hook till it is at the required position - 10

sprintf(szScratch2, "%s.HookAt %f %f",
    m_szName.c_str(), vHookSrcTgt[1],
    0.10*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.10*dOperTime;

// 4. Tether sling to load - 10
// Attach point is -pivot

sprintf(szScratch2, "ATTACH %s %s (%f,%f,%f)",
    szObject.c_str(), m_szHookName.c_str(),
    -vPivot[0], -vPivot[1]-0.777, -vPivot[2]);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.10*dOperTime;

// 5. Hoist hook to safety - 15
sprintf(szScratch2, "%s.HookAt %f %f",
        m_szName.c_str(), dSafeHookElev, 0.15*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// The vertical orientation of the object must be set
// to its target vertical orientation during the hoist
double dVertRotAdj = dDestVertRot - dSrcVertRot;

// If more than 180 degrees of rotation is involved,
// go the other way around
if(dVertRotAdj > 180) dVertRotAdj -= 360;
if(dVertRotAdj < 180) dVertRotAdj += 360;

double dVertRotAdjTime =
    (0.15*dOperTime)*dObjHeight/               (dSafeHookElev-vHookSrcTgt[1]);

sprintf(szScratch2, "ROTATE %s VERT %f %f",
        szObject.c_str(), dVertRotAdj, dVertRotAdjTime);

if(dVertRotAdj != 0 && dVertRotAdj != 360)
    dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.15*dOperTime;

// 6. Swing cab to align with target + adjust boom -
// 20
sprintf(szScratch2, "%s.SwingToward (%f,%f,%f) %f",
        m_szName.c_str(), vHookDestTgt[0],
        vHookDestTgt[1], vHookDestTgt[2],
        0.20*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "%s.BoomOver (%f,%f,%f) %f",
        m_szName.c_str(), vHookDestTgt[0],
        vHookDestTgt[1], vHookDestTgt[2],
        0.20*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.20*dOperTime;

// 7. Lower hook till it is at the required target -
// 15
sprintf(szScratch2, "%s.HookAt %f %f",
    m_szName.c_str(), vHookDestTgt[1],
    0.15*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// The horizontal orientation must be adjusted here
// during the descent.
// First, find the rotation that object will end up in
// when swung over destination
dcvObject* pCCCrawler =
    dcvGetObject(m_szCrawlerName.c_str());
csVec3f vCurCranePos = pCCCrawler->GetCurPosn();

double dBoomRotWhenAboveLoadSrc =
    GetHeading(vCurCranePos[0], vCurCranePos[2],
    vHookSrcTgt[0], vHookSrcTgt[2]);
double dBoomRotWhenAboveLoadDest =
    GetHeading(vCurCranePos[0], vCurCranePos[2],
    vHookDestTgt[0], vHookDestTgt[2]);

double dLoadSwingAmt = dBoomRotWhenAboveLoadDest -
    dBoomRotWhenAboveLoadSrc;

double dHorObjRotAftSwing = dSrcHorRot +
    dLoadSwingAmt;

while(dHorObjRotAftSwing < 0)
    dHorObjRotAftSwing += 360;

// The rotation that it should be in when in place is
// dDestVertRot
double dHorRotAdj = dDestVertRot - dHorObjRotAftSwing;
// If more than 180 degrees of rotation is involved,  
// go the other way around
if (dHorRotAdj > 180) dHorRotAdj -= 360;
if (dHorRotAdj < 180) dHorRotAdj += 360;

sprintf(szScratch2, "ROTATE %s HOR %f %f",
        szObject.c_str(), dHorRotAdj, 0.15*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.15*dOperTime;

// 8. Detach the load - 10
sprintf(szScratch2, "DETACH %s", szObject.c_str());
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "PLACE %s AT (%f,%f,%f)",
        szObject.c_str(), vDest[0],
        vDest[1], vDest[2]);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "HORIZORIENT %s %f",
        szObject.c_str(),dDestHorRot);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "VERTORIENT %s %f",
        szObject.c_str(),dDestVertRot);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

return true;
}

int
dcvCrawlerCrane::PutThisThere(std::list<std::string>  
        szArgList)
{
    // CraneName.PutThisThere <object> <objheight>  
    // <pivotpt> <targetpt> <horornt> <vertornt>  
    // <clearance> <time>;

}
// ASSUMPTIONS:
// The Crane’s hook is already at the object’s
// attachment point ready to be tethered.
// The pivot point is the place on an object where the
// hook will be attached to it. This point is in the
// object's local space.
// Any change in vertical orientation occurs during
// the hoist
// Any adjustment to horizontal orientation occurs
// during lowering
// After detaching, object is placed and oriented
// globally
// Clearance is the amount by which the hook must
// lower its load to reach the object at its target.

// This function essentially concatenates and executes
// the following basic Vitascope language statements
// in one shot. Of course, the argument values are
// different each time
//ATTACH Shape Hook (0,-0.6,0);
//TIME 8994.52;
//SCALE Cable (0,-5.00,0) 26.86;
//SLIDE Hook (0,5.00,0) 26.86;
//TIME 9021.38;
//TGTROTATE Cabin HOR 296.57 29.37;
//TGTROTATE Cable VERT 11.73 29.37;
//TGTROTATE Boom VERT -11.73 29.37;
//TIME 9050.75;
//TGTSCALE Cable (1,46.05,1) 26.66;
//TGTSLIDE Hook (0,-46.05,0) 26.66;
//TIME 9104.79;
//DETACH Shape;

// Check the number of arguments
CheckNumArguments(8, szArgList);

std::list<std::string>::iterator myListIterator =
    szArgList.begin();

std::string szObject = *myListIterator;
std::string szObjHeight = * (++myListIterator);
std::string szPivotPt = * (++myListIterator);
std::string szTargetPt = * (++myListIterator);
std::string szTgtHorRot = * (++myListIterator);
std::string szTgtVertRot = * (++myListIterator);
std::string szVertClearance = * (++myListIterator);
std::string szOperTime = *(++myListIterator);

dcvObject* pdcvObject =
    dcvGetObject(szObject.c_str());

double dObjHeight = atof(szObjHeight.c_str());

csVec3f vSource = pdcvObject->GetCurPosn();

csVec3f vPivot =
    dcvGetcsVec3fFromToken(szPivotPt.c_str());

csVec3f vHookSrcTgt = vSource + vPivot;

dcvObject* pdcvTgtObject =
    dcvGetObject(szTargetPt.c_str());

csVec3f vDest;

if(pdcvTgtObject) // There is an object by that name.
{
    // Get the target object's global position, it
    // might be attached.
    csMatrix4f ColMajorObjTransMat;
    ColMajorObjTransMat =
        GetGlbNonScaledTransformMatrix(pdcvTgtObject);

    vDest.set(0, ColMajorObjTransMat.get(3,0));
    vDest.set(1, ColMajorObjTransMat.get(3,1));
    vDest.set(2, ColMajorObjTransMat.get(3,2));
}
else // No object, vector passed in.
{
    vDest =
        dcvGetcsVec3fFromToken(szTargetPt.c_str());
}

csVec3f vHookDestTgt = vDest + vPivot;

double dSrcHorRot = pdcvObject->GetCurHorRotn();

// Convert to positive heading
while(dSrcHorRot < 0) dSrcHorRot += 360;

double dSrcVertRot = pdcvObject->GetCurVertRotn();

// Convert to positive heading
while(dSrcVertRot < 0) dSrcVertRot += 360;
double dDestHorRot = atof(szTgtHorRot.c_str());

// Convert to positive heading
while(dDestHorRot < 0) dDestHorRot += 360;

double dDestVertRot = atof(szTgtVertRot.c_str());

// Convert to positive heading
while(dDestVertRot < 0) dDestVertRot += 360;

double dVertClear = atof(szVertClearance.c_str());
double dOperTime = atof(szOperTime.c_str());

// Check that operation time is positive
if(dOperTime <= 0)
{
    char szErrorMessage[256];
    sprintf(szErrorMessage, "A crawler crane lift "
            "must have a positive duration.");
    throw(szErrorMessage);
}

// Subtasks to be performed or scheduled and
// associated time percentages
// 1. Tether sling to load - 15
// 2. Hoist hook to safety - 20
// 3. Swing cab to align with target + adjust
//    boom - 25
// 4. Lower hook till it is at the required
//    target - 25
// 5. Detach the load - 15

char szScratch2[MAX_STATEMENT_LENGTH];
double dTaskTime = dcvGetCurSimTime();

// Compute the safe elevation for the hook. vDest[1]
// is the elevation of the target position.
double dSafeHookElev = vDest[1] + dVertClear;

// 1. Tether sling to load - 15
// Attach point is -pivot
sprintf(szScratch2, "ATTACH %s %s (%f,%f,%f)",
    szObject.c_str(), m_szHookName.c_str(),
    -vPivot[0], -vPivot[1]+0.777, -vPivot[2]);

dcvExecuteStatement(szScratch2);
// Required for scheduling future events
dTaskTime += 0.15*dOperTime;

// 2. Hoist hook to safety - 20
sprintf(szScratch2, "%s.HookAt %f %f",
    m_szName.c_str(), dSafeHookElev, 0.15*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

// The vertical orientation of the object must be set
// to its target vertical orientation during the hoist
double dVertRotAdj = dDestVertRot - dSrcVertRot;

// If more than 180 degrees of rotation is involved,
// go the other way around
if(dVertRotAdj > 180) dVertRotAdj -= 360;
if(dVertRotAdj < 180) dVertRotAdj += 360;

double dVertRotAdjTime =
    (0.15*dOperTime)*dObjHeight/
    (dSafeHookElev-vHookSrcTgt[1]);

sprintf(szScratch2, "ROTATE %s VERT %f %f",
    szObject.c_str(), dVertRotAdj, dVertRotAdjTime);

if(dVertRotAdj != 0 && dVertRotAdj != 360)
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.20*dOperTime;

// 3. Swing cab to align with target + adjust
// boom - 25

sprintf(szScratch2, "%s.SwingToward (%f,%f,%f) %f",
    m_szName.c_str(), vHookDestTgt[0],
    vHookDestTgt[1], vHookDestTgt[2],
    0.20*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "%s.BoomOver (%f,%f,%f) %f",
    m_szName.c_str(), vHookDestTgt[0],
    vHookDestTgt[1], vHookDestTgt[2],
    0.20*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.25*dOperTime;

// 4. Lower hook till it is at the required
// target - 25

sprintf(szScratch2, "%s.HookAt %f %f",
    m_szName.c_str(), vHookDestTgt[1],
    0.15*dOperTime);

dcvScheduleStatementExecution(dTaskTime, szScratch2);

// The horizontal orientation must be adjusted here
// during the descent. First, find the rotation that
// object will end up in when swung over destination

dcvObject* pCCCrawler =
    dcvGetObject(m_szCrawlerName.c_str());

csVec3f vCurCranePos = pCCCrawler->GetCurPosn();

double dBoomRotWhenAboveLoadSrc =
    GetHeading(vCurCranePos[0], vCurCranePos[2],
               vHookSrcTgt[0], vHookSrcTgt[2]);

double dBoomRotWhenAboveLoadDest =
    GetHeading(vCurCranePos[0], vCurCranePos[2],
               vHookDestTgt[0], vHookDestTgt[2]);

double dLoadSwingAmt = dBoomRotWhenAboveLoadDest -
    dBoomRotWhenAboveLoadSrc;

double dHorObjRotAftSwing = dSrcHorRot +
    dLoadSwingAmt;

while(dHorObjRotAftSwing < 0)
    dHorObjRotAftSwing += 360;

// The rotation that it should be in when in place is
// dDestVertRot

double dHorObjRotAdj = dDestVertRot - dHorObjRotAftSwing;
// If more than 180 degrees of rotation is involved,
// go the other way around
if(dHorRotAdj > 180) dHorRotAdj -= 360;
if(dHorRotAdj < 180) dHorRotAdj += 360;

sprintf(szScratch2, "ROTATE %s HOR %f %f",
    szObject.c_str(), dHorRotAdj, 0.15*dOperTime);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

// Required for scheduling future events
dTaskTime += 0.25*dOperTime;

// 5. Detach the load - 15
sprintf(szScratch2, "DETACH %s", szObject.c_str());
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "PLACE %s AT (%f,%f,%f)",
    szObject.c_str(), vDest[0],
    vDest[1], vDest[2]);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "HORIZORIENT %s %f",
    szObject.c_str(),dDestHorRot);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

sprintf(szScratch2, "VERTORIENT %s %f",
    szObject.c_str(),dDestVertRot);
dcvScheduleStatementExecution(dTaskTime, szScratch2);

return true;
}

int
dcvCrawlerCrane::Travel(std::list<std::string> szArgList)
{
    // CraneName.Travel <path name> OR <points> <time>
    // Check the number of arguments
    int nArgsInList = szArgList.size();

    if(nArgsInList < 2)
char szErrorMessage[128];
sprintf(szErrorMessage, "Incorrect number of arguments passed.");
throw(szErrorMessage);
}

// Get the travel time first
std::string szTravelTime = szArgList.back();
szArgList.pop_back();

// Now the list either contains a path name or one or more points
std::string szPathOrFirstPoint = szArgList.front();

char szScratch2[MAX_STATEMENT_LENGTH];

if(m_bHasJustBackedUp)
{
    sprintf(szScratch2, "DETACH %s", m_szCrawlerName.c_str());
dcvExecuteStatement(szScratch2);

    // Now attach the imaginary object back to this truck at its RGP
    sprintf(szScratch2, "ATTACHNOSCALE %s %s '(-%f,0,0)'", m_szImagObjName.c_str(), m_szCrawlerName.c_str(), m_dRGPDistance);

dcvExecuteStatement(szScratch2);
}

// Check if there is a path by that name
if(NULL != dcvGetPath(szPathOrFirstPoint.c_str()))
{
    if(m_bHasJustBackedUp)
    {
        // Place the truck inside the forward travel path by the wheel base distance
        sprintf(szScratch2, "PLACE %s ON %s AT %f", m_szCrawlerName.c_str(), szPathOrFirstPoint.c_str(), m_dRGPDistance);

dcvExecuteStatement(szScratch2);
    }
}
// Construct the MOVE statement to be executed
m_szScratch = "MOVE " + m_szCrawlerName + " " + szPathOrFirstPoint + " " + szTravelTime;
dcvExecuteStatement(m_szScratch.c_str());

m_bHasJustBackedUp = false;
return true;

// There is no path by that name. That means a series
// of points have been specified. Construct a path
// based on the crawler's current position and
// the supplied point and then execute the MOVE
// statement.
dcvObject* pCCCrawler = dcvGetObject(m_szCrawlerName.c_str());

csVec3f vCurCranePos = pCCCrawler->GetCurPosn();

char szScratch3[MAX_STATEMENT_LENGTH];
sprintf(szScratch2, "'(%f,%f,%f)'", vCurCranePos[0], vCurCranePos[1], vCurCranePos[2]);
sprintf(szScratch3, "%f", dcvGetCurSimTime());

std::string szNewPathName = m_szName + szScratch3;

// Construct the PATH statement to be executed
m_szScratch = "PATH " + szNewPathName + " " + szScratch2;

std::list<std::string>::iterator myListIter;
for(myListIter = szArgList.begin() ; myListIter != szArgList.end() ; myListIter++)
    { m_szScratch = m_szScratch + " " + *myListIter; }
dcvExecuteStatement(m_szScratch.c_str());
// Construct the MOVE statement to be executed
m_szScratch = "MOVE " + m_szCrawlerName + " " + szNewPathName + " " + szTravelTime;

dcvExecuteStatement(m_szScratch.c_str());

m_bHasJustBackedUp = false;

return true;
}

int
dcvCrawlerCrane::Backup(std::list<std::string> szArgList)
{
    // CraneName.Backup <path name OR points> <time>;

    // Check the number of arguments
    int nArgsInList = szArgList.size();

    if(nArgsInList < 2)
    {
        char szErrorMessage[128];
        sprintf(szErrorMessage, "Incorrect number of arguments passed.");
        throw(szErrorMessage);
    }

    char szScratch2[MAX_STATEMENT_LENGTH];

    // Prepare the crane for backing up if necessary
    if(!m_bHasJustBackedUp)
    {
        // Detach the imaginary object from the crane
        sprintf(szScratch2, "DETACH %s", m_szImagObjName.c_str());
        dcvExecuteStatement(szScratch2);

        // Now attach this crane to the imaginary object. We will then manipulate the imaginary object
        // to cause the backhoe to move in reverse
        sprintf(szScratch2, "ATTACHNOSCALE %s %s'(-%f,0,0)'", m_szCrawlerName.c_str(), m_szImagObjName.c_str(), m_dRGPDistance);
    }
dcvExecuteStatement(szScratch2);
}

// Get the travel time first
std::string szTravelTime = szArgList.back();
szArgList.pop_back();

// Now the list either contains a path name or one or
// more points
std::string szPathOrFirstPoint = szArgList.front();

// Check if there is a path by that name
if(NULL != dcvGetPath(szPathOrFirstPoint.c_str()))
{
    if(!m_bHasJustBackedUp)
    {
        // Place the imaginary object inside the
        // backup path by the RGP distance
        sprintf(szScratch2, "PLACE %s ON %s AT %f",
            m_szImagObjName.c_str(),
            szPathOrFirstPoint.c_str(),
            m_dRGPDistance);

        dcvExecuteStatement(szScratch2);
    }
}

// Construct the MOVE statement to be executed
m_szScratch = "MOVE " + m_szImagObjName + " " +
    szPathOrFirstPoint +
    " " + szTravelTime;

dcvExecuteStatement(m_szScratch.c_str());

m_bHasJustBackedUp = true;

return true;
}

// There is no path by that name. That means a series
// of points have been specified. Construct a path
// based on the crawler's current position and
// the supplied point and then execute the MOVE
// statement.
dcvObject* pCCImagObj =
    dcvGetObject(m_szImagObjName.c_str());


csVec3f vCurImagObjPos = pCCImagObj->GetCurPosn();

char szScratch3[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "'(%f,%f,%f)'", vCurImagObjPos[0], vCurImagObjPos[1], vCurImagObjPos[2]);

sprintf(szScratch3, "%f", dcvGetCurSimTime());

std::string szNewPathName = m_szName + szScratch3;

// Construct the PATH statement to be executed
m_szScratch = "PATH " + szNewPathName + " " + szScratch2;

std::list<std::string>::iterator myListIter;

for(myListIter = szArgList.begin() ; myListIter != szArgList.end() ; myListIter++)
{
    m_szScratch = m_szScratch + " " + *myListIter;
}

dcvExecuteStatement(m_szScratch.c_str());

// Construct the MOVE statement to be executed
m_szScratch = "MOVE " + m_szImagObjName + " " + szNewPathName + " " + szTravelTime;

dcvExecuteStatement(m_szScratch.c_str());

m_bHasJustBackedUp = true;

return true;
}

int
dcvCrawlerCrane::TetherLoad(std::list<std::string> szArgList)
{
    // CraneName.TetherLoad <object> <pivot_point>

    // Check the number of arguments
    CheckNumArguments(2, szArgList);

std::string szObjName = szArgList.front();
szArgList.pop_front();
std::string szPivot = szArgList.front();

csVec3f vPivot;
vPivot = dcvGetcsVec3fFromToken(szPivot.c_str());

char szScratch2[MAX_STATEMENT_LENGTH];

sprintf(szScratch2, "ATTACH %s %s (%f,%f,%f)",
        szObjName.c_str(), m_szHookName.c_str(),
        -vPivot[0], -vPivot[1]-0.777, -vPivot[2]);

dcvExecuteStatement(szScratch2);

return true;
Appendix G
ParticleWorks Add-On for VITASCOPE Reference

This appendix describes the animation statements available in the ParticleWorks add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Defining and Controlling Fluid Object Volumes

ParticleWorks statements allow instantiation and interactive manipulation of dynamic particle systems. The statements can be used together in interesting ways to describe realistic-looking masses of several common fluid construction materials such as dirt, concrete, slurry, and water.

The FUZZYOBJECT Statement

The FUZZYOBJECT statement instantiates and initializes a particle system source. The type of the source domain defines the shape of the defined particle source. Particles will be generated randomly in this shape. Once instantiated, a “fuzzy” object can be treated as any other VITASCOPE object. In particular, the object can be placed in the scene, attached to another object, and even moved on motion paths using the PLACE, ATTACH, and MOVE statements. The most common usage, however, is to attach a defined “fuzzy” object to another rigid scene object.

Syntax:    FUZZYOBJECT <ObjName> <DomainType> <Parameters>;
Example:   FUZZYOBJECT Concrete DISC (0,0,0,0,-1,0,0.2);
Example:   FUZZYOBJECT Spray LINE (0,0,-1.9,0,0,1.9);

A domain can be thought of as a representation of a region of space. Valid types of ParticleWorks domains include Points, Lines, Triangles, Planes, Rectangles, Boxes, Spheres, Cylinders, Cones, Discs, and Blobs. Each domain requires different kinds of parameters to be defined as indicated in the table below.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Parameters</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>$x, y, z$</td>
<td>Single point</td>
</tr>
<tr>
<td>Line</td>
<td>$x_1, y_1, z_1, x_2, y_2, z_2$</td>
<td>Endpoints of a line segment</td>
</tr>
<tr>
<td>Triangle</td>
<td>$x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3$</td>
<td>Vertices of a triangle</td>
</tr>
<tr>
<td>Plane</td>
<td>$o_x, o_y, o_z, n_x, n_y, n_z$</td>
<td>Point $o$ is on the plane; $n$ is a normal vector to the plane</td>
</tr>
<tr>
<td>Rectangle</td>
<td>$o_x, o_y, o_z, u_x, u_y, u_z, v_x, v_y, v_z$</td>
<td>Point $o$ is on the plane; $u$ and $v$ are non-parallel basis vectors in the plane</td>
</tr>
<tr>
<td>Box</td>
<td>$x_1, y_1, z_1, x_2, y_2, z_2$</td>
<td>Minima and maxima of an axis-aligned box</td>
</tr>
<tr>
<td>Sphere</td>
<td>$o_x, o_y, o_z, r_{radius1}, r_{radius2} = 0.0$</td>
<td>Point $o$ is the center of the sphere; $r_{radius1}$ and $r_{radius2}$ are the outer and inner radii</td>
</tr>
<tr>
<td>Cylinder</td>
<td>$x_1, y_1, z_1, x_2, y_2, z_2, r_{radius1}, r_{radius2} = 0$</td>
<td>The two points are the endpoints of the cylinder’s axis; $r_{radius1}$ and $r_{radius2}$ are the outer and inner radii</td>
</tr>
<tr>
<td>Cone</td>
<td>$x_1, y_1, z_1, x_2, y_2, z_2, r_{radius1}, r_{radius2} = 0$</td>
<td>The first point is the cone’s apex; Second is the other endpoint of the cone’s axis; $r_{radius1}$ is cone’s base radius; $r_{radius2}$ is the base radius of another cone to subtract from the first cone to create a conical shell.</td>
</tr>
<tr>
<td>Disc</td>
<td>$c_x, c_y, c_z, n_x, n_y, n_z, r_{radius1}, r_{radius2} = 0.0$</td>
<td>Point $c$ is the center of the disc in the plane with normal $n$; $r_{radius1}$ and $r_{radius2}$ are the outer and inner radii</td>
</tr>
<tr>
<td>Blob</td>
<td>$x, y, z, stddev$</td>
<td>The point is the center of a normal probability density of standard deviation $stddev$</td>
</tr>
</tbody>
</table>
**The SetSource Statement**

This statement sets/changes the source of the generated particles for a defined “fuzzy” object.

Syntax:  

```
<FuzzyObjectName>.SetSource <DomainType> <Parameters>;
```

Example:  

```
Spray.SetSource DISC (0,0,0,0,-1,0,0.2);
```

**The SetVelocity Statement**

This statement sets/changes the initial velocity of the generated particles for a defined “fuzzy” object. The sampled velocity vectors can be pictured as having their tails at the origin and their tips in the specified domain (i.e. volume).

Syntax:  

```
<FuzzyObjectName>.SetVelocity <DomainType> <Parameters>;
```

Example:  

```
Spray.SetVelocity LINE (-0.3,-0.2,0,-0.3,0.2,0);
```

**The SetColor Statement**

This statement sets/changes the color of the generated particles for a defined “fuzzy” object. The full color space is defined over the 0.0 to 1.0 range along the red, green, and blue axes. For instance, a linear color domain defined by the points (1, 0, 0) and (1, 1, 0) will choose points on a line between red and yellow. Points outside the 0.0 to 1.0 range are clamped.

Syntax:  

```
<FuzzyObjectName>.SetColor <DomainType> <Parameters>;
```

Example:  

```
Spray.SetColor LINE (1,1,1,0.5,0.6,0.9);
```

**The SetMaxParticles Statement**

This statement sets/changes the maximum number of particles that can exist in the animated scene at any given time instant. This statement is indirectly used to control the
rate at which new particles are generated. In particular, new particles will not be generated unless the old ones are destroyed if the maximum number of particles for a “fuzzy” object has been reached.

Syntax: \(<\text{FuzzyObjectName}>\.\text{SetMaxParticles} \ <\text{Number}>;\)
Example: Nozzle.SetMaxParticles 50000;

**The SetRate Statement**
This statement sets/changes the number of particles that are generated by a defined “fuzzy” object at every animated time instant. This is the rate of particle emission.

Syntax: \(<\text{FuzzyObjectName}>\.\text{SetRate} \ <\text{Number}>;\)
Example: Spray.SetRate 500;

**The SetAgeLimit Statement**
This statement sets/changes the animated time after which particles that have been generated by a defined “fuzzy” object must be destroyed. In particular, all particles older than AgeLimit are automatically destroyed.

Syntax: \(<\text{FuzzyObjectName}>\.\text{SetAgeLimit} \ <\text{AgeLimit}>;\)
Example: Spray.SetAgeLimit 15;

**The SetSpeedLimits Statement**
This statement imposes speed limits on the particles that have been generated by a defined “fuzzy” object. In particular, any particle traveling slower than the minimum speed is automatically sped up and any particle moving faster than the maximum speed is slowed down.

Syntax: \(<\text{FuzzyObjectName}>\.\text{SetSpeedLimits} \ <\text{MinSpeed}> \ <\text{MaxSpeed}>;\)
Example: Spray.SetSpeedLimits 5 20;
The SetCoeffFriction Statement
This statement sets the value of the coefficient of friction to be used when generated particles interact with other solid surfaces during animation.

Syntax:  \(<\text{FuzzyObjectName}>\).SetCoeffFriction <Value>;
Example: Concrete.SetCoeffFriction 0.1;

The SetCoeffRestitution Statement
This statement sets the value of the coefficient of restitution to be used when generated particles interact with other solid surfaces during animation.

Syntax:  \(<\text{FuzzyObjectName}>\).SetCoeffRestitution <Value>;
Example: Concrete.SetCoeffRestitution 0.05;

The AddObstruction Statement
This statement defines a physical barrier surface that obstructs the flow of generated particles. Good examples of such obstructions are the sides of concrete forms that physically contain the placed concrete’s “particles”.

Syntax:  \(<\text{FuzzyObjectName}>\).AddObstruction <DomainType> <Parameters>;
Example: Concrete.AddObstruction PLANE (0,-0.6,0,0,1,0);

The SetPrimitive Statement
This statement sets/changes the geometric primitive that is used to draw generated particles on the screen. The choice is between points and lines. In the case of lines, each particle is drawn as a line specified by two vertices - the particle's position and the particle's position minus velocity, yielding a line in the direction that the particle is traveling. In the case of points, the particles are obviously drawn at their positions.
Syntax: `<FuzzyObjectName>.SetPrimitive <PrimType>;`
Example: `Spray.SetPrimitive POINTS;`
Example: `Spray.SetPrimitive LINES;`

**The SetSize Statement**

This statement sets/changes the size of the geometric primitive that is used to draw generated particles on the screen. In the case of points, it represents the point size in pixels; in the case of lines, the line width (also in pixels) is affected.

Syntax: `<FuzzyObjectName>.SetSize <ParticleSize>;`
Example: `Spray.SetSize 2;`

**The SetSink Statement**

This statement sets/changes the ground level (Y coordinate) below which entering particles are destroyed.

Syntax: `<FuzzyObjectName>.SetSink <SinkLevel>;`
Example: `Spray.SetSink 0;`

**The SetModel Statement**

This statement allows each generated particle to be displayed as a CAD model. This, for example, can allow each concrete blob (represented by a particle) to be portrayed as a textured model instead of a point or line.

Syntax: `<FuzzyObjectName>.SetModel <CADFileName>;`
Example: `Concrete.SetModel ConcBlob.wrl;`

**The SetGravity Statement**

This statement allows the value of the gravitational constant $\ddot{G}$ to be changed. This can be useful to visually simulate buoyancy underwater or loss of gravity in outer space.
Syntax:  <FuzzyObjectName>.SetGravity <Value>;
Example:  Nozzle.SetGravity 1.635;

**Depicting Environmental Effects**

ParticleWorks also implements higher-level animation statements to describe environmental effects. In particular, statements to describe the occurrence of rain and snow are provided. These statements build upon the core ParticleWorks statements, i.e. the RAIN statement, for example, internally calls several core ParticleWorks statements described earlier.

**The RAIN Statement**

This statement can be used to depict the occurrence of rain in an animation. The statement accepts the description of the rain intensity (DRIZZLE, LIGHT, MEDIUM, or HEAVY) as an argument. Optionally, the intensity can be numerically specified on a scale of 0 to 100 where DRIZZLE = 25, LIGHT = 50, MEDIUM = 75, and HEAVY = 100. The second and third examples provided below are thus functionally equivalent.

Syntax:  RAIN <Description> [ <Intensity> ];
Example:  RAIN LIGHT;
Example:  RAIN INTENSITY 75;
Example:  RAIN MEDIUM;

**The SNOW Statement**

This statement can be used to depict the occurrence of snow in an animation. The statement accepts the description of the snow intensity (DRIZZLE, LIGHT, MEDIUM, or HEAVY) as an argument. Optionally, the intensity can be numerically specified on a scale of 0 to 100 where DRIZZLE = 25, LIGHT = 50, MEDIUM = 75, and HEAVY = 100. The second and third examples provided below are thus functionally equivalent.
Syntax: \texttt{SNOW} \texttt{<Description>} \texttt{[<Intensity>]};

Example: \texttt{SNOW} \texttt{LIGHT};

Example: \texttt{SNOW} \texttt{INTENSITY} \texttt{100};

Example: \texttt{SNOW} \texttt{HEAVY};
Appendix H
KineMach Add-On for VITASCOPE Reference

This appendix describes the animation statements available in the KineMach add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Defining and Manipulating Tower Cranes

The TOWERCRANE Statement

The TOWERCRANE statement allows the definition of a generic tower crane of any desired size. The initial height of the tower and the length of the jib (i.e. maximum lifting radius) are specified as input. More than one KineMach tower cranes (of possibly different sizes) can coexist in an animation at any given time. KineMach internally calls several core VITASCOPE language statements and assembles a tower crane of the desired configuration when a TOWERCRANE statement is processed. KineMach looks for the required CAD components (i.e. CAD models) in the /data subdirectory of the directory where VITASCOPE is installed. These required CAD models are automatically placed in the /data subdirectory when KineMach is correctly installed.

Syntax: TOWERCRANE <Name> <Height> <Radius>;
Example: TOWERCRANE Crane1 40 30;

The PlaceAt Statement

As its name implies, this statement is used to place an instantiated tower crane at the desired location in the scene.

Syntax: <TowerCraneName>.PlaceAt <Position>;
Example: Crane1.PlaceAt (40,0,30);
**The HoistBy Statement**

This statement is used to raise the hook (i.e. the cable) of an instantiated tower crane by the specified amount in the given time.

Syntax: \(<\text{TowerCraneName}>\).HoistBy <Amount> <Duration>;

Example: Crane1.HoistBy 15 17.2;

**The LowerBy Statement**

This statement is used to lower the hook (i.e. the cable) of an instantiated tower crane by the specified amount in the given time.

Syntax: \(<\text{TowerCraneName}>\).LowerBy <Amount> <Duration>;

Example: Crane1.LowerBy 25 19.1;

**The HookAt Statement**

This statement is used to specify the elevation (i.e. height – indicated as a Y coordinate) at which the hook of an instantiated tower crane should be in a given amount of time. KineMach automatically figures out whether the hook must be raised or lowered to accomplish the task.

Syntax: \(<\text{TowerCraneName}>\).HookAt <Elevation> <Duration>;

Example: Crane1.HookAt 0.25 12.5;

**The SwingBy Statement**

This statement is used to swing the jib of an instantiated tower crane by a given amount in a given time. Positive values result in anticlockwise rotation and negative values in clockwise rotation.

Syntax: \(<\text{TowerCraneName}>\).SwingBy <Amount> <Duration>;

Example: Crane1.SwingBy 135 55;
**The SwingToward Statement**

This statement is used to swing the jib of an instantiated tower crane such that it aligns itself with the given position or scene object in the specified amount of time. KineMach automatically figures out which is the shortest direction to swing the jib in.

Syntax: \(<\text{TowerCraneName}>\).SwingToward \(<\text{Point/ObjectName}>\) \(<\text{Duration}>\);

Example: Crane1.SwingToward (10,0,10) 45;

Example: Crane1.SwingToward Shape10 45;

**The TrolleyTravelBy Statement**

This statement is used to move the trolley of an instantiated tower crane along the jib in the given amount of time. Positive values for the amount of motion result in outward motion, i.e. the trolley moves away from the tower; and negative values result in inward motion, bringing the trolley closer to the tower.

Syntax: \(<\text{TowerCraneName}>\).TrolleyTravelBy \(<\text{Amount}>\) \(<\text{Duration}>\);

Example: Crane1.TrolleyTravelBy 15 25;

Example: Crane1.TrolleyTravelBy -12 23;

**The TrolleyTravelTo Statement**

This statement is used to instruct the trolley of an instantiated tower crane to move along the crane’s jib until the trolley is right above the given object or position, within the given amount of time. KineMach automatically figures out which way the trolley must move (i.e. away from or toward the tower).

Syntax: \(<\text{TowerCraneName}>\).TrolleyTravelTo \(<\text{Point/ObjName}>\) \(<\text{Duration}>\);

Example: Crane1.TrolleyTravelTo (10,0,10) 25;

Example: Crane1.TrolleyTravelTo Shape10 25;
**The PutThatThere Statement**

This advanced statement is used to instruct an instantiated tower crane to perform a full lift. In particular, the statement instructs a tower crane to pick up the specified object and install it at the indicated target location in the given amount of time. KineMach automatically figures out the motions of its components required to execute the task. The statement requires that the height of the object being lifted (for clearance computations), and the hook attachment point of the object be specified. In addition, the object’s target configuration (orientations) and the minimum clearance to maintain during the animated operation can be optionally indicated as arguments to the PutThatThere statement.

**Syntax:**

```
<TowerCraneName>.PutThatThere <ObjectName>
   <ObjectHeight>
   <AttachmentPoint>
   <TgtPosn/TgtObject>
   [<TgtHorRotn>]
   [<TgtVertRotn>]
   [<MinClearance>]
   <Duration>;
```

**Example:**

```
Crane1.PutThatThere  Compressor1 6 (0,6,0) (15,12,10) 0 0 15 1100;
```

**Example:**

```
Crane2.PutThatThere  Compressor2 6 (0,6,0) PedestalTop 0 0 15 1100;
```

**The PutThisThere Statement**

This advanced statement is similar to the PutThatThere statement described above. The only difference is that when this statement is used, the hook of the crane is assumed to be already tethered to the object to be lifted. The PutThisThere statement instructs a tower crane to install an already tethered object at the indicated target location in the given amount of time. KineMach automatically figures out the motions of its components required to execute the task. The statement requires that the height of the object being lifted (for clearance computations), and the hook attachment point of the object be
specified. In addition, the object’s target configuration (orientations) and the minimum clearance to maintain during the animated operation can be optionally indicated as arguments to the PutThisThere statement.

**Syntax:**

\(<\text{TowerCraneName}.\text{PutThisThere} \ <\text{ObjectName}> \ <\text{ObjectHeight}> \ <\text{AttachmentPoint}> \ <\text{TgtPosn/TgtObject}> \ [<\text{TgtHorRotn}>] \ [<\text{TgtVertRotn}>] \ [<\text{MinClearance}>] \ <\text{Duration}>;\)

**Example:**

\(\text{Crane1.PutThisThere Bucket1 2 (0,2,0) (15,12,10) 0 0 5 120;}\)

**Example:**

\(\text{Crane2.PutThisThere Bucket2 2 (0,2,0) DeckTop 0 0 5 120;}\)

---

**The ChangeHeight Statement**

The ChangeHeight statement is used dynamically change the height of the tower of an instantiated tower crane. This is useful for portraying typical tower cranes that increase in height as the structure they are constructing (e.g. a skyscraper) rises. The actual process of installing additional tower segments is not portrayed; only the tower is scaled in the vertical direction over the indicated time duration.

**Syntax:**

\(<\text{TowerCraneName}.\text{ChangeHeight} \ <\text{NewTowerHeight}> \ <\text{Duration}>;\)

**Example:**

\(\text{Crane1.ChangeHeight 50 2500;}\)

---

**The TetherLoad Statement**

This statement attaches the specified scene object to the hook of an instantiated tower crane at the desired attachment point.
Defining and Manipulating Crawler-Mounted Cranes

The CRAWLERCRANE Statement

The CRAWLERCRANE statement allows the definition of a generic crawler-mounted lattice boom crane of any desired size. The width of the crawlers and the length of the boom are specified as input. More than one KineMach crawler cranes (of possibly different sizes) can coexist in an animation at any given time. KineMach internally calls several core VITASCOPE language statements and assembles a crawler-mounted crane of the desired configuration when a CRAWLERCRANE statement is processed. KineMach looks for the required CAD components (i.e. CAD models) in the /data subdirectory of the directory where VITASCOPE is installed. These required CAD models are automatically placed in the /data subdirectory when KineMach is correctly installed.

Syntax: CRAWLERCRANE <Name> <CrawlerWidth> <BoomLength>;

Example: CRAWLERCRANE Crane2 10 65;

The PlaceAt Statement

As its name implies, this statement is used to place an instantiated crawler-mounted crane at a desired location in the scene.

Syntax: <CrawlerCraneName>.PlaceAt <Position>;

Example: Crane2.PlaceAt (40,0,30);

The PlaceOn Statement

This statement is used to place an instantiated crawler-mounted crane on a predefined motion path.
Syntax: \(<\text{CrawlerCraneName}\>.\text{PlaceOn} \ <\text{PathName}>;\)
Example: Crane2.PlaceOn TravelToLoad;

**The HoistBy Statement**
This statement is used to raise the hook (i.e. the cable) of an instantiated crawler-mounted crane by the specified amount in the given time.

Syntax: \(<\text{CrawlerCraneName}\>.\text{HoistBy} \ <\text{Amount}> \ <\text{Duration}>;\)
Example: Crane2.HoistBy 15 17.2;

**The LowerBy Statement**
This statement is used to lower the hook (i.e. the cable) of an instantiated crawler-mounted crane by the specified amount in the given time.

Syntax: \(<\text{CrawlerCraneName}\>.\text{LowerBy} \ <\text{Amount}> \ <\text{Duration}>;\)
Example: Crane2.LowerBy 25 18.9;

**The HookAt Statement**
This statement is used to specify the elevation (i.e. height – indicated as a Y coordinate) at which the hook of an instantiated crawler-mounted crane should be in a given amount of time. KineMach automatically figures out whether the hook must be raised or lowered to accomplish the task.

Syntax: \(<\text{CrawlerCraneName}\>.\text{HookAt} \ <\text{Elevation}> \ <\text{Duration}>;\)
Example: Crane2.HookAt 0.25 12.5;

**The SwingBy Statement**
This statement is used to swing the cabin of an instantiated crawler-mounted crane by a given amount in a given time. Positive values result in anticlockwise rotation and negative values in clockwise rotation of the crane’s cabin.
Syntax:  <CrawlerCraneName>.SwingBy <Amount> <Duration>;
Example:  Crane2.SwingBy 135 55;

**The SwingToward Statement**

This statement is used to swing the cabin of an instantiated crawler-mounted crane such that it aligns the crane’s boom with the given position or scene object in the specified amount of time. KineMach automatically figures out which is the shortest direction to swing the cabin in.

Syntax:  <CrawlerCraneName>.SwingToward <Point/ObjName> <Duration>;
Example:  Crane2.SwingToward (10,0,10) 45;
Example:  Crane2.SwingToward Shape10 45;

**The LowerBoom Statement**

This statement is used to lower the boom of an instantiated crawler-mounted crane by the indicated amount (specified in degrees) within the specified time.

Syntax:  <CrawlerCraneName>.LowerBoom <Amount> <Duration>;
Example:  Crane2.LowerBoom 25 43;

**The RaiseBoom Statement**

This statement is used to raise the boom of an instantiated crawler-mounted crane by the indicated amount (specified in degrees) within the specified time.

Syntax:  <CrawlerCraneName>.RaiseBoom <Amount> <Duration>;
Example:  Crane2.RaiseBoom 25 54;
The BoomAngle Statement
This statement is used to explicitly indicate the angle (specified in degrees) that the boom of an instantiated crawler-mounted crane should be at within the indicated amount of time. A value of 0 indicates the boom is horizontal; a value of 90 represents a vertical boom.

Syntax: \(<\text{CrawlerCraneName}>.\text{BoomAngle} \ <\text{TargetAngle}> \ <\text{Duration}>;\)

Example: \(\text{Crane2.BoomAngle 45 33;}\)

The BoomOver Statement
This statement will automatically adjust the boom angle of an instantiated crawler-mounted crane in such a way that the crane’s hook is right above the indicated position or scene object. This statement must be used together with the SwingToward statement. That statement aligns the cabin of the crane in a desired configuration; this statement then adjusts the boom angle.

Syntax: \(<\text{TowerCraneName}>.\text{BoomOver} \ <\text{Point/ObjectName}> \ <\text{Duration}>;\)

Example: \(\text{Crane2.BoomOver (10,0,10) 45;}\)
Example: \(\text{Crane2.BoomOver Shape10 45;}\)

The PutThatThere Statement
This advanced statement is used to instruct an instantiated crawler-mounted crane to perform a lift. In particular, the statement instructs a crawler crane to pick up the specified object and install it at the indicated target location in the given amount of time. KineMach automatically figures out the motions of its components required to execute the task. The statement requires that the height of the object being lifted (for clearance computations), and the hook attachment point of the object be specified. In addition, the object’s target configuration (orientations) and the minimum clearance to maintain during the animated operation can be optionally indicated as arguments to the statement.
Syntax: `<CrawlerCraneName>.PutThatThere <ObjectName>`

`<ObjectHeight>`
`<AttachmentPoint>`
`<TgtPosn/TgtObj>`
`[<TgtHorRotn>]`
`[<TgtVertRotn>]`
`[<MinClearance>]`
`<Duration>`;

Example: `Crane2.PutThatThere Transformer1 6 (0,6,0)`
` (15,12,10) 0 0 15 1100;`

Example: `Crane2.PutThatThere Transformer2 6 (0,6,0)`
` BaseTop 0 0 15 1100;`

The PutThisThere Statement

This advanced statement is similar to the PutThatThere statement described above. The only difference is that when this statement is used, the hook of the crane is assumed to be already tethered to the object to be lifted. The PutThisThere statement instructs a crawler-mounted crane to install an already tethered object at the indicated target location in the given amount of time. KineMach automatically figures out the motions of its components required to execute the task. The statement’s arguments are similar to those of the PutThatThere statement.

Syntax: `<CrawlerCraneName>.PutThisThere <ObjectName>`

`<ObjectHeight>`
`<AttachmentPoint>`
`<TgtPosn/TgtObj>`
`[<TgtHorRotn>]`
`[<TgtVertRotn>]`
`[<MinClearance>]`
`<Duration>`;
The Travel Statement

This statement is used to move an instantiated crawler-mounted crane on a pre-defined path in the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead of a path name. In this case, a path is dynamically constructed from the crane’s current position and the specified points. This dynamic path definition is sometimes useful when the crane’s motion paths cannot be determined a-priori.

Syntax:  
<CrawlerCraneName>.Travel  <PathName/Points>  
<Duration>;

Example:  
Crane2.Travel TravelToLoad 230;

Example:  
Crane2.Travel (10,0,10) (20,0,0) 230;

The Backup Statement

This statement is used to move an instantiated crawler-mounted crane in reverse on a pre-defined path in the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead of a path name. In this case, a path is dynamically constructed from the crane’s current position and the specified points. This dynamic path definition is sometimes useful when the crane’s motion paths cannot be determined a-priori. In case the crane is positioned at the end of another path when this statement is called, then the last segment of that path must coincide with the first segment of the backup path used as this statement’s argument.

Syntax:  
<CrawlerCraneName>.Backup  <PathName/Points>  
<Duration>;

Example:  
Crane2.Backup MoveAway 40;

Example:  
Crane2.Backup (10,0,10) (20,0,0) 130;
**The TetherLoad Statement**

This statement attaches the specified scene object to the hook of an instantiated crawler-mounted crane at the desired attachment point.

Syntax: \(<\text{CrawlerCraneName}>.\text{TetherLoad} \ <\text{ObjectName}>\)
\(<\text{AttachmentPoint}>;\)

Example:  \(\text{Crane2.TetherLoad Shape10} \ (0,0.5,0);\)

**Defining and Manipulating Backhoes**

**The BACKHOE Statement**

The BACKHOE statement allows the definition of a generic crawler-mounted hydraulic backhoe of any desired size. The width of the crawlers is specified as input. More than one KineMach backhoes (of possibly different sizes) can coexist in an animation at any given time. KineMach internally calls several core VITASCOPE language statements and assembles a backhoe of the desired scale when a BACKHOE statement is processed. KineMach looks for the required CAD components (i.e. CAD models) in the /data subdirectory of the directory where VITASCOPE is installed. These CAD models are automatically placed in the /data subdirectory when KineMach is correctly installed.

Syntax:  \(\text{BACKHOE} \ <\text{Name}> \ <\text{CrawlerWidth}>;\)

Example:  \(\text{BACKHOE Hoe1} \ 5;\)

**The PlaceAt Statement**

As its name implies, this statement is used to place an instantiated backhoe at a desired location in the scene.

Syntax:  \(<\text{BackHoeName}>.\text{PlaceAt} \ <\text{Position}>;\)

Example:  \(\text{Hoe1.PlaceAt} \ (40,0,30);\)

**The PlaceOn Statement**

This statement is used to place an instantiated backhoe on a predefined motion path.
Syntax:   <BackHoeName>.PlaceOn <PathName>;
Example:  Hoe1.PlaceOn TravelToLoadArea;

**The BoomAngle Statement**
This statement is used to explicitly indicate the angle (specified in degrees) that the boom of an instantiated backhoe should be at within the indicated amount of time. A value of 0 indicates the boom is horizontal; positive values raise the boom and negative values lower it. The valid range for the boom angle is from -45 to +80 degrees with respect to the parent cabin.

Syntax:   <BackHoeName>.BoomAngle <TargetAngle>  
<Duration>;
Example:  Hoe1.BoomAngle 45 5;

**The StickAngle Statement**
This statement is used to explicitly indicate the angle (specified in degrees) that the stick of an instantiated backhoe should be at within the indicated amount of time. A value of 0 indicates the stick is in line with the parent boom, i.e. when the stick angle is zero, the boom and the stick lie on a straight line. Higher values raise the stick and lower values lower it. The valid range for the stick angle is from -135 to 0 degrees with respect to the parent boom.

Syntax:   <BackHoeName>.StickAngle <TargetAngle>  
<Duration>;
Example:  Hoe1.StickAngle -45 7;

**The BucketAngle Statement**
This statement is used to explicitly indicate the angle (specified in degrees) that the bucket of an instantiated backhoe should be at within the indicated amount of time. A value of 0 indicates the struck surface of the dirt contained in the bucket is perpendicular to the parent stick. Higher values take the contained dirt away from the stick (dump the
dirt), whereas lower values bring the dirt surface closer to the stick (contain the dirt). The valid range for the bucket angle is from 0 to +90 degrees with respect to the parent stick.

**Syntax:**
```
<BackHoeName>.BucketAngle <TargetAngle> <Duration>;
```

**Example:**
```
Hoe1.BucketAngle 55 4;
```

### The SwingBy Statement

This statement is used to swing the cabin of an instantiated backhoe by a given amount in a given time. Positive values result in anticlockwise rotation and negative values in clockwise rotation of the backhoe’s cabin.

**Syntax:**
```
<BackHoeName>.SwingBy <Amount> <Duration>;
```

**Example:**
```
Hoe1.SwingBy 115 6;
```

### The SwingToward Statement

This statement is used to swing the cabin of an instantiated backhoe such that it aligns the hoe’s boom/stick/bucket with the given position or scene object in the specified amount of time. KineMach automatically figures out which is the shortest direction to swing the cabin in.

**Syntax:**
```
<BackHoeName>.SwingToward <Point/ObjName> <Duration>;
```

**Example:**
```
Hoe1.SwingToward (10,0,10) 5;
```

**Example:**
```
Hoe1.SwingToward Stump1 5;
```

### The Travel Statement

This statement is used to move an instantiated backhoe on a pre-defined path in the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead of a path name. In this case, a path is dynamically constructed from the hoe’s current position and
the specified points. This dynamic path definition is sometimes useful when the hoe’s
motion paths cannot be determined a-priori.

Syntax: \(<\text{BackHoeName}>\).\text{Travel} \ <\text{PathName/Points}> \\
\quad \ <\text{Duration}>;

Example: \(\text{Hoe1.Travel TravelToLoadArea 210;}

Example: \(\text{Hoe1.Travel (10,0,10) (20,0,0) 40;}

\textbf{The Backup Statement}

This statement is used to move an instantiated backhoe in reverse on a pre-defined path in
the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead
of a path name. In this case, a path is dynamically constructed from the hoe’s current
position and the specified points. This dynamic path definition is often useful when the
hoe’s motion paths cannot be determined a-priori. In case the hoe is positioned at the end
of another path when this statement is called, then the path’s last segment must coincide
with the first segment of the backup path used as this statement’s argument.

Syntax: \(<\text{BackHoeName}>\).\text{Backup} \ <\text{PathName/Points}> \\
\quad \ <\text{Duration}>;

Example: \(\text{Hoe1.Backup MoveAwayFromCut 24;}

Example: \(\text{Hoe1.Backup (10,0,10) (20,0,0) 28;}

\textbf{The BucketAt Statement}

This statement is used to specify the 3D position at which the digging edge of the bucket
of an instantiated backhoe should be at in a given amount of time. KineMach
automatically computes (using inverse kinematics techniques) all the rotations required of
the hoe’s components in order for its bucket’s edge to reach the desired target.

Syntax: \(<\text{BackHoeName}>\).\text{BucketAt} \ <\text{TargetPoint}> \ <\text{Duration}>;

Example: \(\text{Hoe1.BucketAt (10,-5,23) 12.5;}

422
**The Dig Statement**

This statement is used to instruct the bucket of an instantiated backhoe to perform a digging stroke and contain the scooped dirt in the given amount of time. The statement assumes that the bucket edge is already at a position ready to dig. In case the hoe is operating on a dynamic ViTerra (see chapter 6) implemented terrain surface, the surface of the virtual earth deforms as the digging stroke is performed. KineMach automatically figures out the bucket’s rotations in performing the digging task.

Syntax:   `<BackHoeName>.Dig <Duration>;
Example:  Hoel.Dig 6.5;

**The DigAt Statement**

This statement is equivalent to calling the BucketAt and Dig statements sequentially. KineMach automatically computes (using inverse kinematics techniques) all the rotations required of the hoe’s components in order for its bucket’s edge to reach the desired target. Then, it instructs the backhoe’s bucket to perform a digging stroke and contain the scooped dirt.

Syntax:   `<BackHoeName>.DigAt <TargetPoint> <Duration>;
Example:  Hoel.DigAt (10,-5,23) 19;

**The Dump Statement**

This statement is used to instruct the bucket of an instantiated backhoe to perform a dumping stroke and release the contained dirt in the given amount of time. The statement assumes that the bucket edge is already at a position ready to dump. In case the hoe is operating on a dynamic ViTerra (see chapter 6) implemented terrain surface, a heap of dirt appears on the surface of the virtual earth as the dumping stroke is performed. KineMach automatically figures out the bucket’s rotations in performing the dumping task.

Syntax:   `<BackHoeName>.Dump <Duration>;
Example:  Hoel.Dump 3.5;
**The DumpAt Statement**

This statement is equivalent to calling the BucketAt and Dump statements sequentially. KineMach automatically computes (using inverse kinematics techniques) all the rotations required of the hoe’s components in order for its bucket’s edge to reach the desired target. Then, it instructs the backhoe’s bucket to perform a dumping stroke and release the contained dirt.

**Syntax:**

```
<BackHoeName>.DumpAt <TargetPoint> <Duration>;
```

**Example:**

```
Hoe1.DumpAt (10,-5,23) 16;
```

**The DigHereDumpThere Statement**

This advanced statement is equivalent to calling the DigAt and DumpAt statements sequentially. KineMach automatically computes (using inverse kinematics techniques) all the rotations required of the hoe’s components in order for its bucket’s edge to reach the source and target positions. It also instructs the backhoe's bucket to perform the digging and dumping strokes at the correct times.

**Syntax:**

```
<BackHoeName>.DigHereDumpThere <SourcePoint> <TargetPoint> <Duration>;
```

**Example:**

```
Hoe1.DigHereDumpThere (-12,5,12) (10,4,-11) 30;
```

**Defining and Manipulating Dumptrucks**

**The DUMPTRUCK Statement**

The DUMPTRUCK statement allows the definition of a generic highway dumptruck of any desired size. The wheel base (distance between the front and rear axles) of the truck is specified as input. More than one KineMach dumptrucks (of possibly different sizes) can coexist in an animation at any given time. KineMach internally calls several core VITASCOPE language statements and assembles a truck of the desired configuration when a DUMPTRUCK statement is processed. KineMach looks for the required CAD components (i.e. CAD models) in the /data subdirectory of the directory where
VITASCOPE is installed. These required CAD models are automatically placed in the /data subdirectory when KineMach is correctly installed.

Syntax: \[ \text{DUMPTRUCK } \text{<TruckName>} \text{<WheelBase>} \]

Example: \[ \text{DUMPTRUCK } \text{Truck1 } 5.1 \]

**The PlaceAt Statement**

As its name implies, this statement is used to place an instantiated dumptruck at a desired location in the scene.

Syntax: \[ \text{<TruckName>.PlaceAt } \text{<Position>} \]

Example: \[ \text{Truck1.PlaceAt } (40,0,30) \]

**The PlaceOn Statement**

This statement is used to place an instantiated dumptruck on a predefined motion path.

Syntax: \[ \text{<TruckName>.PlaceOn } \text{<PathName>} \]

Example: \[ \text{Truck1.PlaceOn LoadToDump} \]

**The Travel Statement**

This statement is used to move an instantiated dumptruck on a pre-defined path in the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead of a path name. In this case, a path is dynamically constructed from the dumptruck’s current position and the specified points. This dynamic path definition is sometimes useful when the dumptruck’s motion paths cannot be determined a-priori.

Syntax: \[ \text{<TruckName>.Travel } \text{<PathName/Points>} \text{<Duration>}; \]

Example: \[ \text{Truck1.Travel LoadToDump 230;} \]

Example: \[ \text{Truck1.Travel (10,0,10) (20,0,0) 230;} \]
The Backup Statement

This statement is used to move an instantiated dumptruck in reverse on a pre-defined path in the given amount of time. Alternatively, a set of 3D coordinates can be indicated instead of a path name. In this case, a path is dynamically constructed from the dumptruck’s current position and the specified points. This dynamic path definition is sometimes useful when the dumptruck’s motion paths cannot be determined a-priori. In case the dumptruck is positioned at the end of another path when this statement is called, then the last segment of that path must coincide with the first segment of the backup path used as this statement’s argument.

Syntax:  
<TruckName>.Backup <PathName/Points> <Duration>;

Example:  
Truck1.Backup ReadyToDump 40;

Example:  
Truck1.Backup (10,0,10) (20,0,0) 130;

The Dump Statement

This statement is used to instruct an instantiated dumptruck to tip its dump bed in the given amount of time. The statement assumes that the dumptruck has maneuvered to the correct dumping position. KineMach automatically computes the dump bed’s rotations in performing the dumping task.

Syntax:  
<TruckName>.Dump <Duration>;

Example:  
Truck1.Dump 55;

The DumpBedAngle Statement

This statement is used to explicitly indicate the angle (specified in degrees) that the dump bed of an instantiated dumptruck should be at within the indicated amount of time. A value of 0 indicates the dump bed is completely lowered. Higher values raise the dump bed. The valid range for the dump bed angle is from 0 to +55 degrees with respect to the parent truck chassis.
Syntax:  

\(<TruckName>\).DumpBedAngle <TargetAngle> <Duration>;

Example:  

Truck1.DumpBedAngle 45 12;

**The Destroy Statement**

This statement is used to explicitly remove instantiated dumptrucks from the scene. The destroyed dumptruck is permanently deleted from the scene. A destroyed dumptruck cannot be manipulated without instatiating it again.

Syntax:  

\(<TruckName>\).Destroy;

Example:  

Truck1.Destroy;
Appendix I

ViTerra Supported Elevation Data Input Formats

This appendix lists the digital elevation data formats that the ViTerra add-on for VITASCOPE can interpret to automatically generate the terrain geometry of virtual construction jobsites.

USGS SDTS DEM (*CATD.DDF)
USGS ASCII DEM (.dem)
Arc/Info ASCII Grid
Arc/Info Binary Grid (.adf)
BSB Nautical Chart Format (.kap)
CEOS (Spot for instance)
First Generation USGS DOQ (.doq)
New Labeled USGS DOQ (.doq)
Military Elevation Data (.dt0, .dt1)
Eosat Fast Format
ERMmapper Compressed Wavelets (.ecw)
ESRI .hdr Labeled
Envisat Image Product (.n1)
FITS (.fits)
Graphics Interchange Format (.gif)
Arc/Info Binary Grid (.adf)
GRASS Rasters
TIFF / GeoTIFF (.tif)
Erdas Imagine (.img)
Atlantis MFF2e
Japanese DEM (.mem)
JPEG JFIF (.jpg)
Atlantis MFF
OGDI Bridge
PCI .aux Labeled
Portable Network Graphics (.png)
Netpbm (.ppm, .pgm)
SAR CEOS
X11 Pixmap (.xpm)
NOAA Polar Orbiter Level 1b Data Set (AVHRR)
Hierarchical Data Format Release 4 (HDF4)
EOSAT FAST Format
Appendix J
ViTerra Add-On for VITASCOPE Reference

This appendix describes the animation statements implemented in the ViTerra add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Defining Dynamic Terrain Surfaces

The TERRAIN Statement

The TERRAIN statement constructs a 3D terrain model from the indicated elevation (geometry) and texture (appearance) data sources. The first argument is a string that specifies the source of the digital elevation data (i.e. DEM) to be used in constructing the terrain model geometry. The second argument is a reference to the aerial image that is to be draped on the constructed geometry to describe its appearance. Appendix I lists all DEM input formats ViTerra supports. In addition, ViTerra supports all common texture image formats (e.g. JPG, GIF, PNG, BMP, etc.).

Syntax: TERRAIN <ElevationDataSrc> <TextureDataSrc>;
Example: TERRAIN '4349CATD.DDF,10,1' '4349Tex.JPG';

The elevation data source string itself is composed of 3 comma-separated values. The first token is the name of the root file that contains the stored digital elevation data; the second token is the distance (in world units) between vertices in the resultant terrain; and the last token is a vertical scaling factor to be applied to the elevation of each loaded vertex. ViTerra requires that the terrain data being loaded be a power of two in both length and width. If that is not the case, ViTerra’s elevation loader will zero-fill the terrain data to the next power of 2 in both directions. Therefore, as far as possible, it should be ensured that the specified elevation data is already a power of 2 in width and length.
The RaiseElevation Statement

This statement can be used to explicitly raise the elevation of the specified point on an instantiated terrain by the indicated amount. This method is, however, not intended to be directly used in animation trace files by animation-authoring simulation models. Instead, they are implemented so that other VITASCOPE extensions (i.e. add-ons) can manipulate instantiated 3D terrains by executing this statement from within their add-on modules.

Syntax: TERRAIN.RaiseElevation <X> <Z> <DeltaY>;
Example: TERRAIN.RaiseElevation 10 -20 0.25;

The LowerElevation Statement

This statement can be used to explicitly lower the elevation of the specified point on an instantiated terrain by the indicated amount. This method is, however, not intended to be directly used in animation trace files by animation-authoring simulation models. Instead, they are implemented so that other VITASCOPE extensions (i.e. add-ons) can manipulate instantiated 3D terrains by executing this statement from within their add-on modules.

Syntax: TERRAIN.LowerElevation <X> <Z> <DeltaY>;
Example: TERRAIN.LowerElevation -10 20 0.25;

The SetElevation Statement

This statement can be used to explicitly set the elevation of the specified point on an instantiated terrain by the indicated amount. This method is, however, not intended to be directly used in animation trace files by animation-authoring simulation models. Instead, they are implemented so that other VITASCOPE extensions (i.e. add-ons) can manipulate instantiated 3D terrains by executing this statement from within their add-on modules.

Syntax: TERRAIN.SetElevation <X> <Z> <TargetY>;
Example: TERRAIN.SetElevation 10 -20 12.25;
Core VITASCOPE Statements Redefined in ViTerra

ViTerra redefines 2 core VITASCOPE language statements; the CREATE statement and the PLACE…AT statement. The CREATE statement is redefined to include as arguments additional information that is required to compute an object’s correct orientations (roll, pitch, and yaw) as the object travels on the arbitrarily uneven ViTerra terrain surface (i.e. terrain following).

The PLACE…AT statement is redefined to take as argument a 2D coordinate instead of the 3D position coordinate required in the original statement. In particular, the redefined statement only requires the specification of the target planar (2D) position (i.e. X and Z components only). ViTerra automatically retrieves the elevation (height) of the terrain surface at the specified planar position (i.e. the Y coordinate corresponding to the specified X-Z position).

In general, ViTerra allows any position on the terrain surface to be specified as a 2D coordinate instead of a complete 3D position. This is applicable to all core and add-on defined statements. For instance, several statements such as PlaceAt, DigAt, DumpAt, and DigHereDumpThere defined in the KineMach add-on (see chapter 5) require a 3D position coordinate as argument(s). ViTerra now allows those arguments to be also specified as 2D planar coordinates thus simplifying input.

The Redefined CREATE Statement

The CREATE statement creates specific VITASCOPE scene objects by instantiating predefined classes. Several objects of the same type can be created from a class. For instance, a fleet of trucks (of the same model) can be created by instantiating several objects from the class that references that truck model’s CAD drawing. The redefined CREATE statement takes as additional arguments the distances to the created object’s left, right, front, and back contact edges. The distances are measured from the object’s local origin. The distances are represented in the arguments dL, dR, dF, and dF in the statement specification below. The figure that follows graphically describes their significance.
Syntax: CREATE <ObjectName> <ClassName> <dL> <dR> 
<dB> <dF> ;

Example: CREATE Truck1 CAT777D 2.5 2.5 2.0 2.6;

ViTerra uses this additional information to compute where the terrain contact points (e.g. tires) of the created object are. Then, when the objects move, their contact points automatically touch the terrain as closely as possible (i.e. terrain following).

**The Redefined PLACE…AT Statement**

The PLACE…AT statement is used to place a created object at a desired position in the scene. The position is specified as a planar 2D coordinate (X and Z components). ViTerra automatically constructs the complete 3D position by retrieving the elevation (height) of the terrain surface (Y component) at the specified planar point.

Syntax: PLACE <ObjectName> AT <PlanarPoint>;
Example: PLACE Truck1 AT '(10,10)';
Appendix K
PathFinder Add-On for VITASCOPE Reference

This appendix describes the animation statements implemented in the PathFinder add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Defining Accurate Motion Trajectories

The PATH2D...SPLINE Statement

This statement defines a motion path trajectory by constructing smoothly connected curved segments that pass through the indicated 2D planar coordinates. In particular, the curved segments are represented as Kochanek-Bartels Tension-Continuity-Bias (TCB) splines. The constructed 2D curved paths are then superimposed on the virtual terrain surface to define a 3D curved motion trajectory that drapes the terrain. The defined paths are dynamic in that they automatically adjust to any deformations that the underlying terrain might undergo without losing their spatial integrity, i.e. the path superimposition is performed in real-time during animation.

Syntax: PATH2D <PathName> SPLINE <Points>;
Example: PATH2D LoadToDump SPLINE '(10,10)' '(10,-30)' '(10,-130)' '(90,-130)';

Paths defined by this statement are directional, i.e. objects moved on defined paths automatically point in the direction of motion and adjust to the potentially uneven terrain (i.e. terrain following). In addition, the paths are accumulating i.e. VITASCOPE will temporarily stop a traveling object before the end of a path if it is blocked by another object. When the blocking object moves away (e.g. travels on a connected subsequent path), the stopped object resumes its motion until it reaches the end of the path or is blocked again. If that was not the case, then multiple objects accumulating at the end of a path would come to rest “within” each other, which never occurs with physical objects.
The NONDIRECPATH2D…SPLINE Statement

Paths defined by this statement are identical to those defined by the PATH2D…SPLINE statement except that they are non-directional. In other words, objects that are moved on trajectories defined by the NONDIRECPATH2D…SPLINE statement slide along the path instead of turning around on corners i.e. moving objects do not turn to face the direction of motion. Instead, objects retain their original configuration (orientation) that was current when motion started.

Syntax: \[\text{NONDIRECPATH2D} \ <\text{PathName}> \ \text{SPLINE} \ <\text{Points}>;\]
Example: \[\text{NONDIRECPATH2D} \ \text{PushRock} \ \text{SPLINE} \ '5,4' \ '7,9';\]

The SetTension Statement

This statement modifies the Tension parameter of the indicated path control point (i.e. knot) of a defined curved (spline) motion path trajectory. A knot’s tension controls how sharply the curve bends at that control point. The knot numbers begin from 1. The valid range for the Tension parameter’s value is from -1 to +1, with the default value being 0.

Syntax: \[\langle\text{CurvedPathName}\rangle.\text{SetTension} \ \langle\text{KnotNumber}\rangle \ \langle\text{Value}\rangle;\]
Example: \[\text{LoadToDump.SetTension} \ 1 \ 1;\]

The SetContinuity Statement

This statement modifies the Continuity parameter of the indicated path control point (i.e. knot) of a defined curved (spline) motion path trajectory. The Continuity parameter provides a smooth visual variation in the curve’s continuity at a knot. For instance, a Continuity value of zero yields derivative continuity at a knot point, whereas non-zero values yield discontinuities. The knot numbers begin from 1. The valid range for the Continuity parameter’s value is from -1 to +1, with the default value being 0.

Syntax: \[\langle\text{CurvedPathName}\rangle.\text{SetContinuity} \ \langle\text{KnotNumber}\rangle \ \langle\text{Value}\rangle;\]
Example: \[\text{LoadToDump.SetContinuity} \ 2 \ 0.5;\]
**The SetBias Statement**

This statement modifies the Bias parameter of the indicated path control point (i.e. knot) of a defined curved (spline) motion path trajectory. A knot’s bias controls how the direction of the curve at that knot changes by varying the weighted combination of one-sided derivatives at that control point. The knot numbers begin from 1. The valid range for the Bias parameter’s value is from -1 to +1, with the default value being 0.

Syntax: \(<\text{CurvedPathName}>\).SetBias <KnotNumber> <Value>;
Example: LoadToDump.SetBias 3 -0.5;

**The PATH2D…LINEAR Statement**

This statement is very similar in structure to the PATH2D…SPLINE statement. The only difference is that this statement defines a motion path trajectory by constructing connected linear segments passing through the indicated 2D planar coordinates. In that sense, this statement resembles the core VITASCOPE PATH statement. The constructed 2D piecewise-linear paths are then superimposed on the virtual terrain surface to define a 3D motion trajectory that drapes the terrain. The defined paths are dynamic in that they automatically adjust to any deformations that the underlying terrain might undergo without losing their spatial integrity, i.e. the path superimposition is performed in real-time during animation.

Syntax: PATH2D <PathName> LINEAR <Points>;
Example: PATH2D LoadToDump LINEAR '(10,10)' '(10,-30)' '(10,-130)' '(90,-130)';

Paths defined by this statement are directional, i.e. objects moved on defined paths automatically point in the direction of motion and adjust to the potentially uneven terrain (i.e. terrain following). In addition, the paths are accumulating i.e. VITASCOPE will temporarily stop a traveling object before the end of a path if it is blocked by another object. When the blocking object moves away (e.g. travels on a connected subsequent path), the stopped object resumes its motion until it reaches the end of the path or is
blocked again. If that was not the case, then multiple objects accumulating at the end of a path would come to rest “within” each other, which never occurs with physical objects.

**The NONDIRECPATH2D…LINEAR Statement**

Paths defined by this statement are identical to those defined by the PATH2D…LINEAR statement except that they are non-directional. In other words, objects that are moved on trajectories defined by the NONDIRECPATH2D…LINEAR statement slide along the path instead of turning around on corners i.e. moving objects do not turn to face the direction of motion. Instead, objects retain their original configuration (orientation) that was current when motion started.

**Syntax:**

```
NONDIRECPATH2D <PathName> LINEAR <Points>;
```

**Example:**

```
NONDIRECPATH2D PushRock LINEAR '(5,4)' '(7,9)';
```

**Describing Accurate, Variable-Speed Motion**

**The SetVelocityProfile Statement**

This statement sets the shape of the default velocity profile for objects traversing a defined motion path. In case no velocity profile shape is specified, objects move on defined paths with a constant average velocity for the entire duration of motion. The shape of the velocity profile is defined by specifying an arbitrary number of velocity-distance pairs.

The specified velocity values can span any positive numerical range and the corresponding indicated distances are the percentile (0 to 100) arc lengths along the path. The shape of the velocity profile is deduced by plotting the path’s percentile arc distance on the abscissa and the corresponding velocity values on the ordinate. When objects are set in motion on such paths (using the MOVE statement), VITASCOPE automatically scales the defined velocity profile shape to accommodate the durations of the communicated activity instances.
Syntax: \(<\text{PathName}>\).SetVelocityProfile <\text{VelPctDistPairs}>;  
Example: LoadToDump.SetVelocityProfile '(0,0)' '(2,20)'  
\[ '(2,75)' '(0,100)' \];

**The Redefined MOVE Statement**

This statement redefines the original VITASCOPE MOVE statement. In particular, the statement accepts additional arguments that allow simulation objects to override a path’s default velocity profile when moving on that path. The shape of the velocity profile is defined by specifying an arbitrary number of velocity-distance pairs. The specified velocity values can span any positive numerical range and the corresponding indicated distances are the percentile (0 to 100) arc lengths along the path. The shape of the velocity profile is deduced by plotting the path’s percentile arc distance on the abscissa and the corresponding velocity values on the ordinate.

This velocity profile overriding mechanism can, for instance, allow animation-authoring simulation models to specify a unique velocity profile per object motion that is a function of the properties (e.g. engine power, loaded mass) of the in-context simulation object (e.g. dumptruck) in a communicated instance of an activity (e.g. haul dirt). When a simulation object is instructed to move on a particular path in the specified simulation time units (TravelDuration), the current (path default or object overridden) velocity profile is appropriately scaled such that the time to traverse the path with that profile shape is equal to the communicated activity instance duration.

Syntax: \( \text{MOVE}\ <\text{ObjectName}> <\text{PathName}> <\text{TravelDuration}> [<\text{VelPctDistPairs}>] \);  
Example: MOVE Truck1 LoadToDump 230 '(0,0)' '(2.5,24)'  
\[ '(2.5,80)' '(0,100)' \];
Appendix L
C-Collide Add-On for VITASCOPE Reference

This appendix describes the statements implemented in the C-Collide add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Controlling Interference Detection and Response

The ACTIVATEOBJECT Statement

This statement turns on collision detection tests for all object pairs involving the specified scene object, i.e. C-Collide will start checking (at every instant) whether the specified object collides with any other scene object during animation.

Syntax: ACTIVATEOBJECT <ObjectName>;
Example: ACTIVATEOBJECT Truck1;

The DEACTIVATEOBJECT Statement

This statement turns off collision tests for all object pairs involving the specified scene object, i.e. C-Collide will stop checking whether the specified object collides with any other scene object during animation.

Syntax: DEACTIVATEOBJECT <ObjectName>;
Example: DEACTIVATEOBJECT Backhoe1;

The ACTIVATEPAIR Statement

This statement turns on collision tests for a specific pair of scene objects. Both the specified objects must have been activated by the ACTIVATEOBJECT statement in order for this statement to work. In general, since ACTIVATEOBJECT automatically activates all object pairs involving the specified object, this statement is only necessary if a specific object pair is subsequently deactivated and must be turned on again.
The ACTIVATEPAIR Statement
This statement turns on collision tests for a specific pair of otherwise active scene objects.

Syntax: ACTIVATEPAIR <Object1Name> <Object2Name>;
Example: ACTIVATEPAIR Truck1 Backhoe1;

The DEACTIVATEPAIR Statement
This statement turns off collision tests for a specific pair of otherwise active scene objects.

Syntax: DEACTIVATEPAIR <Object1Name> <Object2Name>;
Example: DEACTIVATEPAIR Truck1 Backhoe1;

The RESPONSEMODE Statement
This statement specifies whether C-Collide must interactively report any detected collisions to the output console and wait for a user response; or record them silently to a specified disk log file that can be inspected at a later time. The choice of the Mode argument is between INTERACTIVE and SILENT.

Syntax: RESPONSEMODE <Mode> [LogFile];
Example: RESPONSEMODE INTERACTIVE;
Example: RESPONSEMODE SILENT CollisionReport.TXT;

Defining Abstract Scene Objects
The ABSTRACTOBJECT Statement
This statement allows the definition of abstract, invisible objects for interference detection purposes. In particular, C-Collide can detect interferences between concrete, visible objects as well as abstract entities such as a region of space. This, for instance, allows the definition of hazardous or restricted work areas and automatically detects any intrusions (i.e. interferences) that might occur during the course of animation.
The semantics of this statement are very similar to the core CREATE statement. The only difference is that the “created” object in this case will not be visible on the screen (even after it has been placed) during animation. C-Collide will however silently monitor any interference other objects encounter with the defined abstract, invisible object.

Syntax: \texttt{ABSTRACTOBJECT <ObjectName> <ClassName>;}
Example: \texttt{ABSTRACTOBJECT HazardArea Box;}

\textbf{Core VITASCOPE Statements Redefined in C-Collide}

C-Collide redefines several core VITASCOPE animation statements. These include the CLASS, ORIENTCLASS, CHANGECLASS, CREATE, ATTACH, ATTACHNOSCALE, PLACE, and DESTROY statements. However, the syntax and the semantics (e.g. number and significance of the arguments) of the original statements are unaltered. C-Collide redefines these statements so that it can perform additional computations required to initialize and manage the interference detection engine. Since the usage of the redefined statements is unchanged, that information (described in Appendix B) is not repeated here.
Appendix M
ExcelWorks Add-On for VITASCOPE Reference

This appendix describes the statements implemented in the ExcelWorks add-on for VITASCOPE. The syntax and semantics of each statement are described with examples of their usage.

Creating and Updating Dynamic Excel Charts

The XYCHART Statement
This statement creates and initializes a new Excel chart sheet. A single data series with the specified name is created and initialized with the specified first data point. This statement can be called several times to create more than one chart sheets, i.e. more than one charts can be active and be manipulated independently at the same time.

Syntax:     XYCHART   <ChartName> <ChartTitle>
            <X-Label> <Y-Label>
            <FirstSeriesTitle>
            <X1> <Y1>;

Example:    XYCHART   Chart2 'Available Concrete Statistics'
            'Simulation Time' 'Concrete (M3)'
            'Amount in Pier Hopper'
            0 0;

The AddSeries Statement
This statement creates a new data series with the specified name, initializes the series with the specified first data point, and adds the series to an existing Excel chart sheet created earlier with the XYCHART statement. This statement can be called several times to add more data series to a chart, i.e. a chart can consist of more than one data series that can be manipulated independently at the same time. The maximum number of data series allowed in a single Excel chart sheet is 255.
Syntax:  
<ChartName>.AddSeries <NewSeriesTitle>

<X1> <Y1>;

Example:  Chart2.AddSeries 'Avg. Concrete in Hopper' 0 0;

**The AppendSeries Statement**

This statement adds an additional data point to an existing data series in an existing Excel chart sheet. The maximum number of data points allowed in a single data series is 32,000. In addition, the maximum number of data points (in all data series) that can be plotted in a single Excel chart sheet is 256,000.

Syntax:  
<ChartName>.AppendSeries <SeriesTitle>

<Xn> <Yn>;

Example:  Chart2.AppendSeries 'Amount in Pier Hopper'

2639.96 5.12;

Example:  Chart2.AppendSeries 'Avg. Concrete in Hopper'

3334.26 2.35;
Vita

Vineet Rajendra Kamat was born in Margao, India on January 11, 1977. He earned a Bachelor of Engineering (B.E.) degree in Civil Engineering at Goa University, India in 1998 and a Master of Science (M.S.) degree in Civil Engineering at the Virginia Polytechnic Institute and State University in 2000. He continued his education at the Virginia Polytechnic Institute and State University, and earned a Doctor of Philosophy (Ph.D.) degree in Civil Engineering in 2003.