

# Fall 2002 Research: Generalized Graphics System

Greg Waltz

June 1, 2003

## 1 Introduction

The methods of representing graphical data have been in development ever since computer graphics become a reality. Many concepts, such as the scene graph and parametric surfaces, have been developed since then. Graphics toolkits and applications have come and gone. Those systems that have stayed are either commercial (Maya) and expensive, proprietary, or otherwise technically unsuitable for general yet powerful graphics tasks. There is yet to be a single and open environment for a wide variety of graphics techniques to be implemented.

The goal of this system is to meet the need for a common platform in which to work with computer graphics in all fields, including research, art, and general graphics application development. To implement such an all encompassing graphics kit, it is necessary to eliminate or simplify some of the most laborious application development tasks. User interfaces, documentation, and extensibility are all considerations in the simplification task. Further, the choice of rendering toolkit (OpenGL, DirectX, Performer, Inventor, and Optimizer to name a few) can make or break a system.

It seemed that with the release of the open source version of Blender, there was no need for this system. But Blender is only intended to be a modeler and even a game development platform. Hopefully, by tackling the aforementioned considerations, promoting code reuse within and amongst applications, and other concepts, this system will prove itself useful in a wide variety of applications.

## 2 Core System and Overview

What is at the heart of all of these goals? The core system consists of mechanisms to store, load, interchange, and run the system components. Much of how the core works is part of the implementation process. Most of the sections below discuss components of the core and how components must work to satisfy the core requirements.

The core is made of the component database, an installation utility, compilation facilities (through the GNU auto tools), the glib utility library, the interface interpreter, and a small utility to initialize the system and load the desired system application (covered later). The core is also responsible for facilitating the interaction between scripts and modules.

The core system is intended to provide a common place for applications and research tools to be developed. This system should allow for much sharing of code. The actual run-time functionality is then determined by the application implementation and what other components are loaded, as described throughout this document.

## 3 Class Hierarchy and Inheritance

To facilitate the generality of the system and prevent recompilation, the concept of run-time class interchangeability is used. This might not be a new idea (it seems that Microsoft's magical .Net does something akin), but it is certainly a valuable one to this system. In this context, class interchangeability means that not only can a class implementation be interchanged with another at compile time, but also at run-time. This idea extends that of a virtual or interface class from compilation to run-time.

General classes will be specified in headers and stored in a common location. Many different implementations of a class can be made and allowed to reside in the same general area (as described in the database section later). Each implementation of a class functions on the same data and has the same functions available as specified in the class header; therefore, each such implementation is interchangeable. All that needs to be done at run-time is unload the current implementation, load a different implementation, and replace the class function references with that of the new implementation. This functionality has some low-level implications that will be discussed in the system implementation document.

Inheritance will be based off of the class headers and not implementation specifics. This restriction should not be a problem and will allow the interchangeability requirement. If an implementation needs to be inherited based on implementation specific information, that may be taken care of within packages (see modules and scripts for packages). For example, a set of OpenGL renderers may depend on inheriting information specific to OpenGL, but other components outside of the OpenGL package should and will not need to know about OpenGL; therefore, that information can be kept within the OpenGL package.

Implementations of a class may need their own specific data at run-time (information not considered in the general class specification). Consider the OpenGL renderers again. OpenGL has optimization facilities such as vertex arrays and display lists. Each displayable object in a scene will be assigned a renderer, in this case the OpenGL implementation thereof. Because of the repetitive nature of graphical displays (the scene is displayed many times), the renderers would remain with their respective scene components. In this case, the OpenGL renderers would like to take advantage of the optimization features, but they must store this information between renderings. However, this information is implementation specific and not a part of the general renderer class specification. Therefore, there needs to be a generic facility to keep track of implementation specific data.

Each class may have an implementation data field. This field would be a generic pointer that can point to any type of data. Furthermore, in the case of renderers, there may be long-term renderers with short-term renderers interspersed (see renderers topic). The long-term renderers should not go away while a short-term one temporarily replaces it; therefore, the implementation data field should be capable of storing any long-term data pointers on a stack or list that matches implementations with their respective data. Care must be taken not to abuse this facility, and should be limited to the "necessary evil" case with renderers.

Similar to VRS, it is important that developers should make inheritance choices based on the class interface, communications considerations, and general class semantics. It is important that a class inherits from classes that will have as much common meaningful functionality as possible (see communications).

## 4 Communications

Communications in this system consist of file input/output, coordination, renderers, and events/signals. File I/O, coordination, and renderers are essentially the same topic, while events/signals represent user input and other types of message passing.

File I/O refers to the loading and storing of scene and system information. Renderers are classes that take care of drawing scene components to a specific format or device. Coordination deals with the topic of being certain that scenes in different applications (possibly on different machines) are dealing with the same scene description. All of these ideas basically refer to the concept of writing to and reading from class instance data, therefore, they are treated the same way. Refer to the sections below for other information on these topics.

Events/signals (referred to as events) allow information to be passed amongst otherwise unrelated components. The most significant case is that of user interaction through the viewport. The user may input mouse motions, clicks, and keyboard presses through the viewport as discussed in the user interface section. Events may also be attached amongst scene components and have nothing to do with user input. For example, some components may be interested if another component has had a collision or some other condition. Classes will be able to accept certain types of general events that contain the event specific information. New events may also be added to the system. Events are registered in the supporting system and the implementation details will be covered in the implementation document.

Another aspect of communication is how to determine which class is used to communicate with a particular class. In the case of the OpenGL renderers, the system needs to know which OpenGL renderer to apply to a particular scene component. This should be a relatively simple task. Each base object class will contain pointers to its communication classes. When a communications component is to be applied to a class, the closest class in the communications implementation that fits the class will be chosen and used. When an instance of that class is to be communicated with, the class definition is queried for which communication implementation is to be used and then that implementation is run.

The following is an example for a renderer. Suppose the following hierarchies:

Object -j Drawable -j Geometry -j VertexObject -j NURBS

Communicator -> Renderer -> OpenGL -> OGLGeometry -> OGLVertexObject

Notice that there is no NURBS renderer for OpenGL. Suppose that a VertexObject instance is to be rendered. The first time a VertexObject is encountered, the class renderer information should be set. In this case, the OpenGL package includes an OGLVertexObject (each communicator must specify which class it works with; it is not determined by name) which will be used to render all VertexObject instances. Every time a VertexObject instance is to be rendered, the reference to the corresponding OpenGL renderer will be found in the class information and subsequently used. Suppose that a NURBS instance is to be rendered. The core system will have a tree representing the class inheritance of the loaded components. After coming upon the first NURBS instance in the scene, it will be found that the closest OpenGL component to NURBS is VertexObject. OGLVertexObject will then be used for all NURBS instances.

The closest communication class scheme presented above is really only useful for rendering or output communication. Even then it may not work appropriately if the class implementors did not consider this scheme. Consider the NURBS class. It would be logical, but not necessary, for the implementation to store the interpolated NURBS surface in the data structures of its parent VertexObject. However, it could also store its control points there or ignore those facilities altogether (see inheritance). Other problems lie in the input of information through the communication scheme. Similar to the NURBS rendering problem, consider changing a NURBS instance at the VertexObject level.

This closest level communication seems to be a necessary part of this system, largely because there may not be classes or implementations for all components (particularly new or experimental ones). However, inheritance considerations must be made by developers to minimize communications trouble (see Inheritance).

## 5 Renderers

Although renderers have the same class interface as file savers, they have some extra considerations that have been taken care of by the implementation data field and class interchangeability. However, a more detailed explanation is necessary.

Renderers allow the system to be displayed using (ideally) any rendering toolkit available. OpenGL is a clear and easy choice, but other options exist including those unforeseen. The generality requirement of the system lead to the idea of graphical toolkit independence. VRS uses this idea to not only render the scene to OpenGL, but also to Renderman and others, which is part of the power of the renderer interchangeability concept. This generality means that new file formats need not be force fit into the system. Modules can be written to handle various renderable classes and these modules may be used to render the scene to that format. This idea also ties back in with that of file I/O and putting data into class instances as written previously.

As in the OpenGL example, some renderers may need to store implementation specific data. Further, there is a distinction betwixt long-term and short-term renderers. An OpenGL renderer would generally be used in the long-term whilst a RenderMan renderer may only be used for one instance. However, these two renderer implementations are used indistinguishably. Suppose that the OpenGL renderers are being used to display the scene, but the user has requested that the scene be output to a RenderMan file. If the implementation data is stored only by a single pointer, the OpenGL renderers must deallocate their data and reallocate it once the RenderMan renderers are done. This allocation would unnecessarily slow the system momentarily. This reason is why a stack or list should keep track of long-term implementation data while a short-term implementation takes over the rendering job. That way the data remains and there is little overhead in switching rendering contexts. The facility of implementation specific data should only be limited to classes in which it is known that this feature will be needed.

## 6 Modules and Scripts

The system will be fleshed out using modules and scripts (dynamic components). Modules are dynamically loaded binaries that represent an implementation of a class. The scripts are written in some interpreted language (Python). Scripts may implement a class, glue code for some specific application, or some other interaction that a user wants to automate. Scripts are useful for fast prototyping and user interaction with the internals of the system.

Both modules and scripts may be used interchangeably, however, mod-

ules should represent more permanent components of the system, i.e. class implementations. It remains to be seen how well scripting can be integrated into the system. If there is significant slowdown when using scripts, then scripts will not play an equivalent role to modules in the downloadable and stable system (though nothing would prevent someone from using scripts equivalently on their own).

To save the programmer trouble and promote the use of the components in end applications, dynamic components (including class headers) will also be responsible for containing their own programmer documentation, user documentation, and user interface (see interaction topic). Although this information could technically be done elsewhere or by other components, keeping this information along with the implementation is particularly advantageous. Documentation has always been an infamous aspect of free software and even commercial software. Automated tools, such as Doxygen, have facilitated the concept of source integrated developer documentation, but this idea can be taken further. In many scene graphs and other bodies of software, not only is the documentation sparse (or non-existent), but the user interface is left up to the end application. It seems that there is much duplicated work and an increase in application development time to keep user documentation and the interface out of the scene graph implementation. After all, the scene graph components would generally remain the same in any application; therefore, keeping all of these parts in one place will obligate and make it easier for the developer to document and make a user interface for his system component. Of course, end applications may override or ignore this information.

There will be many components to the system, some of which will implement the same classes. For example, parallel processing is one of the system design goals. Perhaps one user needs threading, another distributed processing, and another has no need for anything parallel at all. If these different needs are to be met, then there will need to be different implementations of the same set of classes. The user would load the implementations based on his needs.

Classes represent the system components through an interface described in a header file. As described above, each class may be implemented in a variety of ways by scripts or modules. For all implementations of a particular class to be interchangeable, they must be able to use just the data structure and function interfaces described in the header. A facility will be provided to allow implementation specific data to be reachable from the class data structure; however, this data will only be useful to the implementation and

should not be used by any other implementation.

Having multiple class implementations leads to the problem of keeping track of the chaos. To prevent system incoherency, all class interfaces may NOT change during a stable system version. It is understood that much restructuring may be done during development versions, but allowing different versions of the same class to coexist would cause too much confusion and administrative trouble. However, there still remains the trouble of having different implementations of the same class.

Each implementation will, of course, contain the name of the class that it implements. It will also contain a set dependency name, implementation name, and version number amongst other information that will be described in subsequent documents. The specific pieces of information above will be used by the database system to keep track of implementations and which ones are to be used.

## 7 Packages

Storing scripts and modules separately in the downloadable copy of the system would be inefficient; there will be many modules and scripts, each of a relatively small size. To promote network transfer efficiency and code organization, related components will be kept in packages. Packages will also contain a general description of the contents and any package specific headers or other data. As shown in the next section, headers should be stored in one place, however, some packages may contain components that implement a class that is not a part of the general system. Perhaps this new class is application specific or an experiment. Whatever the case, the header for a class that is not part of the general system must come along with the package.

## 8 Database

With the possibility of a wide array of classes and implementations, there is a great need to keep track of all of this data. The database is a part of the core that is responsible for keeping track of all the dynamic system components (modules and scripts) but not the core itself.

When a package is installed in the system, the database registers each component of the package and its location in the filesystem. If the pack-

age is compatible with the system version, the next step in registration is to prevent duplication. The implementation version is checked against any database entries that match the class and implementation names. If the implementation version number is the same, then the component need not be installed. Otherwise, this new component is installed.

If the component implements a new class not already registered in the system, then the class's header file must also be obtained and registered. It seemed like a good idea to require headers to be stored in the packages, but there would be much duplication of headers and difficulty in assuring that all headers for a particular class are identical. Therefore, locating and including general header files in package downloads will be a part of the download system. Headers for non-general classes will come along with the package and should be installed with the other class headers. Although this case is an exception, the database should check that there is not already such a class installed. If a class with the same name already exists in the system, the headers must be compared for compatibility with a utility such as diff. Although more sophisticated measures may be taken to check compatibility, it is not necessary. The user will be given the choice to allow the new component's installation by either overriding (suggesting that the conflicting classes are indeed compatible) or overwriting the previous class installation.

After a component has been checked for compatibility, there still remains the condition of priority. With the possibility of so many implementations of the same class, the core needs to know which implementation to load if it is not explicitly specified. There need only be two levels of priority: default and non-default. The user will have the ability to set all or individual components of a particular package as default at installation or run-time. The non-default components may be selected by end applications which would override user settings, but only for that particular application at run-time. The use of the priority is to prevent conflicts and manual implementation choice at run-time.

Class implementations other than the application class, are not allowed to choose which implementations are loaded. There would be a great deal of thrashing if implementations were swapped in and out of the system because of fickle and poorly designed implementations. The general case is that classes and their implementations should only be dependent upon other general classes, but not other implementations. Which implementation of a class is used should be indistinguishable to other implementations that use that class. The other less common case is that a set of implementations are dependent on eachother. Consider the case of a set of OpenGL renderers

or that of a UI interpreter and its widget implementations. Both sets are dependent on a particular external library (a GUI toolkit in the UI case). Having both OpenGL and RenderMan renderers loaded at the same time, for example, will have unpredictable and undesired results when the scene is rendered. A naive solution would be to introduce package dependency in the database; components would only be loaded if other components of the same type were loaded. An improvement to this solution would be to designate a central required component of a set of dependent components. In the case of the UI set, the UI interpreter would be the central component. The other components of the set would only be loaded if the matching interpreter had been loaded. Furthermore, when a central component is loaded, all non-matching components of the previous set must be removed before loading the new set.

The use of implementation sets should work out well with UI's and renderers. However, an extension to this concept appears to be necessary. Consider the OpenGL and Renderman renderers case and their ability to have implementation specific UI's (described in the UI section). A problem arises when the user selects output to RenderMan: the OpenGL display should be maintained whilst the user makes any settings to the RenderMan renderers through the UI. But that situation suggests that either the two renderer sets be interchanged constantly and that only the RenderMan UI be displayed or that the two sets be loaded simultaneously. It may be a good idea to make a further distinction in communications with the long and short term problem. Long-term and short-term renderers would inherit from the general renderer class (become a subclass instead of an implementation) and would be treated simultaneously alongside each other. Perhaps there is another solution to this problem, but the problem is not fatal and will be rectified in the implementation stage.

The database now knows which classes are available, what implementations they have (if any), and which implementations to use. The database will be queried by other parts of the core as components are loaded and unloaded.

## 9 Scene Modeling

Scenes are described by a directed acyclic graph (DAG) as opposed to a tree. But, like a tree, the DAG will have a root node that signifies the beginning of

the description and where traversal should begin. As shown in other systems like VRS [1] and Blender [3], DAG's allow repeated scene components to be shared. In particular, the geometry and surface attributes of objects may be identical in many objects. Making distinct copies of what is essentially the same data, as would be the case in a true tree structure, wastes much space and complicates changes made to the scene. The graph components will be generic nodes that do not describe the scene directly. Generic nodes allow the exchange of specific descriptions like object geometry without the need to alter the actual scene. VRS uses multiple inheritance to combine a few different types (MLeaf, MMono, and MPoly) of nodes into one multi-purpose node in the graph. There seems to be no significant need to make this distinction; therefore, a node in this system will be capable of being a leaf and having one or more children without multiple inheritance. It may be necessary, however, at the implementation stage to make some changes to the organization of node inheritance (but not with multiple inheritance).

In order to facilitate animation, VRS makes a distinction between behaviours and shapes. Shapes use geometry and object attributes such as orientation and surface properties to describe the scene. Behaviours describe how the scene may change with time, inter-object interaction, or user interaction. Behaviours are constraints on object attributes and sometimes objects themselves. As such, VRS's distinction between shapes and behaviours is quite important and there will be a similar distinction in this system. These two components will have different semantics and will be stored in separate DAG's. The shape and behaviour nodes will inherit from the generic node(s) for use in their respective graphs.

The constraints involved in handling behaviours is a difficult topic. Constraints will be considered more fully in later stages of system development. They have only been considered now to note that the system will be designed to allow for constraint solving to be added. Ideas concerning constraints will be drawn from TBAG [2] and particularly VRS.

It follows that the important topic for initial system development is shapes. As stated, shapes are the generic nodes in the scene's shape DAG. The shape graph represents the part of the scene that can be rendered. Shapes convey renderable data by referencing object data (geometry, attributes, et al). Shapes may also reference other shapes to build objects out of subgraphs. Attributes of a shape will affect its subgraph, though, the attributes can be overridden by shapes in the subgraph. Object geometry, on the other hand, will not affect a shape's subgraph as it is specific to that

particular shape. The generic shape class may be inherited and its interface overridden to add new functionality; however, it may be unwise to add new components to the interface and expect the scene traversal algorithm to handle them. Having more than one or a few generic shape node types would defeat the purpose of making them generic: simplicity of traversal. Adding more shape node types is strongly discouraged, unless they can be used to full effect with the inherited interface.

The final topic of modeling a scene is how to interpret or traverse the graph. Both the behaviour and shape graphs should be made of generic nodes for the reason stated above. The algorithm may be as simple as starting at the root and considering the graph one node at a time. More complicated solutions may divide the traversal amongst separate threads or machines for parallel computation. In any case, an important function, as described in VRS, is that of maintaining an attribute stack. Because attributes at one node affect the node's subgraph, the traversal algorithm must know what the current state of attributes is at any given point in the graph.

## 10 Object Data

Because shape nodes need to be generic and object description is an issue separate from the scene graph, there needs to be a class hierarchy for object data. Object data consists of what, where, and how to render the object. "where" describes the position and orientation in the scene's space. "how" describes the other properties of an object such as texturing and coloring. "how" and "where" are considered to be the attributes of an object. The case of "what" includes the physical geometry of an object as well as other, perhaps, non-visual renderables such as sound. Integrating sound as an object is important to adding realism to virtual worlds and allowing acoustic simulations. Example attributes of a sound are volume and direction, whereas the sound would be the raw sound itself.

Object data can be multiply referenced to prevent the waste of system memory. Referencing also makes it easier to change repeated components of a scene as a particular component need only be changed in one place. Of course, if some repeated component is to be made different while other object referers should remain the same, all that needs to be done is to copy and change the data for the intended component.

Another use of the distinction between object data and the scene graph

is that multiple object pieces of object data can be referenced by a shape node. For example, in Blender, one may make different and unjoined surface meshes. A shape may contain any combination of these meshes and shapes may share meshes. Allowing shapes to do this let's the user treat multiple pieces of geometry as one object without having to worry about constraining multiple objects to the same transformation or other attributes.

In that last case, it becomes problematic for users if only shapes may specify object attributes. In the case of a shape having multiple meshes, the user may not want to have each mesh to have the same surface properties. If only shapes could specify properties, then the meshes would have to be in separate shapes; however, that may be rather unintuitive for the user. Therefore, the "what" component of object data may specify it's own attributes. Shape specified attributes may be overridden, complimented, or ignored by that specified by an object. For example, it would make sense for an object geometry's position information to work in conjunction with the shape's position. Object specified surface properties, on the other hand, would do well with ignoring or overriding (a shape might force all of its object geometries to use its surface attributes or it may allow its attributes to be the default).

## 11 Scene Interaction and Useability

A shortcoming of many scene graphs is that they do not consider how the user will interact with the scene or the system in general. Some, like VRS, have the facilities to allow for scene widgets as is often the case in interactive VRML worlds. But scene widgets are generally not enough for modeling, animation, and simulation applications. The reasoning in favor of this omission is that it is an application and platform specific issue. By making that separation, similar applications that allow the user to interact with the scene in similar ways will have duplicated work. Secondly, applications will have more specific code and the inter-application reusability of which will be highly limited.

As previously described in the module and script section, classes and their implementations will contain their own user interface (UI) to promote ease of interaction with the scene. The user interface specification should be generic and not specific to any particular toolkit. It will be an XML script with a set of standard tools like buttons and sliders. Callbacks into the class or implementation will also be specified in the interface specification. Extensions may be made to the user interface by modules and, possibly, the

python scripts.

The interface script will be given to an interpreter that will match the XML description to a particular GUI toolkit and the callbacks. This way, the GUI toolkit used need only be known to the interpreter and its components, but no other part of the system. Using another GUI is as "simple" as writing another interpreter implementation and loading it.

Each class may specify two UI's for global and instance uses. A global UI will affect all instances of the class, which is of use in cases such as rendering. For example, the user might like to globally turn off shadows when rendering. The instance UI would only affect a particular instance of a class. To continue from the rendering example, the user may turn on shadows globally, select a shape, and turn that shape's shadows off.

Similar to classes, class implementations can specify global and instance UI's. Particular implementations may have specific modifiable parameters. For example, an OpenGL renderer might have optimization parameters such as the use of vertex arrays or display lists; a RenderMan renderer may have it's own concerns about which shader to use.

Global UI's are shown together as are instance UI's, but the distinction betwixt class and implementation should be made in some way such as partitioning (up to the UI interpreter). Furthermore, the interpreter has the ability to show or not show certain types of UI. End applications may not want the user to be concerned with different aspects of scene interaction. The application can tell the UI interpreter whether to show the different UI types and where to show them.

Furthermore, in many applications, users interact with scenes in ways other than GUI tools. Users usually are given the capability of grabbing and manipulating scene data with a mouse on the viewport. The only means of interaction commonly available to users are the keyboard and mouse. Many other methods of user interaction with the computer are converted to keyboard and mouse based information so that it is compatible with existing software. Even if three-dimensional input methods become commonplace, they would only add one more dimension to the current two-dimensional mouse input. Therefore, the user's interaction with the scene through the viewport will consist of two or three-dimensional motion and keyboard input.

From this base input information, events are sent to relevant parts of the scene, whether they be the currently selected data or an object that maps to the mouse position on the viewport. Behaviours assigned to the scene will respond accordingly to the user input. These facilities allow for the

construction of scene widgets. These widgets are made of subgraphs of both the shape and behaviour graphs.

These scene widgets are a common concept in VRML worlds, but they can also be applied to object modeling. Consider a user specifying a NURBS patch. The user may position each control point anywhere he wishes. Each of these points would then be a scene widget in this system. The points would be represented visually by a pixel or square object. A behaviour would be attached to the point objects that would respond to mouse events as movement in the scene space. Suppose further that each point could be weighted in the directions of each edge emanating from the point. The weights are commonly represented in applications by more pixel or square objects attached to the control point by a line. The user can move increase or decrease the weights by moving the point objects away from or closer to the control point respectively.

Clearly, scene widgets are a good, if not necessary, way for the user to interact with the scene. To enable this aspect of the UI, the base classes (particularly that of the viewable object hierarchy) will have a function to build a graph of their scene based UI. This scene UI can be enabled or disabled at will. For example, in Blender, there is an "Edit" mode in which one can manipulate a shape's internal geometry instead of the shape's overall position and orientation. Entering this "Edit" mode exposes what is essentially the scene UI so that the user can change the underlying geometry. Leaving this mode hides the scene UI and returns the user to manipulating the overall shape's attributes.

Another aspect of usability is undo/redo. Most application users today expect some kind of application facility to keep track of operations and undo or redo them at the user's command (referred to as undo from no on). However, undo is a complicated task for the programmer and system designer.

Many may argue that this feature is not necessary because the user can easily facilitate it through saving different versions of his work. Blender and other applications follow an automated approach to file version saving. In Blender, it is possible to tell the application to save the scene at specified time intervals. This is a useful feature and an easy, if not brute force, method of implementing undo. However, if the intervals are not fine grained enough, then the user will be frustrated when he makes several changes between intervals and can not return to those just after the last save.

Other approaches to undo have been to devise a core set of system functions that work in both directions (they have an inverse). The Euler operators

described by Mantylae [4] are such an example. However, limiting a system to operations that have an inverse, let alone operations that have an easily obtainable inverse, will severely cripple system functionality or at least greatly increase development time. Furthermore, for such a highly extendable system as described thus far, it would be difficult to guarantee that all extensions to the system can be inverted.

The proposed solution is to use a similar approach to Blender. The entire scene would be saved with a new version number (if the user desires) at the specified interval. The difference from Blender is that all changes in between scene saves will be saved as with regard to the last full save. These smaller saves will be much the same as what the diff utility produces between different files with the exception that the whole scene file need not be generated. Instead, saved class instances will be aware of where in the last saved file they reside. When a small save is made, it only need inquire the changed class instance for its new data and position in the full file. The problem with diff's is that they do not work in reverse (to the author's knowledge). The reconstructed scene would need to start from the full file and apply all the diff's in order to the point that the user wants to go to. This problem is why the diff system would still require full saves at regular intervals.

## 12 Saving and Loading and Networking.

As written on the topic of undo, this system will facilitate the loading and saving of class instances. Saving and loading scene (and even other system) information is another common task amongst applications. In fact, many scene graph systems, such as Performer, offer this functionality. The question then becomes not so much why (as users generally want to save or load information in their applications) but how.

Storing information is a common yet complex task in the world of computer graphics. Graphics entail specific and, often times, voluminous data. XML seems to be the logical choice for storing information in this system. Although the resulting files may be a bit bloated at times, XML is a common standard with free libraries available to both write and read XML information. Because XML is text based, it can be compressed to a high degree using gzip or some other utility. The compressed XML would be the smaller binary form of the file and there would be no need to have separate binary and text formats.

Because each class's data structure contains the only information needed to represent that object (from implementation interchangeability), only the class data needs to be stored for each class instance. The task then remains to actually write or read that information to or from the class instances. XML libraries will be responsible for interpreting the files, but such libraries will know nothing about this system. Secondly, writing code by hand to handle every class in the system would be a simple but laborious task, particularly during development versions when class interfaces are allowed to change.

The goals of this system include development simplification and the promotion of code reuse and maintenance. In keeping with these goals, a tool is necessary to automate the production of general file communication code. Such a tool may sound daunting at first, but consider the fact that all class data information is known through the headers. A parser could extract the data information from a class header. The data types and field names could then be used to extract from or input information to a class instance automatically. Any information that need not be stored could be flagged with a common descriptor in the header files. It should be a worthwhile task to implement such a tool.

The closing note on this section is on how it relates to networking. In networked applications of this system, such as distributed processing and scene coordination betwixt applications, it would be easy to apply the saving and loading mechanisms to data coordination. Such coordination is necessary in the the given situations so that all applications involved are working with the same data.

## 13 How to Make an Application

It may seem that this system directly specifies the application. It does contain much of the redundant information amongst applications with a bent towards modeling and animation. However, the end application has control of what components are used and how much the user can interact with them. The application may also specify its own UI and suppress all UI information in the components. How, then, to build an application?

Any implementation of an application will have an implementation of the generic system application class. Implementations of the generic class will be registered, as usual, in the database. When the user runs the core system, he may specify a particular application or let the system load the default.

The application specifies the overall layout of the system's interface. It will state whether and where certain UI component types will be available (remember scene, global, and instance UI's). It will also specify its own UI component layout. The application will do all this through its own callbacks and XML UI description.

Application modules are loaded in much the same way as any other module, except that they are loaded first. Applications can not shut off their own UI's accidentally (e.g. an application turns off global implementation UI's, which would turn off the application's UI if applications were treated equivalently to other components). Finally, application packages may come with a list of (dis)allowed or required class or implementation types. Such a list would be stored alongside the application module in the system installation tree.

## 14 Animation and Time

Although constraints are being saved for later work, it is important to note how the system will treat time. The authors of TBAG stressed the fact that they consider time as being continuous, but the system samples it discretely. Many animation systems see time as discrete moments such as frames. Frames are very inflexible, though intuitive to animation specification. Therefore, the system will represent time as a continuous phenomena. Any component that needs to consider time may sample it at discrete instances.

For example, a user's animation specification may be done through a spline. The spline's control points would most likely be at regular intervals (frames) and the interpolation of the spline would yield the continuous time. Events, on the other hand, would not be restricted to frames, but could occur at any instant in time.

## 15 Code Distribution

As stated before, system components will be stored in packages and downloaded from a server before being installed on the target system. The CPAN system from Perl seems to be a rather useful method of installation. In that system, when the user tries to install a new package, the local system checks

to see if the package's dependencies are installed. If the local system does not meet the dependency requirements of the package, the local system will query a server (mirror or original) for the required packages, then download and install them. A similar scheme would be quite useful for distributing such a wide array of classes and their implementations. However, this distribution mechanism is not necessary and need not be further considered until later in development.

## 16 Parallel Processing

Another overall system design goal is the ability to integrate parallel processing into the system. It is not essential to the system, but it is an interesting research topic and very useful processing tool.

It should be possible to integrate parallel processing at a low level. The most obvious place seems to be in base object classes like that of the graph nodes. However, this topic should be considered in the early development process even if it is not implemented immediately. If the low level base classes have hooks for parallel processing (i.e. mutex's), then classes that inherit from them may apply those hooks knowing where they will be needed. This way, when parallel processing is actually implemented, there will be less of a need to fit the system retroactively.

It has yet to be shown whether adding parallel processing of one sort or another will require recompiling the core. If it does, then that means that parallel processing implementations will not be as interchangeable as other class implementations. Hopefully, it will be found in the next stage (implementation specification) that recompilation is not necessary. Recompilation of the other system components should not be necessary, however, as they should only require the hooks from the base classes that can be dynamically reachable at run-time.

## 17 Topics that Need Further Consideration

Dimension Standard: This is the specification of a standard for dimensions in the scene space. This is useful for coordinating relative positioning and sizes amongst objects. It is also useful for real-world simulations.

Constraints and General Animation: As written above, constraints will

facilitate animation by linking behaviours and attributes. However, constraints are a difficult subject and need not be considered in detail until a later stage.

Caching: Components of a scene may need computation that is not stored anywhere in that component's class inheritance lineage. Caching may be a useful tool to prevent recomputation, but it is not necessary to consider now.

These topics and others will be considered later in the development process. The next step to follow this document is the implementation specification of the core functionality described herein.

## 18 Citations

[1] Dollner, Jurgen, and Hinrichs, Klaus "Object-Oriented 3D Modeling, Animation and Interaction" Journal of Visualization and Computer Animation, 8(1):33-64, 1997

[2] C.E., G. Schechter, R. Yeung, S. Abi-Ezzi (SunSoft), "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications", Proceedings of SIGGRAPH '94, 421-434 (1994)

[3] Blender [www.blender3d.org](http://www.blender3d.org)

[4] Mntyl Martti, "An introduction to solid modeling", Rockville, MD : Computer Science Press, c1988