Computer-Aided Civil and Infrastructure Engineering **17** (2002) 228–245

Scene Graph and Frame Update Algorithms for Smooth and Scalable 3D Visualization of Simulated Construction Operations

Vineet R. Kamat & Julio C. Martinez*

Department of Civil and Environmental Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

Abstract: *One of the prime reasons inhibiting the widespread use of discrete-event simulation in construction planning is the absence of appropriate visual communication tools. Visualizing modeled operations in 3D is arguably the best form of communicating the logic and the inner working of simulation models and can be of immense help in establishing the credibility of analyses. New software development technologies emerge at incredible rates that allow engineers and scientists to create novel, domain-specific applications. The authors capitalized on a computer graphics technology based on the concept of the scene graph to design and implement a general-purpose 3D visualization system that is simulation and CAD-software independent. This system, the Dynamic Construction Visualizer, enables realistic visualization of modeled construction operations and the resulting products and can be used in conjunction with a wide variety of simulation tools. This paper describes the scene graph architecture and the frame updating algorithms used in designing the Dynamic Construction Visualizer.*

1 INTRODUCTION

Construction operations have been modeled for many years using discrete-event simulation (Huang and Halpin, 1994; Tucker et al., 1998; Martinez and Ioannou, 1999). Discrete-event simulation is a powerful objective function evaluator that is well suited to the design of construction operations. However, decision makers often do not have the training or the time to check the validity of simulation models and thus have little confidence in the results (Ioannou and Martinez, 1996). This is largely due to the fact that construction simulation tools provide users with a large amount of numerical and statistical data, but are not designed to illustrate the modeled operations graphically.

The capability to visualize modeled construction operations in 3D can be of substantial help in describing the intricacies of simulation models and in obtaining valuable insight into the subtleties of construction operations that are otherwise nonquantifiable and unpresentable. By communicating the logic and the inner working in a comprehensible manner, 3D visualization can facilitate the *validation* and *verification* of complex simulation models, thus providing an opportunity to convince all parties involved that models indeed reflect reality (Biles and Wilson, 1987; Cox, 1988; Robinson, 1997; Tucker et al., 1998; Henriksen, 1999; Jain, 1999; Law and Kelton, 2000; Rohrer, 2000). These are essential in establishing the *credibility* of simulation analyses, without which the results will not be used in decision-making (Law and Kelton, 2000).

1.1 Background

Visualization is a broad term in the realm of construction planning. 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 for construction planning may be appropriately termed visualization. These activities include, but are not limited to, the animation of construction schedules (i.e., 4D CAD)

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

(McKinney et al., 1996), design analysis of equipment in physical simulation environments (e.g., Working Model), visualization of assembly sequences and real-time virtual interactive modeling of construction equipment (e.g., IV++) (Op den Bosch, 1994), scenario creation and visualization for interference analysis (e.g., Bentley Dynamic Animator) (Bentley Systems, 2000), and interactive animation of construction equipment and product model–based information access over the internet using VRML (Campbell, 2000; Lipman and Reed, 2000).

None of the available 3D visualization tools are able to visualize construction operations modeled using discrete-event construction simulation systems. 3D visualization of modeled construction operations involves being able to see the graphical depiction of the operations being carried out with the same logical and physical relationships that are embedded in the underlying simulation models.

Model *verification* is the process of determining whether a model accurately reflects the model developer's idea of the existing or proposed system. On the other hand, the aim of *validation* is to determine whether simulation models accurately represent the real-world system under study. Validation is carried out by consulting people who are intimately familiar with the operations of the actual system, but who are not necessarily proficient with modeling tools. Simulation models are termed as credible when the models and their results are accepted as being valid, and they are used as an aid in making decisions (Law and Kelton, 2000). 3D visualization facilitates both model verification and validation and can thus help establish the credibility of simulation analyses. In addition, it can provide subtle visual details about the modeled operations that can be critical in making decisions.

Modeled construction operations have been visualized in several ways in the past. Schematic models capable of dynamically displaying associated model information (Huang and Halpin, 1994) and graphical iconic animation (Liu and Ioannou, 1993; Shi and Zhang, 1999) have been used to illustrate simulated construction operations. Although schematic visualization is somewhat good at model verification, it has little applicability in model validation.

2D system visualization tools such as Proof (Wolverine Software, 1995; Henriksen, 1999) have been effectively used to visually communicate some modeled construction (Ioannou and Martinez, 1996; Martinez, 1998) and mining operations (Sturgul and Seibt, 1999), as well as operations in other disciplines. Proof graphically illustrates changes in the state of modeled systems in one plane of motion by changing the position, shape, and/or color of icons representing resources. 2D visualization, although effective in communicating the logic of many simulation models, inherently lacks the real-world 3D capabilities that are indispensable for the realistic visualization of many complex construction operations.

Some manufacturing simulation systems display real-time 3D animations of modeled operations during simulation runs. Examples of such systems include Delmia's Quest (Delmia, 2000) and AutoSimulations' AutoMod (AutoSimulations, 2000). These systems have modeling constructs and built-in 3D templates of common manufacturing environments and equipment that enable users to simulate and visualize most manufacturing operations (Donald, 1998; Phillips, 1998). 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 spread out over a vast area (e.g., earthmoving, tunneling, paving, etc.). Although some construction operations can be modeled and visualized in 3D using manufacturing simulation systems, they are generally unable to effectively handle the additional complications introduced by the dramatic changes in the geometry of the construction site as work progresses (Tucker et al., 1998).

In addition, in all these systems, the simulation engines are tightly coupled with their built-in visualizers. This compels model developers who desire to visualize their models in 3D to learn and use a different simulation tool than the one with which they are proficient. For instance, a GPSS (Schriber, 1995) user intending to visualize his/her models in 3D would have to entirely recreate the models in a different 3D-enabled simulation system. The time and effort invested by modelers in achieving proficiency in a particular simulation system of choice is phenomenal.

Furthermore, these 3D visualization–enabled systems are tied to their own simulation engines based on *process interaction*. This makes them quite effective for modeling manufacturing systems but not a clear choice for construction. The use of these systems to model and animate construction operations requires a radical change in the frame of thought of construction model developers (Oloufa, 1993; Tucker et al., 1998). Simulation strategies (process interaction vs. activity scanning) and their impact on construction operations modeling are discussed in detail in the literature (Martinez and Ioannou, 1999).

Given the current state of affairs, the authors experienced the need for a generic 3D visualization system that would be capable of realistically depicting modeled construction operations as well as the evolving construction products in 3D virtual space.

1.2 The initiative

New software development technologies emerge at incredible rates, allowing engineers and scientists to create domain-specific applications that far surpass previous attempts. The design of a generic 3D construction operations visualizer presents numerous interesting challenges. Construction sites are highly and, at times, unpredictably dynamic. Construction products are built by performing numerous operations that involve complex interactions between multiple pieces of equipment, labor trades, and materials. In selecting suitable high-level computer graphics tools to approach the design of a 3D construction visualization system, the authors saw most promise in a computer graphics technology based on the concept of the scene graph. Scene graph application programming interfaces (APIs) facilitate and speed up the process of creating complex, domain-specific 3D graphics applications.

The authors capitalized on scene graph technology to design and implement a general-purpose 3D visualization/animation system, the Dynamic Construction Visualizer, that enables realistic visualization of modeled construction operations and the resulting products. This paper describes the scene graph architecture and the frame updating algorithms used in the design of the Dynamic Construction Visualizer.

2 THE SCENE GRAPH

2.1 Overview

Conceptually, a 3D computer graphics scene can be created in one of two ways. A scene can either be modeled as a single unit or can be "assembled" from discrete components. Modeling the scene as a single unit in a modeling package involves creating and placing geometrical objects at appropriate positions and with appropriate orientations in the scene using the package's built-in tools. The scene entities so created and placed are fixed in position and orientation and can be manipulated within the design environment of the modeling package. Such static scene models are primarily used for CAD-type applications, where it is usually necessary to render, visualize, and examine large data sets ranging from individual components, to subassemblies, to entire complex layouts.

On the other hand, scenes "assembled" from discrete components within domain-specific applications allow the components to be dynamically manipulated (positioned, oriented, and scaled) within the scene. Such control over individual scene components is essential for animation. The individual scene components can be modeled separately and independently using various modeling packages or can be obtained from disparate sources, such as CAD model vendors. Scene graph technology speeds up and facilitates the creation of domain-specific scene assembling and manipulating applications that can use components modeled with various modeling packages and CAD file formats. The afforded flexibility and usability of the "assembling" utilities (i.e., scene graph APIs) and of the applications created therewith increase with their ability to use components created within different modeling packages such as 3D Studio (.3ds), AutoCAD (.dxf), MicroStation (.dgn), and VRML (.wrl). This capability motivated the authors to explore the possibility of using scene graph technology in designing the Dynamic Construction Visualizer.

2.2 Scene graph characteristics

Scene graph APIs allow applications to assemble and manipulate scenes in a hierarchical data structure of objects called *nodes* that can be arranged in a directed acyclic graph (DAG) structure. This hierarchical structure is called a *scene graph*. A node is an object that can be part of or entirely comprise a scene graph. Each node is a collection of one or more *fields* (values) and *methods* that together perform a specific function. Each node encapsulates the semantics of what is to be drawn but not how it is to be drawn. The user creates one or more scene subgraphs and attaches them to a virtual universe, often referred to as the *root node*. The individual connections between scene graph nodes always represent a directed relationship (i.e., parent to child).

Figure 1 presents the structure of a simple scene graph. Scene graph nodes can be divided into two subclasses: group and leaf nodes. Group nodes assemble together one or more child nodes. A group node can point to one or more children but can have only one parent. Leaf nodes contain the actual definitions of shapes (geometry), lights, fog, sounds, and so forth. A leaf node has no children.

In Figure 1, the node "Jobsite" is the root node of the scene graph and is a group type node. Scene subgraphs are created and attached to the root node to completely encapsulate the entire jobsite in the scene graph. For instance, in Figure 1, subgraphs rooted at nodes "Excavator," "Truck," and "Terrain" are created and attached to the root node of the scene graph. The nodes "Excavator," "Truck," and "Terrain" are group type nodes. The node "Excavator" groups together all the nodes that completely describe an excavator (i.e., the base, the cabin, the boom, the stick, and the bucket). The nodes "Base," "Cabin," "Boom," "Stick," and "Bucket" contain the geometrical description of the individual excavator components and are leaf-type nodes. These individual excavator component models can be imported from any CAD modeling package whose file format is supported by the scene graph API's geometry loader. In fact, different components may be modeled in different CAD modeling packages. For instance, in the scene graph in Figure 1, the base and the cabin of the excavator may have been modeled in AutoCAD (in .dxf file format) and the other digging components in 3D Studio (in .3ds file format). Similarly, the group node "Truck" groups together the leaf nodes "Base" and "Bed"

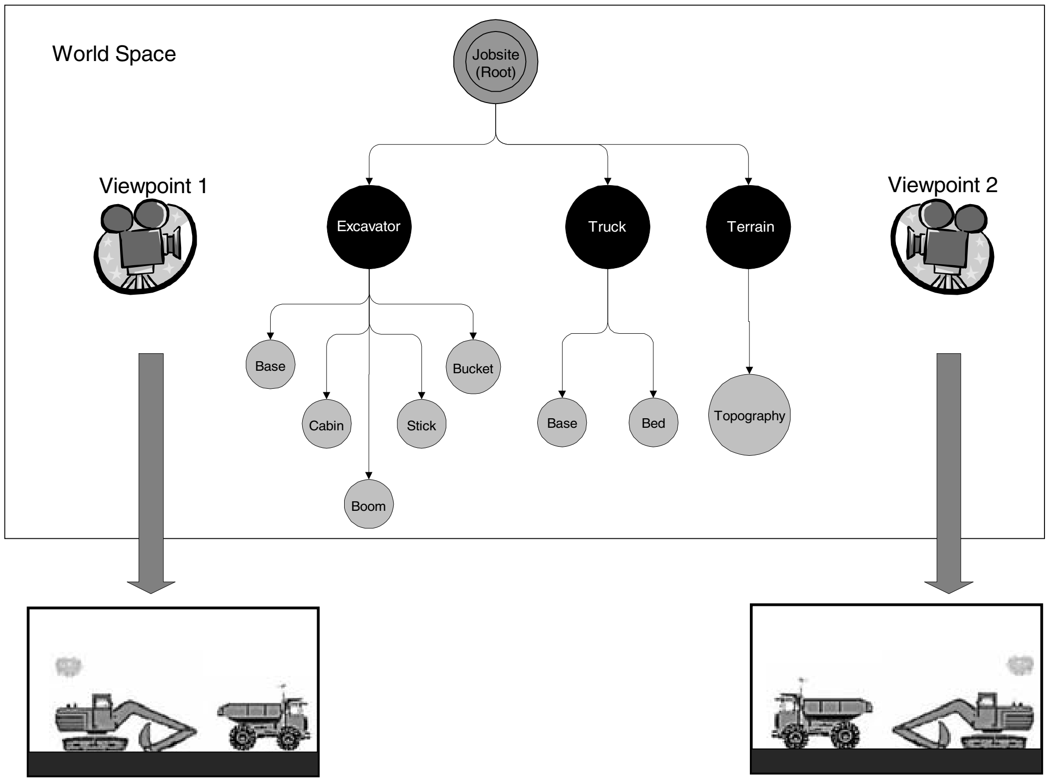

Fig. 1. Scene graph hierarchy and role of the camera.

to completely encapsulate the geometrical description of a truck and add it to the scene. The group node "Terrain" consists of only one child node "Topography" that is a leaf node and contains the geometrical description of the jobsite terrain.

Scene graph nodes hold only data. They encapsulate the semantics of what is to be drawn but not how it is to be drawn. An *action* is an operation that can be carried out on one or more nodes in a scene graph. Actions trigger the events that are prescribed in the scene graph nodes. An action visits each node in the scene graph (i.e., traverses the scene graph) and uses the data contained in each node to display, modify, or augment the state of the scene graph. Examples of actions include rendering the scene (drawing) and intersection testing (collision detection). Although actions are a part of the scene graph API's functionality, they do not have a nodal representation. The behavior of an action on a node is a function of both the action and the node. For instance, if a drawing action were applied to the root node "Jobsite" of the scene graph in Figure 1, the action would operate on all of the nodes in the scene graph and render all the visible objects (i.e., the terrain, an excavator, and a truck) contained therein. The hierarchy of the scene graph determines the order in which the nodes are acted upon when an action is applied to the scene graph. This hierarchy is established by the order in which the group nodes are added to the sub-branches in a scene graph branch.

2.3 Creating scene graphs

The coordinate system of the root node in a scene graph is known as the *world space* and is the principal frame of reference. This three-dimensional system is the basis for defining and locating in space all objects in a scene, including the observer's position and line of sight. On the other hand, a *local space* is used to define the geometry of an object (i.e., size, location, and orientation) independently of the world space. This is done to define geometrical objects independently without giving them fixed specific sizes, locations, and orientations in the world space. Each scene component imported from a CAD modeling package is defined in its own local space.

A scene is created by appropriately sizing and placing geometrical objects (each created and defined in its own local space) at appropriate positions and orientations in the world space. This is accomplished using transformation nodes. Transformation nodes allow scene graph builders to set and manipulate the location (translation), rotation, and scale of their child nodes. Transformation nodes are group-type nodes that translate the local coordinates of their child nodes into the coordinates of their parent nodes. If there is more than one transformation node in a hierarchy of nodes, each transformation node translates the coordinates of its children into the coordinates of its parents all the way up the hierarchy until a final transformation node translates the coordinates of geometrical objects into those of the root node (i.e., world space). Typically, transformation nodes are placed between geometrical object nodes or group-type nodes and the rest of the scene graph. For instance, in Figure 1, the group-type nodes “Excavator,” “Truck,” and “Terrain” need to be transformation nodes in order to be able to place and orient the scene components at desired positions in the scene “Jobsite.”

2.4 Visualizing scene graphs

A scene graph stores the description of a scene by encapsulating relevant data in a hierarchy of suitably arranged nodes. Visualizing a scene encapsulated in a scene graph involves obtaining a suitable view of the hierarchy from any desired viewpoint. Scene graph APIs provide several utilities to position and orient a viewpoint within the world space. The viewpoint in the world space is analogous to a video camera whose image is continuously transmitted to the viewport (rectangular window on the computer screen) through which a viewer views the scene. The position and orientation of a viewpoint (camera) can be dynamically manipulated within scene graph applications to obtain different views of the same scene graph. Conversely, more than one viewpoint (camera) can be positioned and oriented in the world space to simultaneously transmit different views of the same scene graph to different viewports (windows).

Although cameras are positioned and manipulated in the world space, they are not a part of the scene graph and hence do not have a nodal representation. They are external mechanisms for visualizing the data encapsulated in the scene graph. Figure 1 also summarizes graphically the relationship between scene graphs and cameras. Viewpoints placed at different positions in the world space provide different views of the same scene graph.

2.5 Animating scene graphs

Scene graphs are constructed by developing an appropriate hierarchical node structure of individual scene components that are scaled, placed, and oriented at desired positions in the world space by suitably setting transformation node fields. Depicting the motion of scene entities (i.e., animation) involves a dynamic relationship between scene graph components. Such a relationship is achieved by dynamically manipulating the *fields* (values) of the transformation nodes in the scene graph.

For instance, assume that the truck in the scene graph in Figure 1 is placed at point $(0, 0, 0)$ in the world space by setting the translation field of its transformation node (“Truck”) to $(0, 0, 0)$. Imagine that it is required to move this truck along the positive X-axis (in world space) in discrete steps at a rate of 1 unit per second. In order to achieve this motion, the X-axis component of the translation field in the transformation node must be incremented by 1 unit every second. For example, the value of the translation field would be $(1, 0, 0)$, $(2, 0, 0)$, and $(3, 0, 0)$ at the end of one, two, and three seconds. The motion (i.e., discrete jumping action) of the truck described above is schematically represented in Figure 2. The figure shows the changing position of the truck along the X-axis with the passage of time. At all times, camera(s) transmit view(s) of the current state of the scene graph to the appropriate viewport(s).

In this example, the truck would appear at point $(0, 0, 0)$ for 1 second, instantaneously jump to point $(1, 0, 0)$ and stay there for another second, and so on. However, realistic animation requires that objects move smoothly between discrete points with the passage of animation time. This

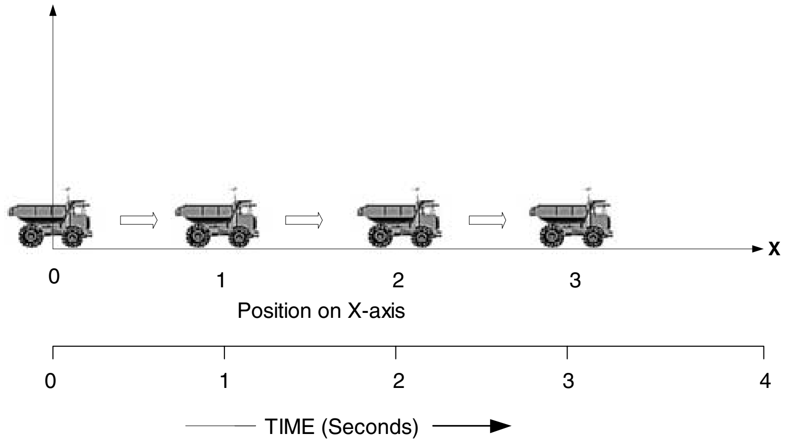

Fig. 2. Discrete-step motion of an object along the positive X-axis.

requires constant monitoring and updating of all moving objects in a scene graph. The frame updating algorithms that are used to accomplish this are explained in the following section.

3 FRAME UPDATING MECHANISMS

3.1 Fixed frame rate

In movie theaters, motion is achieved by projecting a sequence of pictures at a rate of 24 per second on the screen. Although viewers watch 24 different frames each second, the human brain blends them into a smooth animation. In fact, most modern projectors display each picture twice at a rate of 48 per second to reduce flickering. Typical computer graphics screens redraw the picture (refresh) approximately 60 to 76 times per second (Woo et al., 1997). High-end graphics workstations can refresh up to about 120 times per second. Refresh rates higher than 120 per second are beyond the point of diminishing returns, since the human eye is only so good.

Computer animation generally involves the computing and updating of the positions and orientations of all the dynamic scene objects before the frame can actually be drawn. Imagine that a constant frame rate of 60 per second is desired during computer animation. To obtain a constant frame rate, a routine similar to the one displayed in Figure 3 will need to be implemented.

The critical component in this arrangement is the sum of the times it takes for the system to compute a typical frame and draw it after clearing the screen. The animation results will degrade progressively depending on how close to 1/60 second it takes to compute, clear, and draw. In the limiting case, when the drawing takes nearly a full 1/60 second, scene objects that are drawn first will be visible for the full 1/60 second and present a solid image on the screen. However, scene objects drawn toward the end will be instantly cleared as the system starts on the next frame. This problem is compounded when the time required to draw an entire frame exceeds 1/60 second. In this case, the system will clear the screen for the next frame even before all scene objects in the current frame are drawn, causing unpredictable and distorted images. In this arrangement, the system does not display completely drawn frames. Instead, the viewer watches the drawing as it happens.

3.2 Double buffering

To alleviate the problem of displaying partially drawn frames, most graphics library implementations provide an arrangement known as double buffering. Double buffering allows the system to maintain two screen images at all times. One of the screen images is displayed while the other

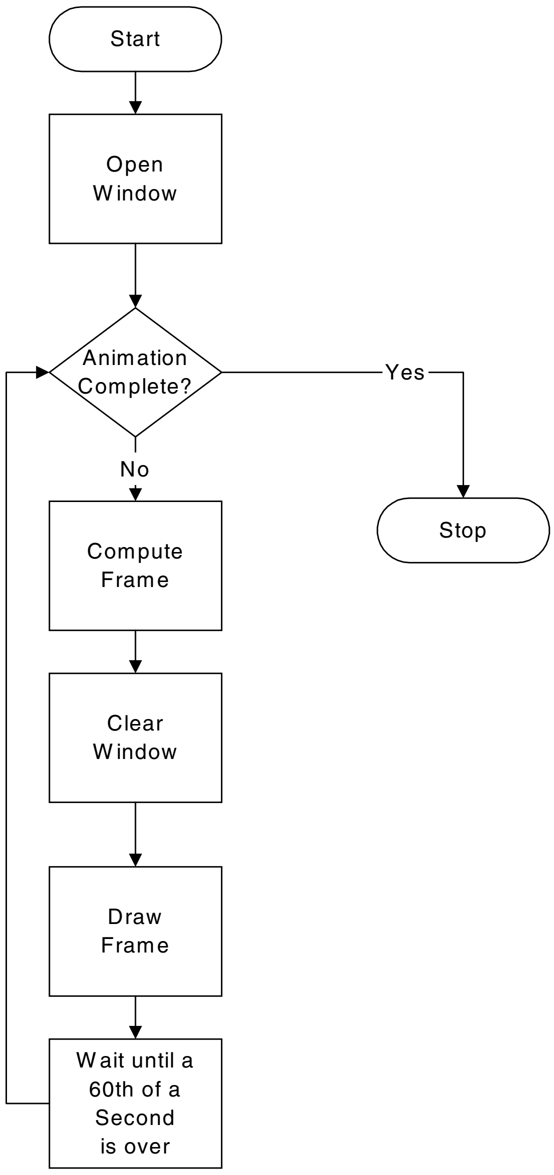

Fig. 3. Fixed frame-rate algorithm.

is being drawn. When the drawing of a frame is complete, the two images are swapped, so the one that was being displayed is now used for drawing, and vice versa. With double buffering, every frame is displayed only when the drawing is complete and the viewer never sees a partially drawn image. This improved arrangement that displays smoothly animated graphics is reflected in Figure 4.

3.3 Variable frame rate

In most double buffering implementations, the swapping of the buffers is generally synchronized with the systems' screen refresh rate. This procedure allows the previous buffer as well as the new buffer to be displayed

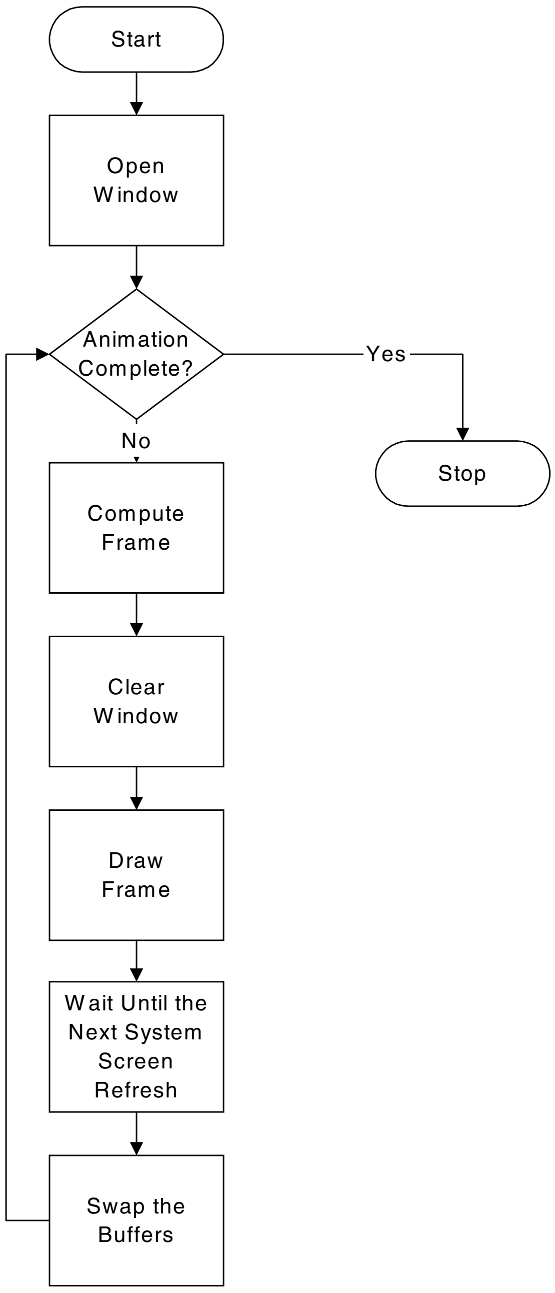

Fig. 4. Double-buffering algorithm.

completely, starting from the beginning (Woo et al., 1997). The implication of implementing such a procedure is that the fastest achievable frame rate in an animation is equal to the system's screen refresh rate. For instance, on a system that refreshes the display 60 times per second, the fastest achievable frame rate is 60 frames per second (fps). If all the frames can be computed, cleared, and drawn in under 1/60 second, an animation will run smoothly at that rate.

However, since 3D computer graphics are typically computationally expensive, more often than not typical frames are too complex to be computed and drawn between the screen refreshes of a standard machine. This delay in displaying the new frame results in the previous frame being displayed more than once. For instance, on the same 60-refreshes-per-second system, if it takes 1/40 second to draw a frame, the animation runs at 30 fps, and the graphics are idle for 1/30–1/40 = 1/120 second per frame. Since a system's screen refresh rate is constant, the obtained frame rate in animations is a multiple of the screen refresh rate. For instance, on the 1/60 second per refresh monitor, the obtainable frame rates are 60 fps, 30 fps, 20 fps, 15 fps, and so on (i.e. $60/1, 60/2, 60/3, 60/4$, and so on). The obtainable frame rate is inversely proportional to the scene complexity in the animation (i.e., the frame rate degrades proportionately as new scene objects and features are added to the scene).

4 THE DYNAMIC CONSTRUCTION VISUALIZER

The Dynamic Construction Visualizer (DCV) is a general-purpose 3D visualization/animation system that combines scene graph technology and an effective double-buffering frame-updating algorithm. The DCV allows simulation model developers to visualize modeled operations with chronological and spatial accuracy in 3D virtual space. The system is independent of any particular simulation-modeling program or CAD modeling software.

4.1 System description

DCV language files unambiguously describe the visual configuration of modeled systems with the passage of time. The DCV is as a "post-simulation" visualization engine that possesses the following characteristics:

- Uses scene graphs to organize and depict construction jobsite scenarios
- Allows the user to navigate easily in the 3D virtual space and place himself/herself at any desired vantage point by controlling the camera using the keyboard or the mouse
- Maintains an independent simulation clock, the speed of which can be controlled by the viewer depending upon the animation speed desired
- Allows the user to jump ahead or back to any desired location in the simulation by specifying a future or past time value
- Permits the viewer to start and pause the animation at any time to make static observations in the modeled system

The DCV language allows the construction and manipulation of complex scene graphs. Modeled operations are visualized in 3D by processing sequential, time-ordered animation commands written in the DCV language. The animation commands are contained in an ASCII text file hereinafter referred to as the trace file. DCV trace files are meant to be generated by simulation software. Any simulation software capable of writing custom text output during a simulation run can generate the trace files automatically. These include most of the programmable generic and special-purpose simulation languages as well as high-level programming languages such as BASIC, FORTRAN, C and C++. Non–language-based simulation software may also be adapted to generate trace files during a simulation run (Henriksen, 1999).

The DCV uses 3D models of all pertinent resources and system entities to depict the simulated operations and the evolving product in 3D. The DCV system does not possess any built-in 3D model building capability. Instead, required 3D models of system entities can be imported from a wide variety of 3D CAD modeling software. The DCV provides direct support for the VRML file format. Geometry files from practically every 3D modeling program (e.g., AutoCAD, MicroStation, 3D Studio) can be easily exported or converted into VRML format.

4.2 Scene graphs and the DCV

The DCV has been designed using the Cosmo3D Scene Graph API. Cosmo3D (Silicon Graphics, 1998) is a C++ toolkit that brings 3D graphics programming to desktop applications. Cosmo3D facilitates the development of complex graphic applications by allowing application developers to use a higher-level interface than the lower-level OpenGL language on which it is based. The scene graph architecture in Cosmo3D allows developers to arrange and manipulate visible objects in a hierarchical structure that facilitates the depiction of the realistic motion of complex construction equipment, such as dumptrucks and forklifts, as well as that of complex hierarchical assemblies, such as cranes and backhoes.

The DCV language is a high-level language that allows users to use simple text statements (commands) to construct and manipulate complex scene graphs. Table 1 lists a few of the DCV commands and provides a concise explanation of their functionality. Figure 5 presents a sample DCV trace file. Figure 6 shows how a scene graph evolves as the statements in the trace file in Figure 5 are processed.

The root node of the scene graph is created immediately when the trace file is opened in the DCV application. Two paths, "LoadToDump" and "DumpToLoad," are defined by specifying the beginning, ending, and all intermediate 3D coordinates of the points constituting the paths. The example defines paths consisting of two segments each. Paths do not have a nodal representation and hence are not part of the scene graph. They are used to compute fields for transforms that place objects along the path. Defined paths are stored into memory by the DCV application until an animation is complete.

Table 1
Selected DCV animation language commands and their functionality

Statement	*Functionality*
TIME	Indicates the simulation time at which all subsequent commands take place
CLASS	Associates a class of simulation entities with their geometric description contained in a CAD file
CREATE	Creates specific simulation objects by instantiating predefined classes
ATTACH	Attaches specific simulation objects to one another
PLACE	Places simulation objects at particular locations or at the beginning of resource movement paths
MOVE	Moves simulation objects on resource movement paths
ROTATE	Appropriately manipulates the orientations of specific simulation objects

The next three statements in the trace file grouped as (A) augment the scene graph, as represented in Figure 6A. The "CLASS" statement defines a leaf node, Terrain, which obtains its geometry from the CAD file Terrain.wrl. Terrain.wrl contains the 3D model of the terrain in VRML format. The "CREATE" statement instantiates a transformation node "ExTerrain" and adds a child object conforming to the geometry defined by class "Terrain" to it. The "PLACE" statement places the created object in the scene by adding the "ExTerrain" transformation node to the root of the scene graph. The object is placed at point $(0, 0, 0)$ in the world space by setting the translation field of the "ExTerrain" transformation node to $(0, 0, 0)$.

The subsequent three statements grouped as (B) similarly define the class "Excavator," create a transformation node "Excvtr1," and place an excavator object at point (5,2,1) in the world space. The corresponding augmented scene

```
            PATH LoadToDump (3,2,1)(0,1,5)(-5,0,3);
            PATH DumpToLoad (-4,0,3)(1,1,5)(2,2,1);

            CLASS Terrain Terrain.wrl;
[(A)]       CREATE ExTerrain Terrain;
            PLACE ExTerrain AT (0,0,0);

            CLASS Excavator EX1100.wrl;
[(B)]       CREATE Excvtr1 Excavator;
            PLACE Excvtr1 AT (5,2,1);

            CLASS Truck A30C.wrl;
[(C)]       CREATE Truck1 Truck;
            PLACE Truck1 ON LoadToDump;

            CREATE Truck2 Truck;
[(D)]       PLACE Truck2 ON DumpToLoad;
```

Fig. 5. Sample DCV trace file to illustrate the construction of a scene graph.

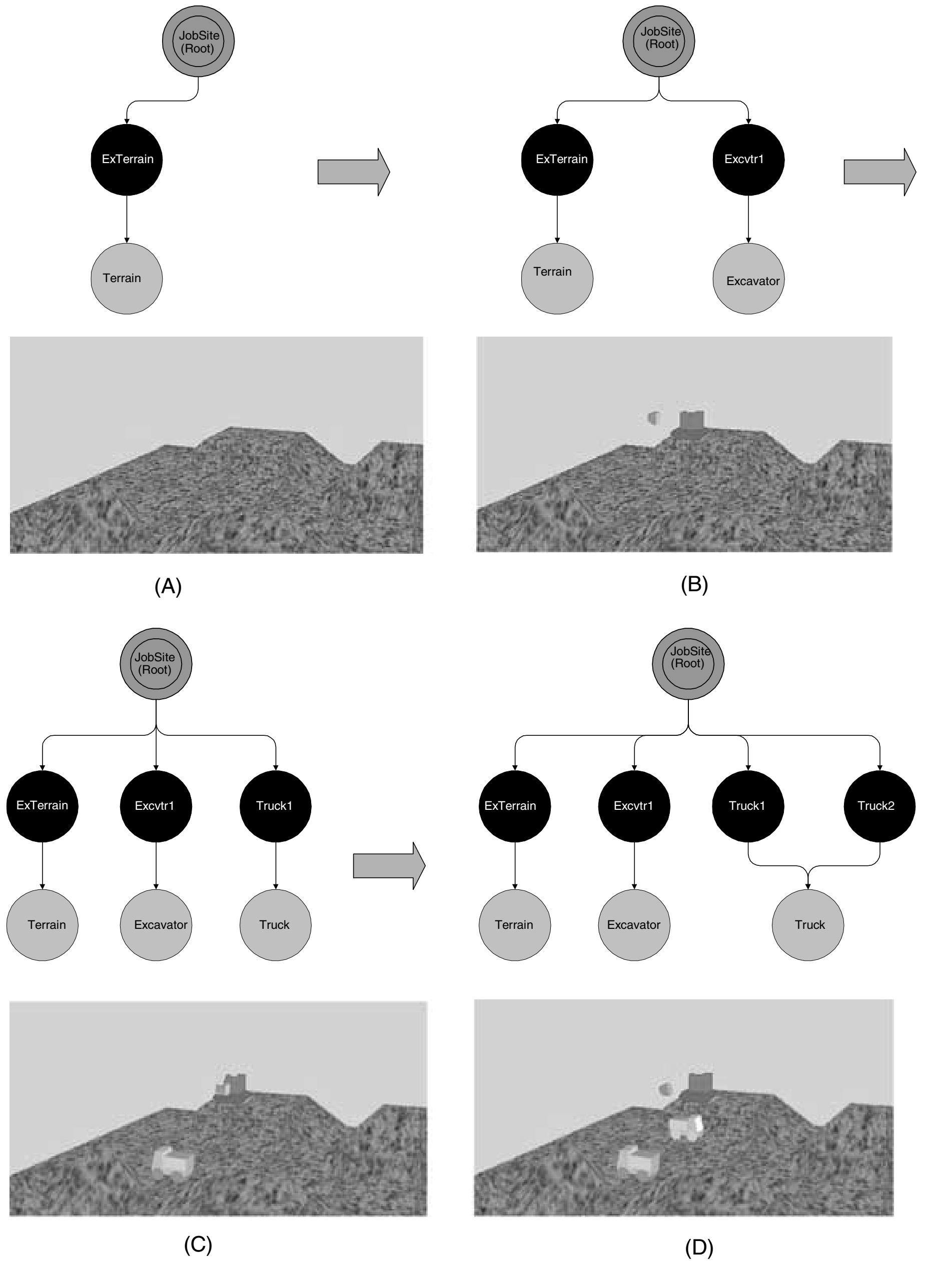

Fig. 6. Evolution of the scene graph.

graph is shown in Figure 6B. The first two trace statements in group (C) define class "Truck" and instantiate the object "Truck1" conforming to the geometry defined in class "Truck." The final statement in group (C) places "Truck1" at the beginning of Path "LoadToDump" (i.e., at point $(3, 2, 1)$) with the appropriate orientation by suitably setting the translation and the rotation fields of the transformation node. The modified scene graph is represented by Figure 6C. The trace statements in group (D) similarly define another truck, "Truck2," and place it at the beginning of the path "DumpToLoad."

Referring to the completed scene graph in Figure 6D, it is interesting to note that both "Truck1" and "Truck2" refer to the same geometry (i.e., leaf node). Therefore, the simultaneous depiction of both trucks involves presenting two different views of the same geometrical object via two different transformation nodes (i.e., "Truck1" and "Truck2") as shown in Figure 6D. This approach is infinitely scalable and is one of the major advantages afforded by scene graph technology. For instance, an entire steel frame structure consisting of hundreds of beams and columns can be depicted by loading only a few CAD models of beams and columns and placing them repeatedly at appropriate locations using multiple transformation nodes. Furthermore, due to the ability of transformation nodes to scale and rotate objects in addition to positioning them, it is theoretically possible to depict the same steel structure using just one CAD model each of a beam and column. Of course, all the beams and columns must use the same type of steel section if this is to be done.

4.3 Scene graph complexity

The scene graph hierarchy depicted in Figure 6D is a simplified version of the scene graph created by the DCV. This is done to facilitate this discussion without involving implementation details. The hierarchy depicted in Figure 6D would only allow the manipulation of the excavator and the trucks as whole units via the transformation nodes "Excvtr1," "Truck1," and "Truck2." In order to depict the articulated motion of the excavator and the trucks (i.e., realistically display the digging action of the excavator and the dumping action of the trucks), it is necessary to control the motion of individual machine components, such as the boom and the stick of the excavator, and the bed of the dumptrucks. Such hierarchical control over individual machine components can be easily achieved by suitably constructing the scene graph.

Figure 7 presents an animation snapshot of a modeled earthmoving operation that was visualized using the DCV. In this animation, the viewer is able to observe the accumulating trucks waiting to be loaded, the trucks maneuvering to get into position under the excavator, the excavator digging the earth and loading the trucks until they are full, the trucks traveling to the dumpsite, accumulating occasionally to enter the dump area, backing up and tipping their load, and then returning to the loading site to begin another cycle.

Figure 8 augments and presents the scene graph hierarchy presented in Figure 6D. This scene graph has the same scene components (i.e., a terrain, an excavator, and two trucks) as the one in Figure 6D. However, scene components are now arranged in a logical, multi-level hierarchy. The DCV implementation permits the construction and manipulation of similar complex scene graph hierarchies via commands such as "ATTACH" and "DETACH" in the trace file. Such hierarchies permit the realistic depiction of complex machine movements. An attached object can move relative to its parent. For instance, with a scene graph hierarchy similar to the one presented in Figure 8, it is easy to depict the boom of the swinging excavator being lifted. The hierarchy that develops during the earthmoving visualization depicted in Figure 7 is similar (except for the number of trucks) to the one depicted in Figure 8. The hierarchy permits the depiction of realistic digging action of the excavator and the dumping action of the trucks.

Resources often need to be combined and act as a group. For example, a flatbed and steel shapes often need to travel together as a loaded flatbed in an animation. Besides facilitating the depiction of complex hierarchical motion, the "ATTACH" and "DETACH" commands allow resources to be combined into a compound resource and compound resources to be broken up into their constituents. Figure 9

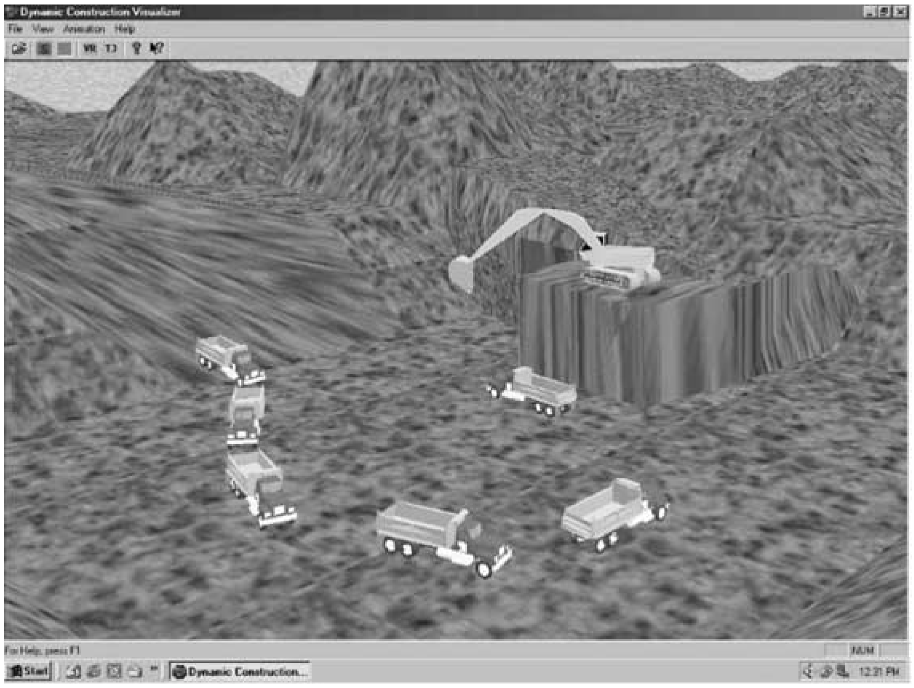

Fig. 7. Animation snapshot of the loading area in an earthmoving operation.

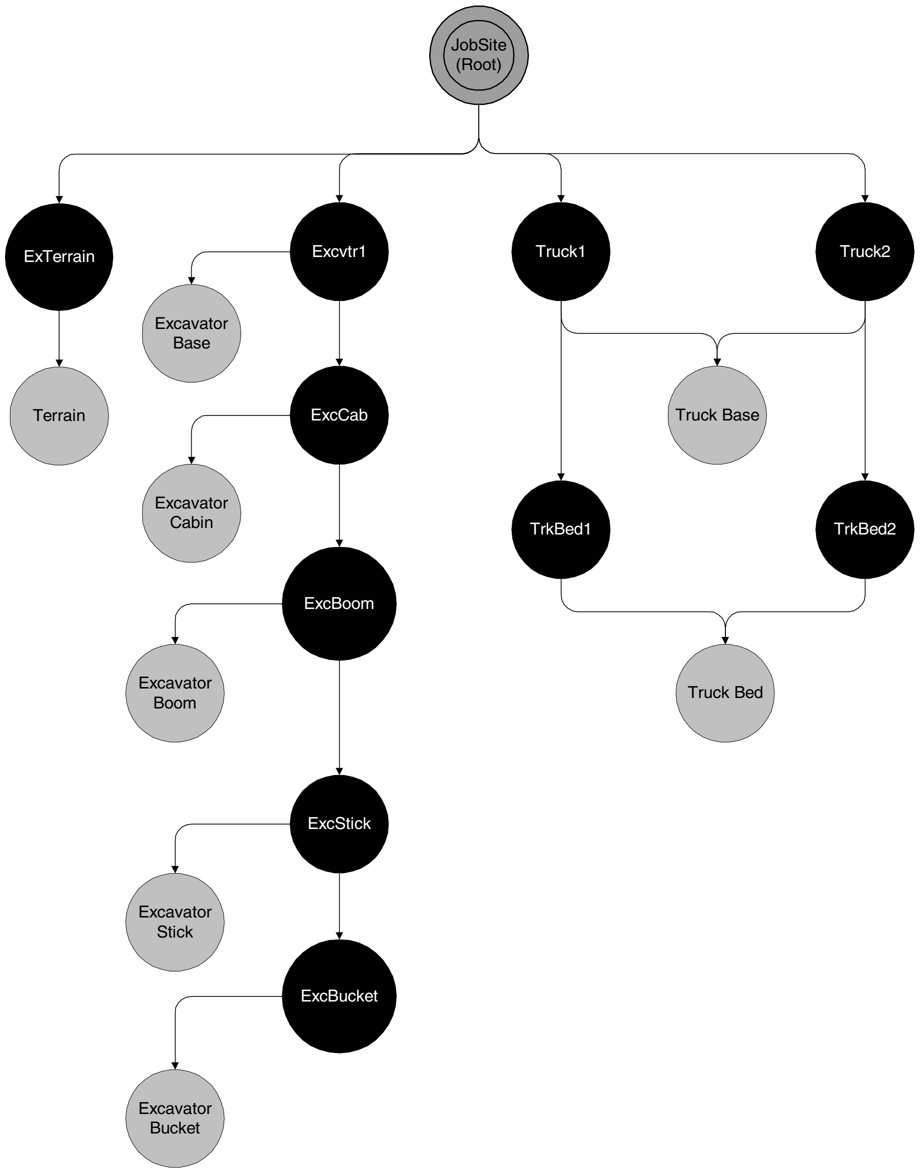

Fig. 8. Depth of a scene graph.

presents an animation snapshot of a modeled block-laying operation that shows a mason and his assistant working on a wall section. This operation was modeled and animated at a very low level of detail. The viewer is able to observe a mason constructing a wall section by laying successive courses of individual blocks. The viewer is also able to observe the materials (blocks and mortar) being delivered to the working floor by a lift (not visible in the snapshot) and being transported to the workface by the mason's assistant.

In this operation, resources often need to be grouped together at times and then separated again into constituents. For instance, the lift and the material loaded onto it need to move (and hence be grouped) together whenever a loaded lift ascends or descends. When the material is unloaded, each resource (the hauled material and the lift) needs to be independently manipulated and hence needs to be ungrouped. Figure 10 presents an example of scene graph modification by the use of the "ATTACH" and "DETACH" commands. Figure 10 represents a portion of the scene graph that develops during the visualization of the block-laying operation in Figure 9.

Imagine that the empty lift is in a lowered position and a handcart full of mortar is placed onto it. The structure of the scene graph at this point would resemble Figure 10a. The lift now needs to haul the loaded handcart to the work floor. Attaching the handcart to the lift at this point would modify the scene graph as in Figure 10b. Any positional change applied to the lift (via transformation node "Lift")

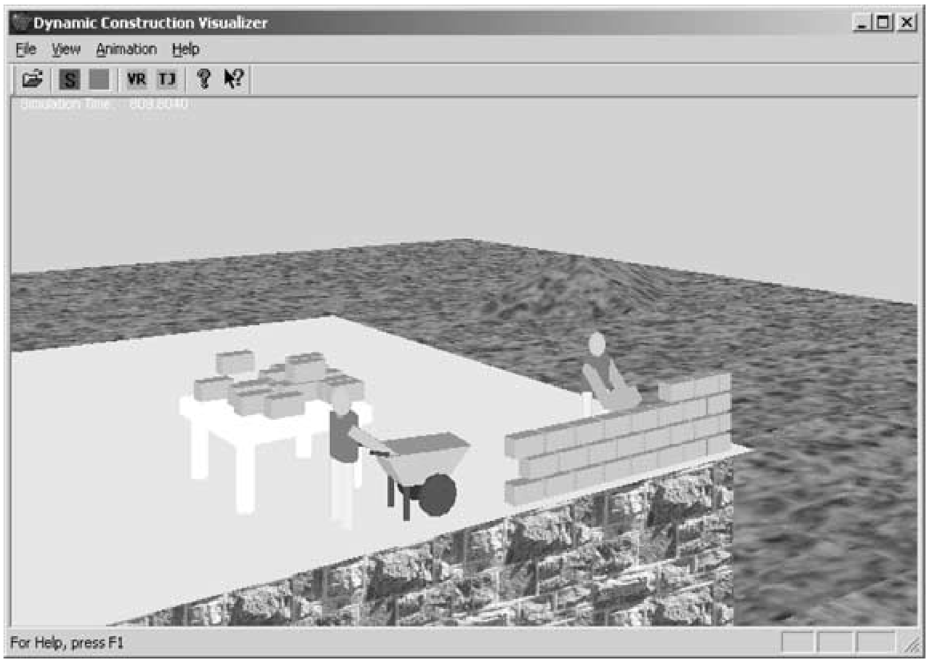

Fig. 9. Animation snapshot of a block-laying operation.

would also apply to the handcart, since the node “Handcart” is a child node of “Lift.” Conversely, positional changes applied to the attached handcart would be relative to the parent lift. When the loaded lift reaches the work floor, the handcart is detached from the lift to revert the scene graph structure to Figure 10a. The lift and the handcart can now be independently manipulated via transformation nodes “Lift” and ‘Handcart,” respectively.

5 ANIMATING CONSTRUCTION SITE ACTIVITIES

The Cosmo3D API provides several utilities that facilitate the depiction of simple animations of scene graph entities, such as the rotation of wheels in a moving car, the constant motion of a swinging pendulum, and other simple translational object motions. However, the dynamic characteristics

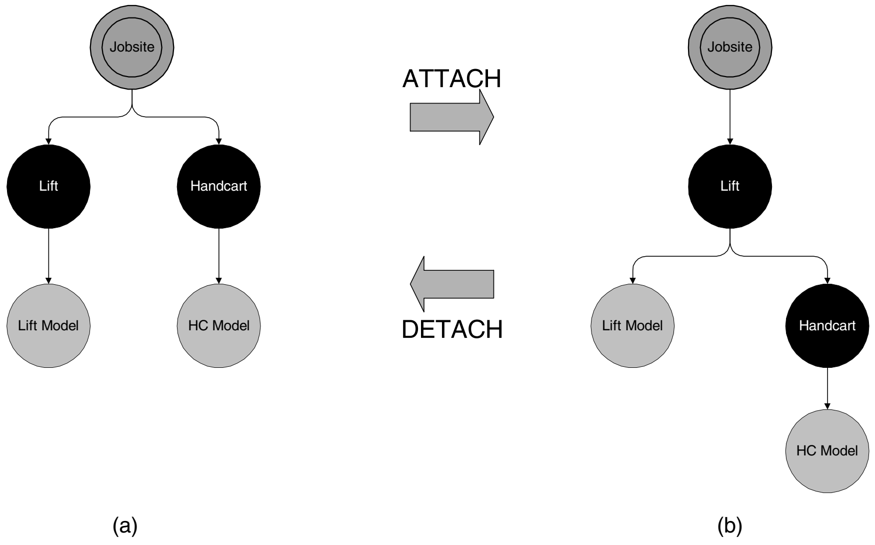

Fig. 10. Dynamic scene graph modification.

of typical construction sites demanded the design and implementation of specialized position and orientation-updating algorithms in addition to the scene graph API's built-in utilities. Understanding the working of the DCV animation clock (explained in the following section) is fundamental to understanding the animation capabilities of the DCV.

5.1 Measuring time

The DCV 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 minute, or a day) as long as it matches the time unit in the simulation model that is driving the animation.

DCV 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) uses seconds as a unit of time, and the viewing ratio is 6, then the DCV animation is running 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. In the DCV application, the user can change the viewing ratio of an animation at any time depending on the animation speed desired.

The primary time-tracking DCV command is TIME. The syntax of the TIME command is as follows:

`TIME` *timevalue*;

The TIME command waits for the animation clock to reach the new value specified. The DCV then executes the commands that follow it until another TIME command is reached. When a TIME statement is encountered in a trace file, the DCV initially verifies that the *timevalue* is greater than or equal to the current animated time. If not, the animation terminates with an error. After ascertaining that the TIME command specifies a future time, the DCV suspends the reading of any more lines from the trace file until the animation time specified by the TIME command has been reached or exceeded. When that happens, the DCV 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, the DCV continues to display the animation as it progresses at a constant, user-specified viewing ratio.

Figure 11 augments and presents the sample DCV trace file previously presented in Figure 5. The implications of visualizing this trace file in the DCV are easily interpretable. Immediately before the onset of the animation, two paths, "LoadToDump" and "DumpToLoad," and three classes, "Terrain," "Excavator," and "Truck," are defined and their representations are stored into memory. The construction of the scene graph begins at the onset of the animation (i.e., at time zero) when the terrain and the excavator objects are created and placed in the scene. Further reading of statements from the trace file is suspended until the animation time equals 6. Thus, a person viewing the animation sees a motionless excavator in the terrain for six animation time units. At this point, a truck, "Truck1," is created and placed in the scene at the beginning of path "LoadToDump." Six animation time units later (at time 12), "Truck1" starts moving along the path "LoadToDump" at a speed that will require 120 animation time units to reach the end of the path. At the same time (12), another truck, "Truck2," is created and placed in the scene at the beginning of path "DumpToLoad." At animation time 18, "Truck2" starts moving along the "DumpToLoad" and will require 90 time units to reach its destination. At the moment "Truck2" starts moving, the already in motion "Truck1" will have completed about 1/20th of its journey.

Figure 11 also displays graphically the processing of commands in the presented trace file. The real-timeline displayed below the animated-timeline assumes a viewing ratio of 6. Immediately after the trace file commands at time 12 are processed, the scene graph for this animation would exactly resemble the one presented in Figure 6D. The motion of the scene objects is achieved by manipulating the values of the transformation nodes in the scene graph. As such, the "structure" of the scene graph remains unaltered during the animation of scene objects unless it is explicitly modified using the "ATTACH" and "DETACH" commands. These commands do change the structure of the scene graph by modifying the parent–child relationships between scene graph nodes. In addition, the scene graph is obviously altered (augmented or diminished) if new scene objects are created or existing scene objects are destroyed dynamically during the animation.

5.2 The DCV frame-updating mechanism

The DCV is intended for the smooth and scalable 3D visualization of modeled construction operations over a wide range of systems ranging from typical laptops and desktops to high-end graphics workstations. As such, the DCV implementation needed to achieve "maximum performance" rendering on all supported systems. Maximum performance means that when there is no contention for rendering resources, and once utilized resources are made available, graphics rendering performance is limited only by the system's raw graphics performance and the graphics software efficiency (Kilgard et al., 1995). The DCV application needed to obtain the *maximum* performance potential of the system on which it was run.

```
PATH LoadToDump (3,2,1) (0,1,5) (-5,0,3);
PATH DumpToLoad (-4,0,3) (1,1,5) (2,2,1);
CLASS Terrain Terrain.wrl;
CLASS Excavator EX1100.wrl;
CLASS Truck A30C.wrl;

TIME 0;
CREATE ExTerrain Terrain;
PLACE ExTerrain AT (0,0,0);
CREATE Excvtr1 Excavator;
PLACE Excvtr1 AT (5,2,1);

TIME 6;
CREATE Truck1 Truck;
PLACE Truck1 ON LoadToDump;

TIME 12;
MOVE Truck1 LoadToDump 120;
CREATE Truck2 Truck;
PLACE Truck2 ON DumpToLoad;

TIME 18;
MOVE Truck2 DumpToLoad 90;
```

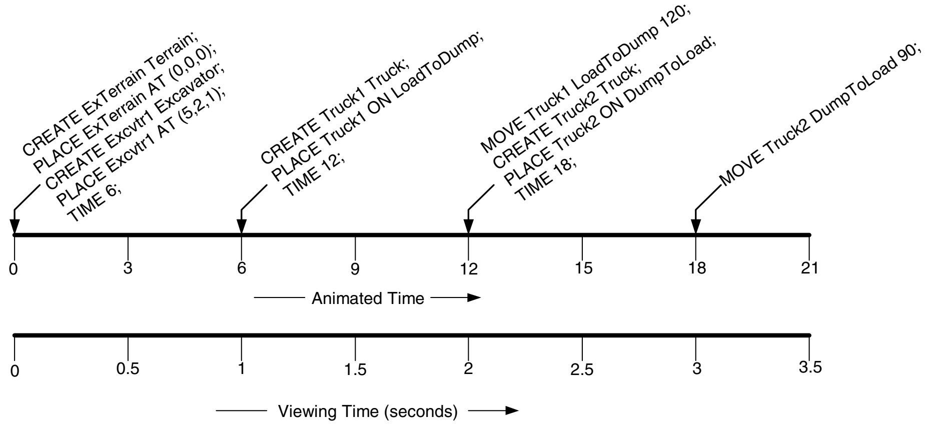
CREATE ExTerrain Terrain;
PLACE ExTerrain AT (0,0,0);
CREATE Excvtr1 Excavator;
PLACE Excvtr1 AT (5,2,1);
TIME 6;
CREATE Truck1 Truck;
PLACE Truck1 ON LoadToDump;
TIME 12;
MOVE Truck1 LoadToDump 120;
CREATE Truck2 Truck;
PLACE Truck2 ON DumpToLoad;
TIME 18;
MOVE Truck2 DumpToLoad 90;
0
3
6
9
12
15
18
21
Animated Time
0
0.5
1
1.5
2
2.5
3
3.5
Viewing Time (seconds)

Fig. 11. Processing of commands in a trace file.

In addition, construction operations range from the relatively simple to the most complex. The complexity of DCV visualizations would depend on the type of operations being visualized as well as on the level of detail incorporated therein. For instance, simultaneously visualizing all modeled operations in the construction of a building would be computationally much more expensive than visualizing a single modeled operation such as the construction of a wall section by a crew of masons. The cause of the increased computational load is that the number of scene objects that would need to be monitored for position and/or orientation changes, updated, and drawn in each frame would be significantly higher in the former visualization.

In addition to the number of scene objects, the desired amount of realism would also influence the complexity of DCV visualizations. For instance, the use of accurate, detailed 3D models and texture-mapped geometrical objects would significantly increase the load on the graphics subsystem and consequently degrade the obtainable frame rate. See Silicon Graphics (1998) and Woo et al. (1997) for a detailed discussion on texture-mapped geometry. The DCV needed to allow the smooth visualization of modeled operations at a user-specified, but constant, animation speed (viewing ratio).

The double buffering, variable frame rate paradigm was found to be most suitable in achieving the design requirements of the DCV and was therefore employed. Figure 12 presents the DCV time advancing and frame updating mechanism, which yields satisfactory results on standard machines without the need for any special hardware, and which takes advantage of extra computing power and special graphics accelerators for increased performance. For instance, in visualizing the earthmoving operation depicted in Figure 7 on a standard laptop computer powered by a 300 MHz Pentium processor, a modest screen refresh rate of 10 frames per second was obtained without the use of any additional specialized graphics hardware. Any increase in the available computing power resulted in a proportional

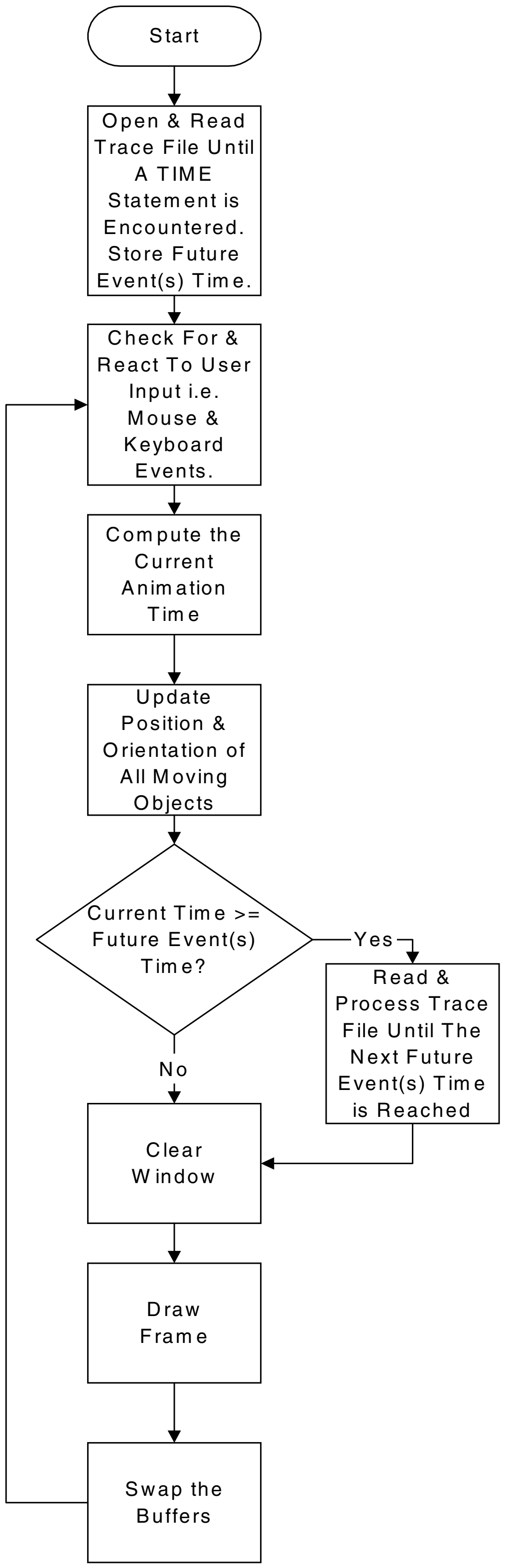
Start
Open & Read Trace File Until A TIME Statement is Encountered. Store Future Event(s) Time.
Check For & React To User Input i.e. Mouse & Keyboard Events.
Compute the Current Animation Time
Update Position & Orientation of All Moving Objects
Current Time >= Future Event(s) Time?
Yes
Read & Process Trace File Until The Next Future Event(s) Time is Reached
No
Clear Window
Draw Frame
Swap the Buffers

Fig. 12. The DCV event loop.

improvement in rendering performance and frame rate. On a desktop featuring a 600 MHz Pentium III processor, 128 megabytes of RAM, and an Nvidia TNT2 graphics card with 16 megabytes of video memory, a consistent full-screen performance in excess of 60 frames per second was observed. This frame-update mechanism is explained in the following sections.

5.3 Relationship between frame updates and the viewing ratio

The viewing ratio should always be maintained at a constant value irrespective of the frame rate being attained. For instance, if the animation speed were maintained at 6, a truck needing 24 animation time units to move along a path should complete its journey in 4 viewing seconds on any system, irrespective of the frame rate. Of course, the number of frames displayed in 4 viewing seconds (animation smoothness) would vary depending on the host systems' capabilities. In addition, the frame rate can also vary on the same system depending on the frame complexity at various times during animations.

Figure 13 graphically explains the relationship between the constant maintained viewing ratio and the obtained variable frame rate in DCV animations. Consider an example similar to the one depicted in Figure 2. Imagine that a truck requires 10 seconds in real life to cover a distance of 100 meters and that 1 unit of distance in world space (point $(0, 0, 0)$ to point $(1, 0, 0)$) represents 100 meters. If a viewing ratio of 10 were adopted during the animation, the truck would require 1 second to cover the distance between point $(0, 0, 0)$ and point $(1, 0, 0)$. Assume, for illustrative purposes, that this animation is run on various systems that refresh their display 10 times per second. In such a scenario, ideally, the journey of the truck should comprise 10 different frames as depicted in Figure 13A.

On systems powerful enough to compute, clear, and draw each frame within 1/10 second, the ideal frame rate of 10 fps illustrated in Figure 13A would be attained and the 1 second journey of the truck would be depicted as 10 different and equally spaced consecutive frames. If the same animation was run on systems that require a little more than 1/10 second but less than 1/5 (i.e., 2 times 1/10) second to produce each frame, a constant frame rate of 5 fps would be attained. As demonstrated in Figure 13B, frames $1, 3, 5, 7$, and 9 would be skipped and, consequently, frames $2, 4, 6, 8$, and 10 would each be displayed twice during consecutive screen refreshes.

In the concluding case, imagine that the same animation is run on systems that can produce typical frames within 1/10 second but require more time to generate and draw certain frames. The increased time to produce certain frames could be due to increased frame complexity or due to a temporary increase in the computational load, such as

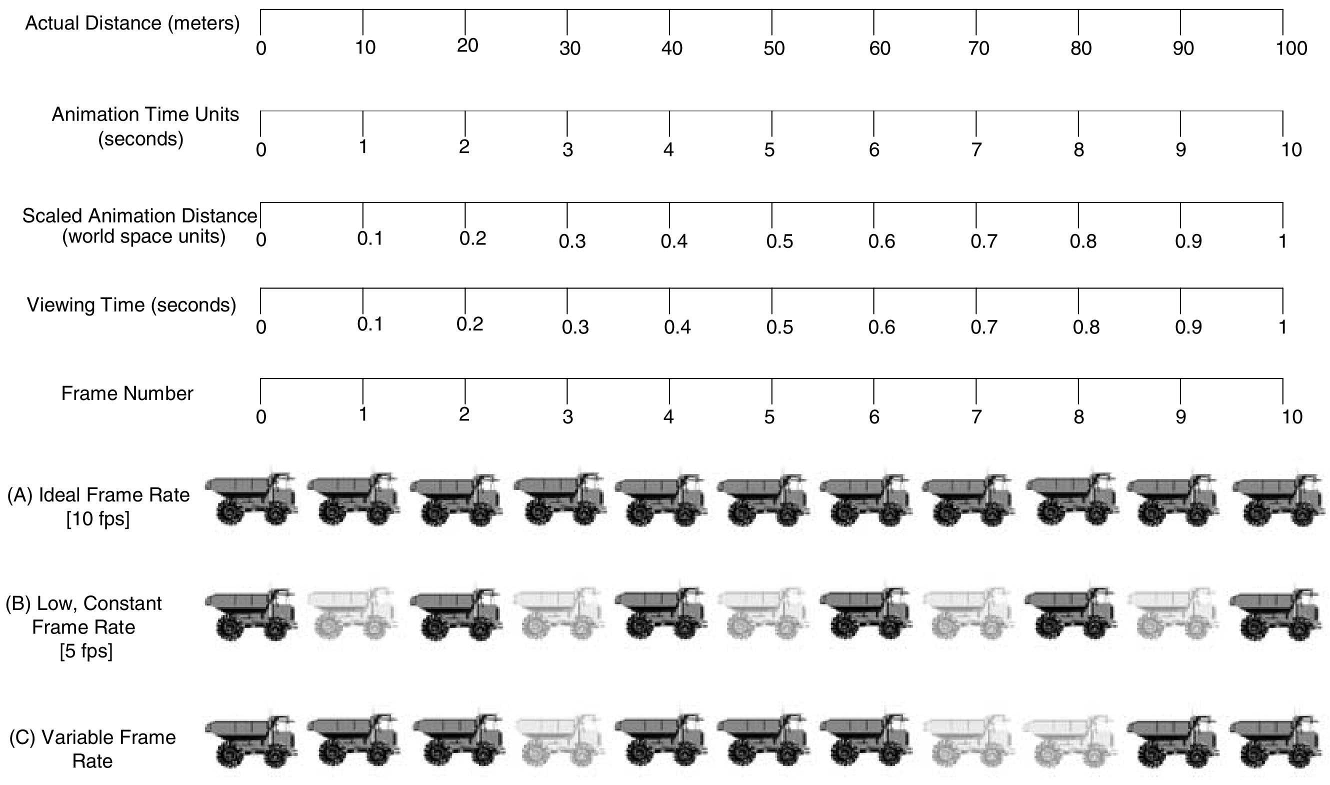

Fig. 13. Relationship between the viewing ratio and the obtained frame rate.

calculating and updating the positions of a large number of objects in the animation. Such a scenario is depicted in Figure 13C.

Frames 1 and 2 are computed and displayed within 1/10 second. However, the subsequent frame (frame 3) is not produced by the system within the next 1/10 second and is therefore skipped (i.e., frame 2 is displayed again during the next screen refresh). The system uses the idle time (accrued by skipping frame 3) and the time allotted for the next frame to compute and draw frame 4. A steady state is temporarily attained and frames 5 and 6 are each produced within 1/10 second. At this point, the system is again unable to produce the next frame (frame 7) within the allotted time and hence skips it, displaying frame 6 for the second time. The system tries to use the accrued idle time and the allotted frame time to produce frame 8. However, it fails to do so and skips frame 8 as well, displaying frame 6 for the third time. Steady state is attained subsequently and frames 9 and 10 are produced within their allotted time.

The vital point to be noted in this discussion is that on all systems on which the animation is run (Figures 13A–C), the journey of the truck requires 1 viewing second irrespective of the frame rate obtained. Frame 10 is drawn at the end of 1 second regardless of the intermediate frames. On all systems, the DCV frame-update algorithm maximizes the number of intermediate frames, and hence the smoothness of the animation, by fully utilizing the host system's frame-generating capacity at all times. The implementation of the DCV thus smoothly animates construction site activities at a fixed user-defined and controllable viewing ratio by achieving maximum possible performance from host systems.

6 CONCLUSION

The design of a scalable system capable of smoothly animating simulated construction operations in 3D requires three key technologies: (1) the conversion of discretely recorded animation information into smooth motion; (2) the spatial organization and rendering of multiple dynamic objects; and (3) the efficient sequencing and timing of frames such that the ratio of viewing to simulated time is constant.

Discrete-event simulation systems can communicate with other processes only at discrete, but possibly random, sets of simulated time points. 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. Smooth animation is continuous, however, and achieving it based on discrete information recorded at non-fixed time steps is a challenge. The straight-line DCV language and the algorithms that the DCV system uses to convert discrete information demonstrate that this is not only possible, but also quite effective.

3D visualizations require 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. 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 algorithms presented here illustrate how the DCV creates, manipulates, and manages such databases while animating simulated construction operations.

In implementing the DCV, the authors experimented with the various frame-updating algorithms described in this paper. The high variability of the graphical and computational processing load required to render and maintain the scene graphs makes the commonly used fixed frame rate algorithms ineffective because they reduce the frame rate to that of the worst case experienced. The variable-rate, double-buffered frame-updating algorithm developed for the DCV is able to achieve the smoothest possible motion and is scalable across a wide range of hardware platforms. While jerkiness is evident only during moments of intense computational and graphical demands, the ratio of simulated to visualized time is maintained with near-zero variation.

ACKNOWLEDGMENTS

The development of the Dynamic Construction Visualizer has been supported by the National Science Foundation (Grant CMS-9733267). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

AutoSimulations, Inc. (2000), AutoMod Simulation Software, http://www.automod.com/simulation/simsoftware.html.

Bentley Systems, Inc. (2001), *Bentley Dynamic Animator Version 2.0 User's Guide*, Exton, PA.

Biles, W. E. & Wilson, S. T. (1987), Animated graphics and computer simulation, *Proceedings of the 1987 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 472–7.

Campbell, D. A. (2000), Architectural construction documents on the Web: VRML as a case study, in *Automation in Construction*, **9**, Elsevier Science, New York, pp. 129–38.

Cox, S. W. (1988), GPSS/PC Graphics and animation, *Proceedings of the 1988 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 129–35.

Delmia, Inc. (2000), Factory Simulation Solutions, http://www.delmia.com.

Donald, D. L. (1998), A tutorial on ergonomic and process modeling using Quest and Igrip, *Proceedings of the 1998 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 297–302.

Henriksen, J. O. (1999), General-purpose concurrent and post-processed animation with PROOF, *Proceedings of the 1999 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 176–81.

Huang, R. & Halpin, D. W. (1994), Visual construction operation simulation: the DISCO approach, *Journal of Microcomputers in Civil Engineering*, **9**(6), 175–84.

Ioannou, P. G. & Martinez, J. (1996), Animation of complex construction simulation models, *Proceedings of the 3rd Congress on Computing in Civil Engineering*, ASCE, Reston, VA, 620–6.

Jain, S. (1999), Simulation in the next millennium, *Proceedings of the 1999 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 1478–84.

Kilgard, M. J., Blythe, D. & Hohn, D. (1995), System support for OpenGL direct rendering, *Proceedings of Graphics Interface '95,* Quebec City, Quebec, http://trant.sgi.com/opengl/docs/Direct/direct.html.

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

Lipman, R. & Reed, K. (2000), Using VRML in construction industry applications, *Proceedings of the Web3D-VRML 2000 Fifth Symposium on Virtual Reality Modeling Language*, Monterey, CA, 119–24.

Liu, L. Y. & Ioannou, P. G. (1993), Graphical resource-based object-oriented simulation for construction process planning, *Proceedings of the 5th International Conference on Computing in Civil and Building Engineering*, ASCE, Reston, VA, 1390–7.

Martinez, J. C. (1998), Earthmover—simulation tool for earthwork planning, *Proceedings of the 1998 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 1263–71.

Martinez, J. C. & Ioannou, P. G. (1999), General purpose systems for effective construction simulation, *Journal of Construction Engineering and Management*, ASCE, **125**(4), 265–76.

McKinney, K., Kim, J., Fischer, M. & Howard, C. (1996), Interactive 4D-CAD, *Proceedings of the 3rd Congress on Computing in Civil Engineering*, ASCE, Reston, VA, 383–9.

Oloufa, A. A. (1993), Modeling and simulation of construction operations, in *Automation in Construction,* **1**, Elsevier Science, New York, 351–9.

Op den Bosch, A. (1994), Design/construction processes simulation in real-time object-oriented environments, PhD Dissertation, Georgia Institute of Technology, Atlanta, GA.

Phillips, T. (1998), AutoMod by AutoSimulations, *Proceedings of the 1998 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 213–8.

Robinson, S. (1997), Simulation model verification and validation: increasing the user's confidence, *Proceedings of the 1997 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 53–9.

Rohrer, M. W. (2000), Seeing is believing: the importance of visualization in manufacturing simulation, *Proceedings of the 2000 Winter Simulation Conference,* Society for Computer Simulation, San Diego, CA, 1211–6.

Schriber, T. J. (1995), Perspectives on simulation using GPSS, *Proceedings of the 1995 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 451–6.

Shi, J. J. & Zhang, H. (1999), Iconic animation of construction simulation, *Proceedings of the 1999 Winter Simulation Conference*, Society for Computer Simulation, San Diego, CA, 992–7.

Silicon Graphics, Inc. (1998), *Cosmo 3D Programmer's Guide*, Mountain View, CA.

Sturgul, J. R. & Seibt, F. (1999), A simulation and animation model of the transport of coal to the port, *Proceedings of the 8th Mine Planning and Equipment Selection Symposium,* Balkema, Rotterdam, Holland.

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

Wolverine Software Corporation (1995), *Using Proof Animation*, Second Edition, Annandale, VA.

Woo, M., Neider, J. & Davis, T. (1997), *OpenGL Programming Guide*, Second Edition, Addison Wesley, Reading, MA.