Computer-Aided Civil and Infrastructure Engineering **24** (2008) 1–13

Algorithm for Accurate Three-Dimensional Scene Graph Updates in High-Speed Animations of Previously Simulated Construction Operations

Prasant V. Rekapalli, Julio C. Martínez*

School of Civil Engineering, Purdue University, West Lafayette, IN, USA

&

Vineet R. Kamat

Department of Civil & Environmental Engineering, University of Michigan, Ann Arbor, MI, USA

Abstract: *Visualization of construction operations is an important technique to communicate the logic of simulation models in detail. Early efforts resulted in a scene graph and frame update algorithm that was capable of converting discrete information from simulation models into smooth and continuous 3D animations. That algorithm did not account for high speed or concurrent animation because the need to do so was not anticipated. Recent advances in computing power and an interest in using the technology for next generation applications now demand accurate high speed and concurrent animations. This article presents the design of the original algorithm at a previously undocumented level of detail and specificity, and that allows for the analysis of its shortcomings when used at high speeds or concurrently with simulation. Two subsequent but still inadequate designs of the algorithm are also presented and analyzed in detail so that they can serve as an illustration of the path toward the final design and place it in proper context. The article concludes with the final design and evaluation of the algorithm, which is accurate at very high animation speeds and supports concurrent animation of simulation models.*

*To whom correspondence should be addressed. E-mail: *julio@purdue.edu*.

DOI: 10.1111/j.1467-8667.2008.00565.x

1 INTRODUCTION

Discrete-event simulation (DES) can add significant value to the design of construction operations. However, this value is only realized if the insights gleaned from the DES model are used in making decisions and increase understanding (i.e., they are credible) (Law and Kelton, 2000). In order for simulation models to be credible, their internal logic must be presented clearly to domain experts and decision makers. Animation in 3D is a very effective means of communication that allows errors in logic to be easily identified and corrected, and planning groups to participate in discussions aimed at improving the plan (Biles and Wilson, 1987; Cox, 1988; Robinson, 1997; Henriksen, 1998; Tucker et al., 1998; Jain, 1999; Rohrer, 2000; Law and Kelton, 2000; Kamat and Martinez, 2003). Kamat and Martinez (2002) present a very detailed and comprehensive literature review on the use of visualization techniques in construction.

For 3D animation to be effective, it must (1) be smooth and continuous to provide a good viewing experience, (2) maintain a constant ratio of animated time to viewing time (hereafter called *animation speed*) to validate the relative speeds of activities, and (3) be temporally and spatially accurate.

For basic effective use of 3D animation, it is only necessary for these characteristics to hold when the animation is post processed (i.e., the animation takes place after the entire model has been simulated), and when the animated speed is modest ("modest" here is subjective and depends on many factors).

More forward-looking applications require these characteristics to hold when animation is concurrent with the simulation (i.e., both the simulation and animation take place in parallel and advance their simulated and animated time simultaneously), and when animation speed is very high (e.g., to achieve an effect similar to time-lapse photography).

The process of animation involves a scene graph that organizes the elemental 3D objects that make up the universe being visualized, and the rendering of the scene-graph for presentation in a display device (i.e., a frame update). To animate what happened in the corresponding simulation, it is necessary to update the scene graph to reflect the movements, rotations, and transformations of the elemental 3D objects that make up the scene; and to subsequently render the scene graph. The continuous cycling of these two steps at rates of 10 or more per second gives the illusion of *apparent motion* (Anderson and Anderson, 1993). Kamat and Martinez (2002) explain in detail how a scene graph is organized and modified to reflect changes in position and orientation of elemental 3D objects. They also explain how a straight-line language, authorable by end-user programmable applications (such as a discrete-event simulation system), can describe the movements and transformations with temporal and spatial accuracy. Kamat and Martinez (2002) present at a coarser level of detail an algorithm for scene graph and frame update that enables effective 3D animation at the basic level (see above) using the straight-line language.

In DES, time advances in discrete, uneven time intervals. It assumes that the state of the system being modeled changes only at these points in time, and it is only then that the simulation system can write animation instructions (statements) in the straight-line language. Because animation should be a continuous, smooth process, the main task of the algorithm is to achieve this from discrete, unevenly spaced information. Please refer to Kamat and Martinez (2002) for an in-depth discussion of this topic.

This article first describes the algorithm presented in Kamat and Martinez (2002) in finer detail, and then analyzes its performance under the more demanding constraints that exist when animation is concurrent with simulation and/or at very high speeds. What constitutes as a "high" or "low" or "moderate" animation speed is model and user dependent (i.e., the time units used to develop the animation model, and the content/purpose of the animation model itself), and therefore these terms are not quantified. The article then presents several successive refinements to the algorithm (and their analysis) that were needed to enable effective 3D animation under the more stringent constraints. It is important to describe how the algorithm evolved to satisfy more demanding constraints because the lessons learned from the process are valuable to others involved in the design of similar algorithms.

2 PERFORMANCE OF SCENE GRAPH AND FRAME UPDATE ALGORITHMS

The performance of a scene graph update algorithm is judged in terms of (1) the frame rate achieved, (2) animation smoothness, (3) maintenance of a constant viewing time to animation time ratio, and (4) the ability to maintain an accurate scene.

Frame rate refers to the number of still images that make up a unit of time in a video/film, and is typically measured in *frames per second* (*fps*). In computer animation, frame rate is the number of animation loops (where each animation loop results in an image being drawn on the display device) achieved in one time unit; and is also measured in frames per second.

Animation loop times directly affect the frame rate that can be achieved. Smaller loop times result in higher frame rates. An animation loop time depends on (1) the computing resources available (CPU, graphics processor, memory), (2) the animation complexity, and (3) the number of animation instructions processed.

As the capability of the computing resources increases, the time required to process the animation, update and render is shorter, and thus increases the achievable frame rates.

Animation complexity is a function of (1) the number of objects in an animation scene, (2) the number and type of simultaneous moving objects in the scene, and (3) the level-of-detail of the elemental CAD models used in the animation. When animation complexity increases, the time required to update and render the scene increases, and consequently the achievable frame rate goes down.

Periodically, the algorithm has to process a set of animation instructions. This is required to update the content of the animation (i.e., introduce new virtual objects or remove existing virtual objects, define new object motions, etc.). When this happens, the animation loop times also increase. When and how many animation instructions are processed depends on the algorithm being used.

3 DESIGN ITERATION 1

The initial algorithm was implemented and extensively tested in pre-release versions of the dynamic construction visualizer (DCV) (Kamat and Martinez, 2001), and its design evolution is presented in Kamat and Martinez (2002). The algorithm contains four distinct processes shown in Figure 1 and explained below: (1) trace file parser process (TFPP), (2) animation time advance process (ATAP), (3) position orientation update process (POUP), and (4) cull draw frame process (CDFP). The TFPP, POUP, and ATAP processes control the scene graph update, and the CDFP process controls the frame update.

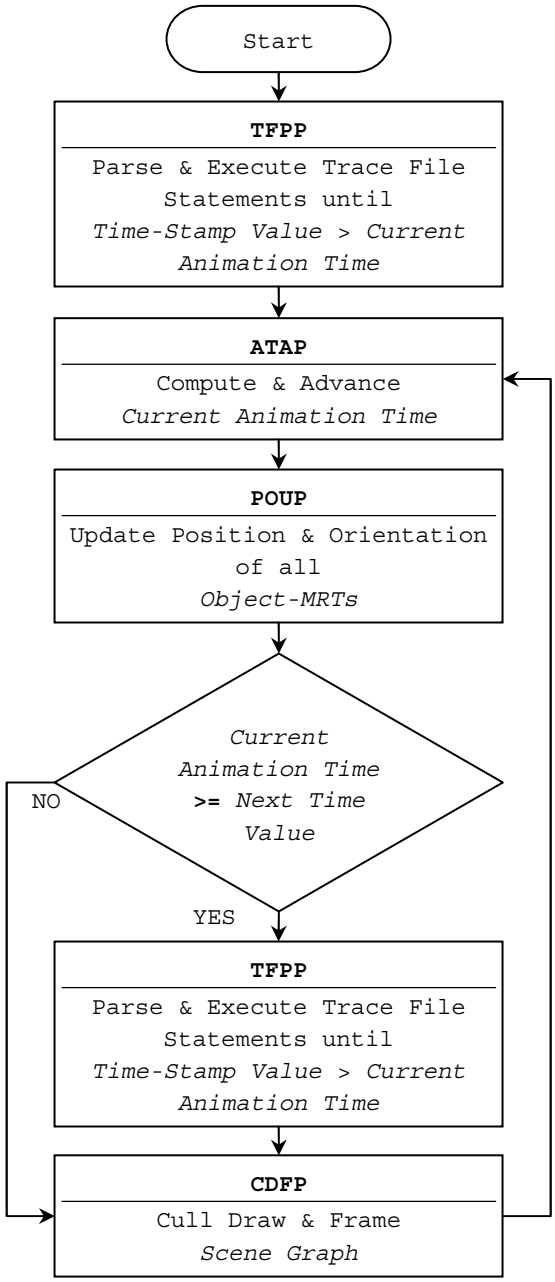

Fig. 1. Scene graph and frame update algorithm from design iteration 1 (Kamat and Martinez, 2002).

3.1 Trace file parser process

The scene graph and frame update algorithm is designed to work with instructions that describe (1) the creation or destruction of objects; (2) instantaneous changes to their state, including position, orientation, scaling, and appearance; (3) attachment and detachment of one object to another; and (4) changes to objects that occur over a period of time such as movement over a path, rotation, and transformation (scaling, bending). Every instruction is associated with a *timestamp value* that indicates when, in simulated time, the instruction takes place. In DCV, for example, instructions are represented by statements, and the "TIME" statement defines the timestamp value for all the animation statements that follow it (until a subsequent TIME statement establishes a new timestamp).

The *trace file parser process* (*TFPP*) (hereafter referred to as the *parser*) interprets instructions. Instructions that indicate instantaneous action result in changes to the scene graph that take place immediately. For example, an instruction to place an object at a specific position would immediately change the translation node of the scene graph that controls its location. Instructions that indicate actions that take place over time are registered with the POUP (see below). Instructions of this type include those that indicate noninstantaneous movement, rotation, or transformation. An instruction may indicate that the hook line of a crane, for example, should reduce its length over a period of time to animate the lifting of a load (the hook line may be attached to a boom on one end, and the load may be attached to the hook line at the other).

The *parser* interprets instructions sequentially until it encounters a timestamp that is larger than the current animation time. This new timestamp becomes the *next time value*, and provides a reference point for reentry into the *parser*. The algorithm does not reenter the *parser* unless the current animation time is greater than or equal to the next time value.

3.2 Animation time advance process

The *animation time advance process* (*ATAP*) (hereafter referred to as the *timer*) is responsible for advancing the animation clock. The animation clock in theory is advanced by discrete amounts based on the time elapsed since the last time advance (in real clock time units)

and the animation speed. This animation time advance mode results from the use of a variable frame rate. The rationale for this is explained in great detail in Kamat and Martinez (2002). The new animation clock time is calculated using the formula:

$$\text{NewTime} = \text{OldTime} + (\text{ElapsedTime} * \text{AnimationSpeed})$$

Animation speed is the user-controlled rate of increase of the animation clock per unit of real time. For example, an animation speed of 10 implies the animation clock will advance by 10 time units for every 1 time unit of real time.

Elapsed time values are usually very small and therefore the animation time advances are also usually small. This creates the illusion that time is advancing smoothly.

3.3 Position orientation update process

The *position orientation update process* (*POUP*) (hereafter referred to as the *updater*) keeps track of the various changes to objects that occur over a period of time. This includes movements, rotations, transformations (referred to as *object-MRT* in the rest of the article) or any other gradual change (e.g., an object could change its color or transparency gradually over some time period). When called, the *updater* updates the scene graph to reflect the state of the objects registered with it. For example, if an object is registered to move at constant speed from position (1,1,1) to position (2,1,1) in 4 time units that started at time 50 (the timestamp in effect when the object was registered with the *updater*), and the animation clock time is 51, its position would be updated to (1.25,1,1). The *updater* also completes the changes indicated by instructions that have expired and then de-registers them.

By design, the *updater* maintains only one object-MRT instance per type per object. For example, an object at any point in time can have only one vertical rotation registered with the *updater*. This does not preclude the same object from having other object-MRTs simultaneously registered with the *updater*. The implication of this design feature is that when the *parser* registers a new object-MRT with the *updater*, if the same object-MRT (with different parameters) is already registered in the *updater*, the new object-MRT overrides the old one. This design feature was chosen for several reasons:

1. An animation scene is created by importing several CAD models, some of which are static objects (e.g., buildings, terrain features). The straight-line language of DCV does not differentiate between static and dynamic objects. It is computationally inefficient to assume that all objects are dynamic because this would require several more parameters to be maintained by each object. Instead, all objects are initially assumed to be static, and when an object-MRT is encountered, the algorithm dynamically converts that object into a dynamic type. The dynamic object then maintains the object-MRT's details, while being registered with the *updater*.
2. The algorithm must be capable of handling a situation where two partially overlapping object-MRTs of the same type on the same object are registered (i.e., a second object-MRT is registered while another object-MRT of the same type on the same object is already in progress). This is important when considering future applications. See example below.
3. An application currently under design involves the use of concurrent simulation-animation to enable interactive simulation control. In this application, the concurrent animation must present the current state of the DES; any changes implemented into the simulation state must be captured by the animation. For example, to reflect a change in the duration of an ongoing activity, the animation must be capable of modifying the duration of object-MRTs that reflect the process within the simulation activity.
4. In other future applications, object-MRT information available might be incorrect, incomplete, or changing over time. This could be either by desire or limitation. For example, when using this algorithm to animate in real time a construction work site, it would be impossible to obtain the actual duration of any activity that is in progress until after the fact. This can be overcome by using a prediction strategy where animation object-MRTs (that represent an ongoing activity) are initially populated using predicted durations (e.g., using past data) and these durations can be updated from time to time based on the actual data obtained from the construction site in real time.
5. Another advantage of this design feature is its ability to capture errors produced in the simulation model. For example, if the situation of two partially overlapping object-MRTs of the same type on the same object were to occur due to an error in the simulation model, the current strategy of handling the situation would provide a visual notification of the error to the end user (because the animation viewed would not conform with the end user's expectations).

The authors believe this design feature is of importance and must be preserved.

3.4 Cull draw frame process

The *cull draw frame process* (*CDFP*) (hereafter referred to as the *renderer*) renders the scene graph to the display device. Before rendering, the *renderer* first calculates the position and/or orientation of the scene camera, which determines the needed culling. Camera position and orientation changes are the result of user interaction during visualization (i.e., when the user navigates) and/or when the camera is attached to an object that is also moving.

3.5 Elapsed time length

The elapsed time values are the cumulative times used by the *updater*, *parser*, and *renderer* between successive *timers*. The *renderer* times are influenced by the complexity of the scene, the camera position processing time, and the number of viewing windows. The *parser* times depend on the number of statements that need to be parsed and are usually negligible in comparison to the *renderer* times; and therefore usually do not affect the frame rate performance. The *updater* times depend on the number of moving objects and are usually negligible in comparison to the *renderer* times. The *updater* times can be significant when modeling some complex objects (e.g., fuzzy objects like concrete, water, etc.).

3.6 Testing, analysis, and limitations

The initial algorithm appeared to meet the performance requirements while displaying an accurate scene at all times when tested at relatively low animation speeds. However, when higher animation speeds were used and the animation ran for long periods of time, certain objects were displayed at incorrect positions and orientations. These errors increased as the animation continued.

The *updater* is a critical step in maintaining the accuracy of the scene. By design, certain object-MRTs may be cumulative in nature (e.g., successive rotations). The instructions that affect such movements assume that the current state of the object is accurate, and simply indicate how the object should change from that point on.

For illustration purposes, consider the boom of an excavator attached to a cabin. For animation purposes, the boom may be rotated vertically up and then down in a continuous fashion. A portion of a DCV trace file that corresponds to one cycle may look as follows:

```
TIME 10;
VERTORIENT ExcBoom 45;
...
TIME 20;
ROTATE ExcBoom VERT -30 2;
TIME 22;
ROTATE ExcBoom VERT 30 2;
TIME 30;
```

ExcBoom's (i.e., the excavator's boom) current vertical orientation is set to 45 degrees prior to the rotations. The first "ROTATE" statement indicates that the boom should rotate –30 degrees starting at animation time 20 and ending two time units later (at time 22). A call of the *updater* at time 22 would thus calculate the vertical orientation of the boom to be $45 - 30 = 15$ degrees. Similarly, and noting the second "ROTATE" statement, a call of the *updater* at time 24 would calculate the vertical orientation of the boom to be 45 degrees, the same as at time 20.

This initial algorithm did not contemplate the possibility of a situation in which a single call to the *parser* would process instructions with more than one unique timestamp. For example, the assumption was that the next timestamp found during the *parser* would have a larger value than the current animation clock. In other words, the real time elapsed between successive calls to the *parser*, multiplied by the animation speed would be smaller than the difference between two unique timestamps.

Due to the laws of probability, however, it is possible for two events in a simulation to occur within a very small time interval. The assumption was that animation instructions resulting from those very close events would not involve the same object and the same type of change.

However, when animation speed is high, and the frame rate is low, this situation does happen. The consequence is that errors take place in calculating the object-MRT during the *updater*, and that these errors are cumulative.

For illustration purposes, consider the above sample case, and a situation where the animation speed is 100 and the frame rate is 25 fps. In this case, the elapsed time is $1/25^{th}$ of a second, and the corresponding animation time increase is 100 times that value, or 4 seconds. Assume that time advances from 19 seconds to 23 seconds. Note that the next time value resulting from the last *parser* is 20. The following is a walkthrough of the *parser* process:

- The *parser* is called when the animation time is 23.
- The *parser* processes the first "ROTATE" statement and registers with the *updater* that ExcBoom is rotating vertically at the rate of –15 degrees per second.
- Because the next timestamp encountered, 22, is less than the current animation clock, the *parser* continues and processes the second "ROTATE" statement.

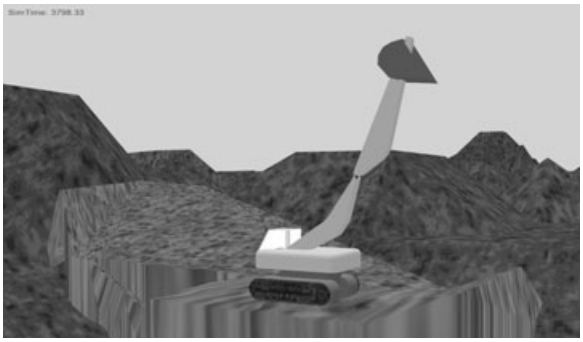

Fig. 2. Scene distortion of an excavator model.

It registers with the *parser* that ExcBoom is rotating vertically at a rate of +15 degrees per second. This effectively overwrites the vertical rotation speed of ExcBoom that was installed by the first "ROTATE" statement.

- The *parser* ends when it encounters the "TIME 30" statement.
- A subsequent *updater* will disregard the first "ROTATE" and will assume that at time 22 the vertical orientation of the boom is 45 degrees instead of 15 degrees.

In practice, the errors are much smaller in magnitude, but do occur with some frequency when animation speed is high and frame rate is low. The accumulation of the small errors becomes visible and results in the scene looking distorted. Figure 2 shows the effect of scene distortion on an excavator model (note how the excavator's boom, stick, and bucket are at such unrealistic angles).

During the early testing phase, because of the very low frame rates achieved with the algorithm, viewing animations at higher speeds caused a loss of apparent motion (i.e., the changes in object-MRTs was so large between each frame that the animation looked like a series of still images). When the testing shifted to a better computing platform, this effect of scene distortion was observed. Increases in available computing performance have made it possible to view animations at higher and higher speeds without loosing the illusion of apparent motion.

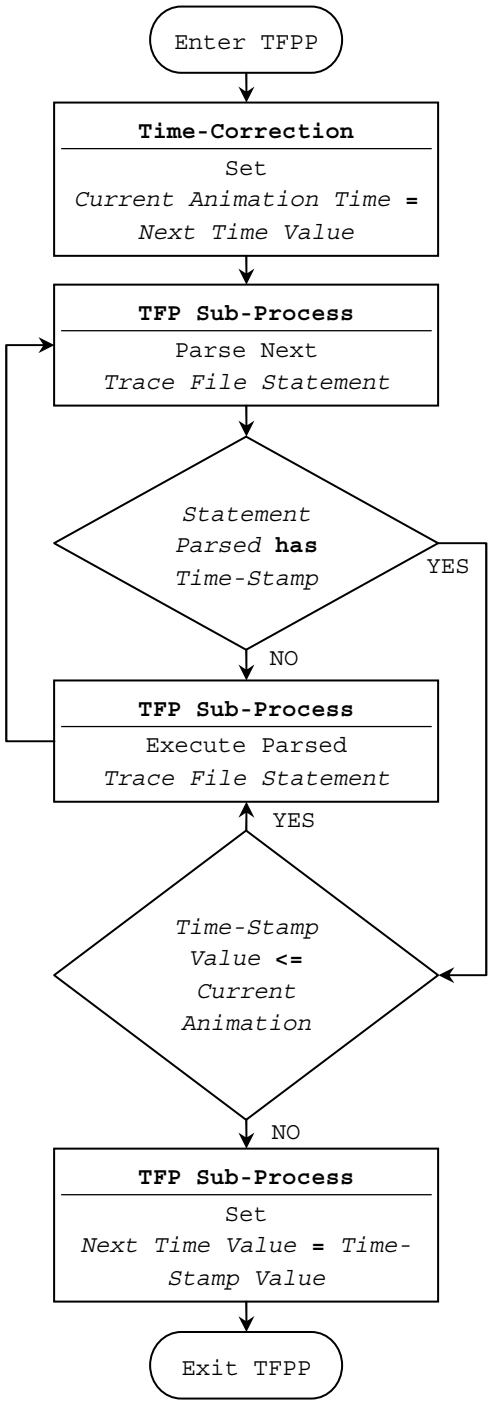

Fig. 3. TFPP of the scene graph and frame update algorithm from design iteration 2.

4 DESIGN ITERATION 2

To eliminate scene distortion, a *time correction* was introduced as the first step in the *parser*. The role of the time correction is to bring back the current animation time to match the next time value (see Figure 3). This limits the *parser* to parsing and executing statements that have the same timestamp value. Doing so eliminates the possibility of registering two object-MRTs of the same type on the same object within the same *parser*, at the expense of maintaining a constant ratio of viewing time to animation time. The time correction

effectively eliminates the chance for any scene distortion to occur.

The algorithm from the second iteration was implemented in DCV and in early versions of VITASCOPE (Kamat, 2003) (an extension of DCV). This algorithm has not been presented in detail in prior literature.

4.1 Testing, analysis, and limitations

The algorithm from design iteration 2 met the performance requirements while maintaining an accurate scene when used to view animations of construction operations at low to moderate animation speeds. When VITASCOPE, implementing this algorithm, was used for animation at high speeds, two new limitations caused by the time correction became evident: (1) animation speed deviation and (2) animation speed variability.

4.1.1 Animation speed deviation. The time correction within the *parser* causes the algorithm to deviate from the user-defined animation speed. As noted earlier, the time correction in the *parser* always reduces the animation time to the next time value. This causes an increase in the total animation time actually viewed, which causes a reduction in the animation speed achieved; and can be calculated as follows:

T_a Actual length of an animation model
T_r Cumulative time correction incurred during a given animation model's run
A_s User-defined animation speed
A_e Effective animation speed achieved for a given user-defined animation speed on a given animation model

$$\frac{T_a}{A_e} = \frac{T_a + T_r}{A_s} \text{ or } \frac{A_e}{A_s} = \frac{T_a}{T_a + T_r} \tag{1}$$

According to Equation (1), the effective animation speed achieved (as a ratio of the user-defined animation speed) decreases as the cumulative time correction incurred in an animation run increases (T_r). The cumulative time correction for an animation model depends on:

1. *The number of unique timestamps*. Time correction occurs only on entry to the *parser*, which occurs for every unique timestamp in the animation model. When using this algorithm, the minimum number of animation loops required to visualize the entire animation equals the number of unique timestamps in the animation.
2. *The user-defined animation speed*. Higher user-defined animation speeds produce larger time advances at the *timer*, which causes larger time corrections in the *parser*. Importantly, T_r increases at a lower rate than A_s. The relationship between the rates of change of T_r and A_s depends on the frame rate achieved. Hence, for a given animation model and frame rate, as A_s increases, even though the ratio of A_e to A_s decreases, the magnitude of A_e increases.
3. *The frame rate achieved*. Lower frame rates produce larger time advances at the *timer*, which causes larger time corrections in the *parser*.

The magnitude of A_e, however, has an upper limit (i.e., a maximum achievable effective animation speed), which can be calculated as follows:

N_n Number of unique timestamps in an animation model
T_e Average elapsed times achieved for a given model using given computing resources

$$A_e[\max] = \frac{T_a}{T_e * N_n} \tag{2}$$

Hence, the maximum effective animation speed is achieved when the algorithm enters the *parser* in every animation loop. Equation (2) shows that an increase in frame rate (i.e., lower T_e values) increases the maximum achievable effective animation speed ($A_e[\max]$).

4.1.2 Animation speed variability. Animation speed deviation, by itself, is not a critical limitation. As mentioned earlier, an important performance measure is the ability to maintain a constant animation time to viewing time ratio (i.e., maintain a constant animation speed). Hence, animation deviation would have been an acceptable limitation if the algorithm maintained a constant animation speed at the effective speed achievable.

Further testing and analysis, which included collection of time correction data from sample models, revealed that the algorithm failed to maintain a constant animation speed. Instead, the algorithm produced variability in the animation speed experienced during the animation run. This variability increased to unacceptable levels at higher animation speeds.

A constant speed implies that a plot of real time versus animated time should be a perfectly straight line passing through the origin and with a positive slope. This straight line, however, is disrupted at every time correction (i.e., the plot deviates from being a straight line). In other words, the plot maintains a straight line (with a slope equal to the user-defined animation speed) between consecutive time corrections.

The ratio of the number of animation loops required to visualize a model to number of animation loops where deviation is encountered can serve as one

measure of animation speed variability. The number of animation loops where deviation occurs equals the number of unique timestamps (N_n). The number of animation loops required to visualize the entire model can be shown to be:

N_a Number of animation loops required to visualize an entire animation model

$$N_a = \frac{T_a}{T_e * A_e} \tag{3}$$

A higher ratio of N_a/N_n implies that a greater portion of the real-time to animated-time plot maintains a straight line. At the maximum effective animation speed, this ratio reaches its lower limit value of 1, and the corresponding real time to animated time plot at this ratio no longer contains straight line portions.

Hence, animation speed variability depends on (note Equation (3) and above discussion):

1. The number of unique timestamps in the animation model (N_n).
2. *The user-defined animation speed*. Increase in the user-defined animation speed increases the magnitude of the effective animation speed, which reduces the number of animation loops required to visualize the entire model.
3. *The frame rate achieved*. Lower frame rates reduce the number of animation loops required to visualize the entire model.

4.1.3 Sample model analysis. To provide a better understanding of the effects of animation speed deviation and variability, time correction data were collected from an animation model of an earthmoving operation at different animation speeds. The data were collected for the first 3,000 time corrections encountered when running the model. The animation speeds used were as follows:

1. A_s range 10 to 100 in steps of 10
2. A_s range 100 to 1,000 in steps of 100
3. A_s range of 1,000 to 10,000 in steps of 1,000

All animation runs used the same computing resources, in this case a 3.73 GHz processor, 256 MB video card, and 4 GB memory.

Using the data collected, for each animation speed, the following results were obtained:

- The ratio of A_e to A_s (plotted in Figure 4 as a percentage).
- The magnitude of A_e (plotted in Figure 4).
- The ratio of N_a to N_n (plotted in Figure 5).
- The "straightness" of the real time versus animated time plot. For each plot (for each A_s) the square of the Pearson product moment correlation coefficient (RSQ) was calculated. Figure 5 also shows the plot for the value of (1–RSQ)∗10^4 for the different A_s.

Figure 6 shows the plot of real time versus animated time for the earthmoving model at an animation speed of 300. The maximum effective animation speed for the earthmoving model is 206 (calculated using

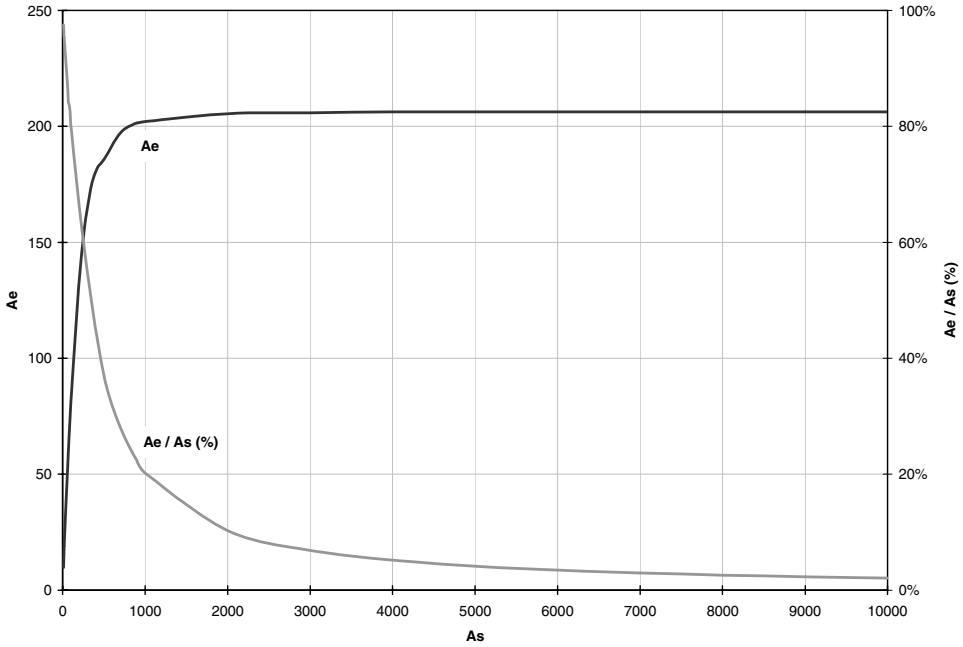

Fig. 4. Animation speed deviation results for sample earthmoving model.

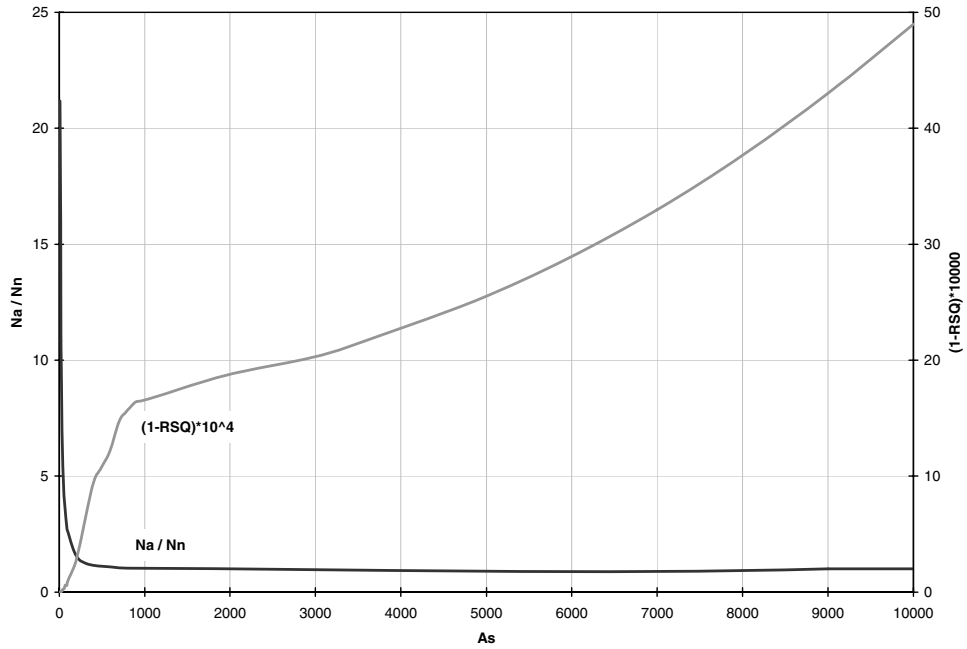

Fig. 5. Animation variability results for sample earthmoving model.

Equation (2) where $N_n = 3{,}000$, $T_e = 0.0156$, and $T_a = 9{,}666.61$).

4.2 Critical animation speed

The earlier discussion notes that the second algorithm presents certain problems as the animation speed used to visualize the model increases. The *critical animation speed* is the animation speed value at which animation speed deviation and/or animation speed variability are noticeable. This is a subjective value because people have different capabilities in detecting/noticing animation speed deviation or variability.

As discussed earlier, there is an upper limit to the effective animation speed achievable (206 for the earthmoving model) for any given model. General

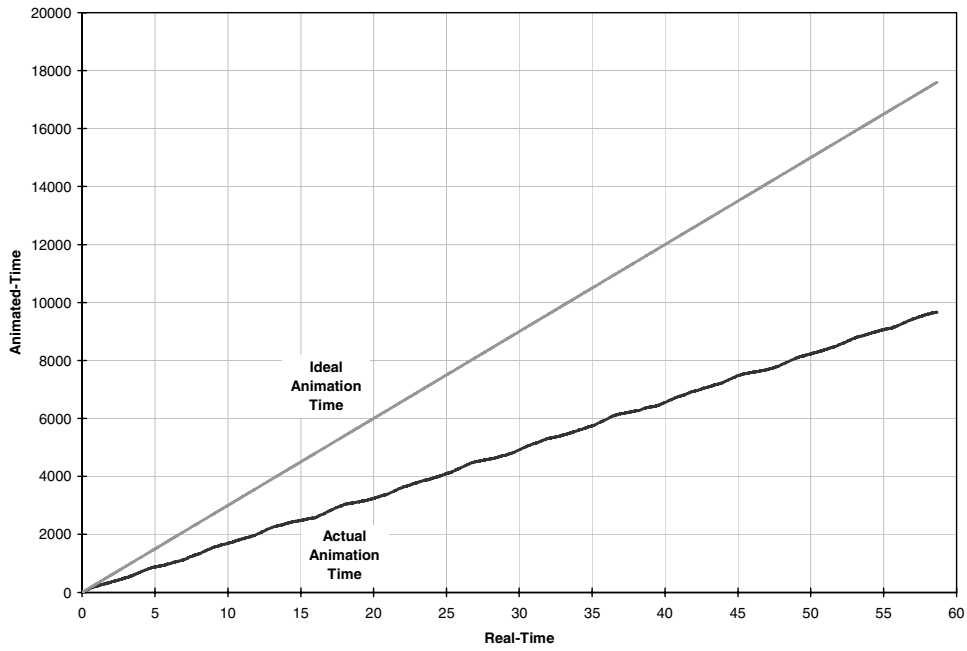

Fig. 6. Actual and ideal animation time plots for the earthmoving model at animation speed of 300.

observations show that when the maximum effective animation speed is reached, animation deviation is very noticeable. Hence, the minimum animation speed at which the maximum effective animation speed is achieved is one way to define the critical animation speed (around 1,000 for the earthmoving model based on this criteria).

The critical animation speed is dependent on the performance of the algorithm, which is dependent on several other factors (noted earlier). The critical animation speed is also dependent on the time scale used in the animation model. For example, an animation model with its time value units in hours, an animation speed of 1 would imply for every 1 second of real time the animation clock is advanced by 1 hour (i.e., 3,600 seconds). Hence, an animation speed of 1 in this case could be well above the critical animation speed.

5 DESIGN ITERATION 3

It is apparent that the time correction contained within the *parser* is the cause for deviation and variability in observed animation speed. At high animation speeds, these issues become more noticeable as they occur with greater magnitude. These problems cannot be avoided, and hence there is a need to modify the existing algorithm without the time correction while also preventing the scene distortion problem.

This was achieved in the third design iteration, where the scene graph and frame update algorithm and the *parser* process within it were modified as shown in Figures 7 and 8 respectively.

The *updater* was introduced into the *parser* such that it takes place before the *parser* interprets instructions with a new unique timestamp. Scene distortion is caused when multiple object-MRTs of the same type on the same object are parsed and executed within a single *parser*. Scene integrity is maintained by ensuring that all existing object-MRTs are updated before the *parser* interprets instructions that have a new unique timestamp.

Consider the sample DCV statements presented earlier, where the animation clock advances from 19 to 23. The following is walkthrough of the algorithm (starting with the *parser*):

1. The *parser* parses the "TIME 20" statement.
2. The *updater* is called from within the *parser* at time 23.
3. The *parser* parses and executes the first "ROTATE" statement, which registers with the *updater* that ExcBoom must rotate vertically by –30 degrees with start and end times of 20 and 22, respectively.
4. The *parser* parses the "TIME 22" statement.

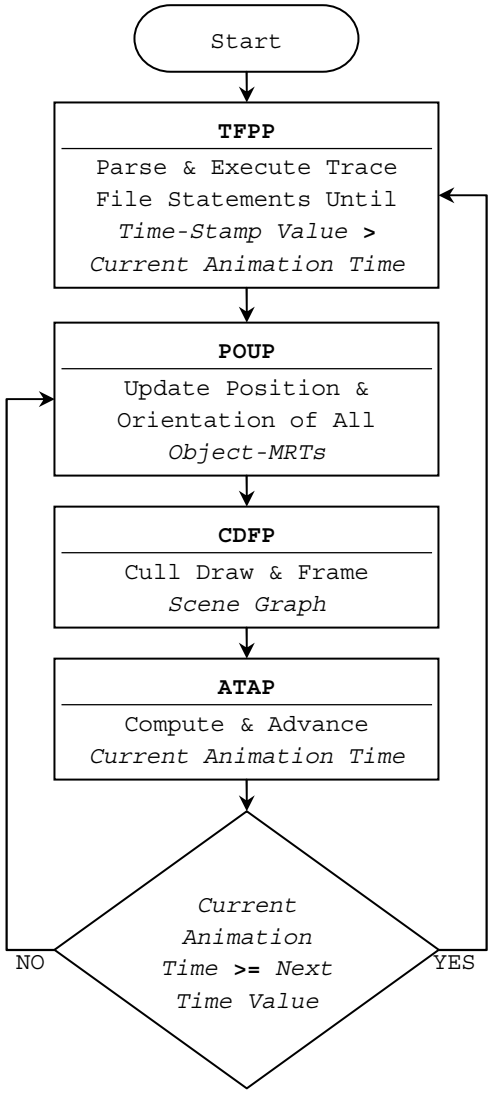

Fig. 7. Scene graph and frame update algorithm from design iteration 3.

5. The *updater* is called from within the *parser* at time 23. Within the *updater*, the registered instruction about ExcBoom's vertical rotation is updated, and is calculated to be 15 degrees. Also, because this instruction has expired, it is deregistered from the *updater*.
6. The *parser* parses and executes the second "ROTATE" statement, which registers with the *updater* that ExcBoom must rotate vertically by 30 degrees with start and end times of 22 and 24, respectively.
7. The *parser* parses the "TIME 30" statement and then exits.
8. The *updater* from the scene graph and frame algorithm is called at time 23, which calculates and updates ExcBoom's vertical rotation to be 30 degrees (15 + 30/2).

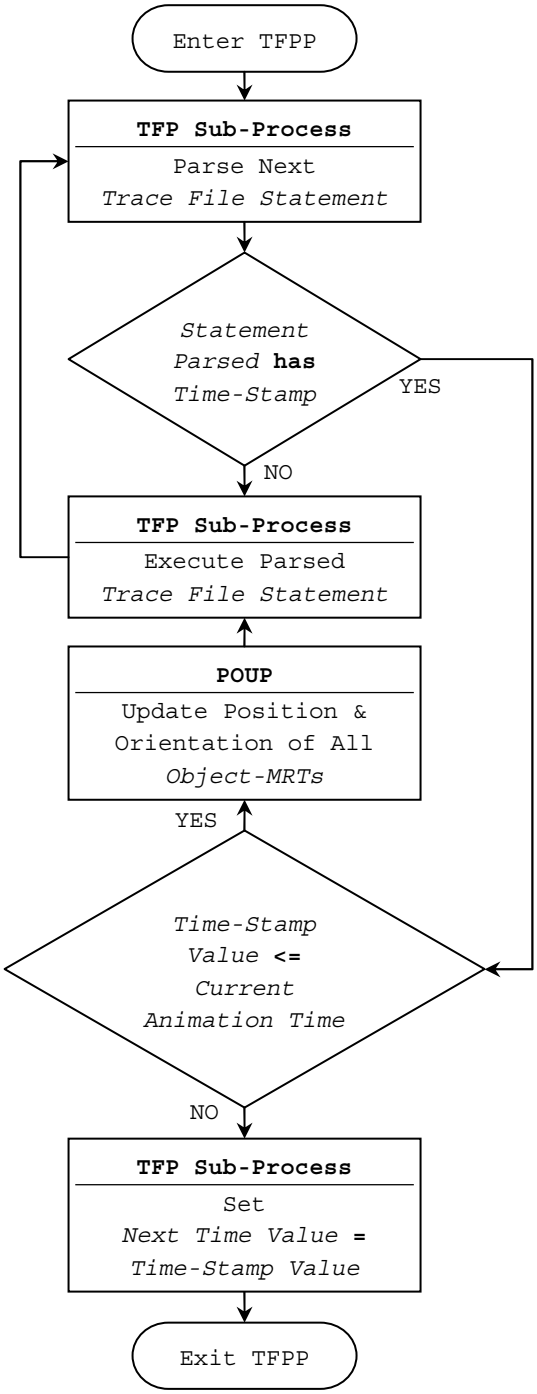

Fig. 8. TFPP of the scene graph and frame update algorithm from design iteration 3.

Note that this algorithm version maintains the design feature of the first algorithm in that a new object-MRT overrides any existing object-MRT of the same type on the same object that was previously registered with the *updater*. The difference here is that the previous object-MRT is partially updated before being overridden.

5.1 Testing, analysis, and limitations

By removing the time correction present in design iteration 2, it was possible to maintain a near perfect constant ratio of animated to real time. The term "near perfect" is used because the animation time ultimately advances in discrete uneven time steps (that depends on the frame rate and animation speed); and therefore a plot of ideal animation time to real animation time will never be an exact match. This algorithm iteration, like the first iteration, does not have any time correction and therefore does not exhibit any animation speed deviation or variability, but unlike the first iteration, this algorithm does not exhibit any scene distortion. Although the algorithm at this stage was adequate, the frame rates achievable when viewing complex animations were lower than with the prior iterations when modeling complex objects that significantly increased the *updater* time. The next iteration in the design involved changes geared to improve frame-rate performance.

6 DESIGN ITERATION 4

The aim of any scene graph and frame update algorithm is to achieve the maximum frame rate possible. Better performance of an algorithm translates into the ability to visualize larger and more complex animations in a smooth and accurate manner, without having to increase computing resources. This design iteration focused on optimizing the previous algorithm.

Optimizing the algorithm involved identifying unnecessary computing processes and eliminating them. The *updater* within the new *parser* was identified as a source of unnecessary computing because it is not necessary to update the position and/or orientation of all moving objects at this stage.

Consider the sample VITASCOPE statements shown below, which include two additional statements ("PATH" and "MOVE").

```
PATH HaulPath (0,0,0) (50,0,0);
TIME 10;
VERTORIENT ExcBoom 45;
...
TIME 20;
MOVE DumpTruck HaulPath 50;
ROTATE ExcBoom VERT −30 2;
TIME 22;
ROTATE ExcBoom VERT 30 2;
TIME 30;
```

The "PATH" statement defines a motion path named HaulPath from coordinate (0,0,0) to coordinate (50,0,0)

(i.e., a length of 50 units). The "MOVE" statement denotes that DumpTruck (refers to a dump truck object in the animation) must travel on the HaulPath in 50 time units (i.e., DumpTruck will move by (1,0,0) units for every 1.0 animation time unit), with start and end times of 20 and 70, respectively.

Consider the previous algorithm walkthrough, in which the *updater* is called three times (two times from within the *parser*, and once from outside). All three times, the *updater* is called at animation time 23. At each of these *updater* calls, DumpTruck's position will be calculated to be (3,0,0). Here, a single update of DumpTruck's object-MRT would have been sufficient.

The previous algorithm was inefficient in that position/orientation updates for all objects might be called multiple times from within the same *parser*. Hence, it was concluded that the *updater* within the new *parser* was a source for unnecessary computing time. Because the *updater* within the *parser* does not have to update the position and/or orientation of all moving objects, and instead performs the action selectively; the original POUP was replaced by a *selective-POUP* (SPOUP) (hereafter referred to as the *selective-updater*) (see Figure 9).

The *selective-updater* is called after parsing an instruction but before executing it, and first checks if the parsed statement will register an object-MRT. If so, the *selective-updater* then checks to see if an object-MRT of the same type and same object is currently registered with the *updater*, and if so updates that particular registered object-MRT.

For the above example, at the *selective-updater* in the optimized *parser*, ExcBoom's vertical rotation is the only object-MRT that is updated (as required); and DumpTruck's object-MRT is not updated (which is accounted for in the *updater* call that follows the *parser*).

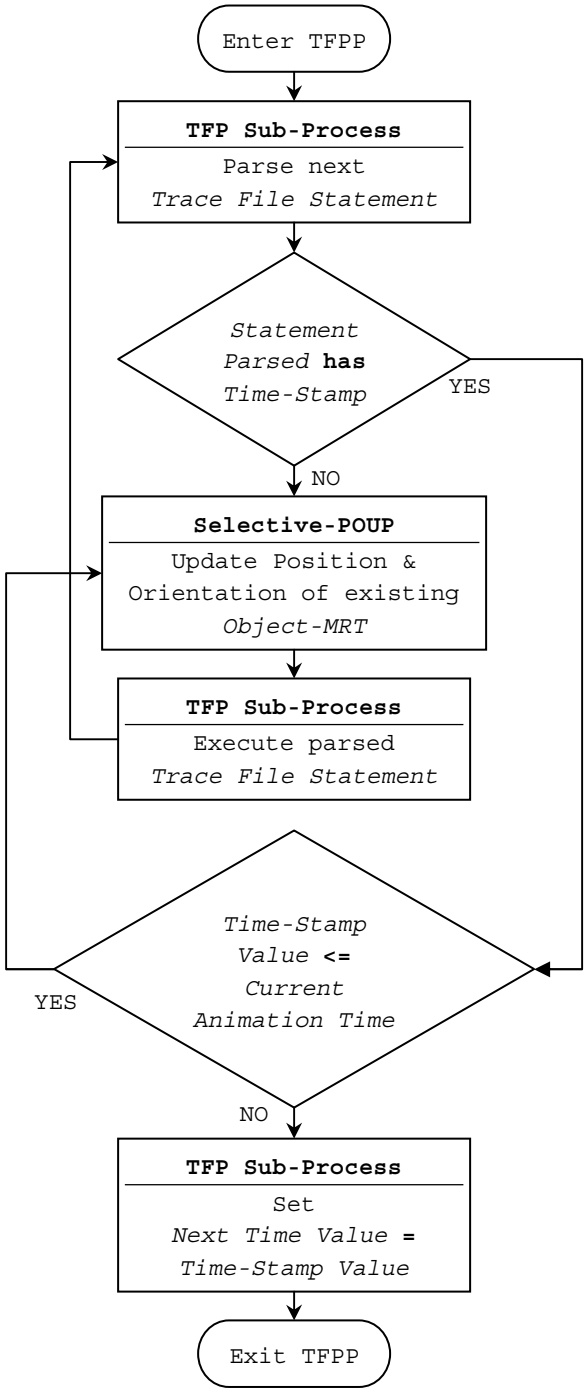

Fig. 9. TFPP of the scene graph and frame update algorithm from design iteration 4.

6.1 Critical animation speed

Previously (in iteration 2), the number of statements parsed and executed in a single *parser* was limited to the number of animation statements with the same timestamp. In contrast, there are no limitations placed on the number of animation statements that can be parsed and executed in a single optimized *parser*. Hence, as the animation speed increases, the number of statements that are parsed and executed in a single *parser* can increase.

Larger animation speeds cause larger time advances, and if the time advances are sufficiently high it can result in a *parser* having to parse statements from multiple timestamps, which increases *parser* times. Hence, at very high animation speeds, *parser* times can become so large that the algorithm no longer maintains an adequate frame rate to produce a smooth and continuous animation.

The animation speed at which the algorithm can no longer maintain an adequate frame rate is the critical animation speed for the optimized algorithm. Initial testing showed that the critical animation speeds

achieved with the fourth algorithm are very large (greater than 10,000 for the earthmoving model). Animation speeds exceeding the critical value are not suitable for any use (even simulating time-lapse photography) because at such speeds, it is nearly impossible to clearly visualize any part of or the entire operation. Note that this limitation is not present in the iteration 2 algorithm because the algorithm limits the number of animation statements that are processed in a single *parser* call.

7 CONCLUSIONS

To enable smooth and continuous 3D animations from discrete-event simulation models, the scene-graph update algorithm must be capable of converting discrete time based animation instructions into continuous animation information. The scene graph and frame update algorithm must achieve three basic performance goals at all animation speeds: (1) a suitably high frame rate; (2) accurate computations of articulations, movements, and transformations of virtual objects during the animation run; and (3) maintain the user-defined viewing ratio between viewing time and animation time throughout the animation run.

At high animation speeds, any scene graph and frame update algorithm will have to parse and execute animation instructions with different timestamp values (which are sequential). If not accounted for, the execution of multiple unique timestamped animation instructions can lead to errors being encountered that ultimately result in a distorted animation scene.

The authors present the evolution of the algorithm's design to achieve all of the performance goals and present the final algorithm that does so. The final algorithm is capable of handling very high animation speeds, which enables the algorithm to be suitable for future adaptation/adoption for next generation applications (e.g., concurrent simulation animation, real time animation of construction site work).

Future work in this area is needed to support more advanced animation modeling requirements such as animating preemption in simulated operations, and constructing new object-MRTs that are dependent on other existing object-MRTs. This work is currently in progress.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the National Science Foundation (NSF) (Award # CMMI-0732560). Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Anderson, J. & Anderson, B. (1993), The myth of persistence of vision revisited, *Journal of Film and Video*, **45**(1), 3–12.

Biles, W. E. & Wilson, S. T. (1987), Animated graphics and computer simulation, *1987 Winter Simulation Conference*, IEEE, Piscataway, NJ, 472–77.

Cox, S. W. (1988), GPSS/PCTM graphics and animation, *1988 Winter Simulation Conference*, IEEE, Piscataway, NJ, 129–35.

Henriksen, J. O. (1998), Windows-based animation with PROOF™, *1998 Winter Simulation Conference*, IEEE, Piscataway, NJ, 241–47.

Jain, S. (1999), Simulation in the next millennium, *1999 Winter Simulation Conference*, IEEE, Piscataway, NJ, 1478–84.

Kamat, V. R. (2003), VITASCOPE: Extensible and scalable 3D visualization of simulated construction operations, Ph.D. dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA.

Kamat, V. R. & Martinez, J. C. (2001), Visualizing simulated construction operations in 3D, *Journal of Computing in Civil Engineering*, **14**(4), 329–37.

Kamat, V. R. & Martinez, J. C. (2002), Scene graph and frame update algorithms for smooth and scalable 3D visualization of simulated construction operations, *Journal of Computer-Aided Civil and Infrastructure Engineering*, **17**(4), 228–45.

Kamat, V. R. & Martinez, J. C. (2003), Validating complex construction simulation models using 3D visualization, *Systems Analysis Modeling Simulation*, **43**(4), 455–67.

Law, A. M. & Kelton, W. D. (2000), *Simulation Modeling and Analysis*, 3rd edn., McGraw-Hill, New York.

Robinson, S. (1997), Simulation model verification and validation: Increasing the user's confidence, *1997 Winter Simulation Conference*, IEEE, Piscataway, NJ, 53–59.

Rohrer, M. W. (2000), Seeing is believing: The importance of visualization in manufacturing simulation, *2000 Winter Simulation Conference*, IEEE, Piscataway, NJ, 1211–16.

Tucker, S. N., Lawrence, P. J. & Rahilly, M. (1998), Discrete-event simulation in analysis of construction processes, *CIDAC Simulation Paper*, Melbourne, Australia, 1–14.