# Act: an easy-to-use and dynamically extensible 3D graphics library

WALDEMAR CELES JONATHAN CORSON-RIKERT

Program of Computer Graphics, Cornell Universtity

{celes,jcr}@graphics.cornell.edu

**Abstract**. This paper presents Act, an object-oriented library for projects from specialized 3D graphics applications to simple interactive educational programs. Act meets the needs of both occasional and expert graphics programmers, offering a high-level access to the OpenGL library and additional features including object creation, manipulation, and interaction. An interpreted language allows the Act library to be extended easily and dynamically.

**Keywords.** 3D graphics library, object-oriented interactivity, educational programming, dynamic software extendibility

## Introduction

The ability to create and manipulate 3D graphic objects has become a highly desirable feature for computer applications, from specialized 3D solid modelers and animation systems to simple educational programs. The very diverse domain of these applications creates the need for a graphics library that can be learned and productively used by occasional graphics programmers, while also providing the flexibility and extensibility needed by experts.

OpenGL™ has emerged as a leading cross-platform graphics library [Thompson, 1996]. OpenGL provides many specialized resources for 3D rendering and can be implemented in hardware to accelerate performance [Neider et al., 1993]. However, learning to write applications that use the OpenGL library effectively is not a trivial task. Working at a higher abstraction level is desirable, even for programmers with well-developed graphics programming skills.

This paper presents Act, a simple and small library providing versatile tools to create and manage 3D applications. Act serves as a layer of abstraction above OpenGL, managing all graphics rendering while providing many additional features such as object creation and interaction. The library is independent of any GUI (Graphical User Interface) library. Act is also fully and dynamically extensible through the use of an interpreted language. Our goal has been to provide both expert and new programmers high-level access to OpenGL functionality and speed in an environment conducive to rapid prototyping.

## Related work

The need for a high level 3D graphics API has been evident for many years, and a number of commercial toolkits already offer solutions for the professional graphics programmer. We will discuss our own approach in the context of two other toolkits which seem to have been designed with similar goals to ours. One is the Open Inventor toolkit [Wernecke, 1994a], developed originally at Silicon Graphics, Inc. but now available also on other platforms through other vendors. The second is the ALICE application environment for programming simulations in virtual worlds, developed by the University of Virginia's User Interface Group [Pausch et al., 1995].

The developers of the Inventor graphics toolkit sought to extend the power of earlier immediate-mode and display-list graphics libraries by incorporating support for direct interaction with objects in 3D [Strauss and Carey, 1992]. The toolkit has matured into Open Inventor, a sophisticated collection of 3D objects using a hierarchical scene graph structure. Object types include geometry, attributes, transforms, and groups, in addition to behaviors. Subportions or the entire graph can be written to or read from disk. Normal C++ facilities for extension through object inheritance are supported [Wernecke, 1994b], and advanced programmers can extend the scene graph concept by adapting templates called node kits. The Inventor file format serves with minor changes as the default file format for VRML [VRML 2.0, 1996].

Inventor comes with a viewer offering a high level of initial functionality, but integration into a full GUI library for custom applications remains problematic across platforms. Users who only wish to build a 3D interface for a domain-specific application are likely to find Inventor relatively complex. The extensions to the scene graph structure offer powerful tools, but the scene graph traversal mechanism can be confusing and sometimes restrictive. For example, Inventor event handlers must be integrated into the scene graph structure itself, and functionality depends on correct placement.

ALICE decouples simulation from rendering and implements a more flexible scene graph structure to avoid pitfalls that programmers experience with strictly hierarchical scene traversal. ALICE makes use of an interpreted language (Python) and offers a number of

features also targeted at new programmers, but appears intended as an application framework for virtual reality simulations rather than a more general 3D graphics API.

The Act library offers a simple but versatile toolkit to create and manage 3D objects, using OpenGL. Objects are active, in the sense that their behaviors can be defined, and interactive, in the sense that their behaviors can depend on other objects or on users' actions. The library is also GUI independent, allowing easy integration with the programmer's GUI of choice.

## Features

Act is an object-oriented library implemented in C++ that offers a high level access to the OpenGL graphics library. Using Act, a client application creates, manipulates, and renders structured graphics scenes. An independent GUI system (or simply a window system) must provide a drawable area where OpenGL renders the images. For interaction, the GUI system sends Act 2D (raster precision) events, which Act converts into 3D events. The client application in turn binds these 3D events to object behaviors. Figure 1 illustrates the structure of a typical Act application.

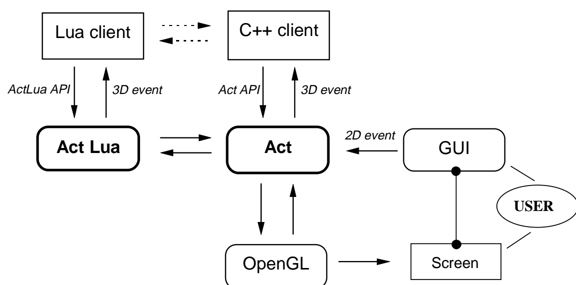


Figure 1: Typical Act application.

Act implements several notable features for graphics scene structure and interaction. Taken together, these features allow any programmer to easily and quickly create sophisticated 3D applications.

- Objects carry their own coordinate systems, which can be directly manipulated;
- The client application can grab events in an extensible way, powerful enough to handle any interaction and program any dependency among objects;
- The library automatically manages and makes available undo and redo resources;
- Through creating 3D logical canvases and constraining the cursor, programming 3D interaction has been simplified without being GUI-dependent;
- The library is accessible and fully extensible through the interpreted language Lua [Ierusalimschy et al., 1996], allowing a very high level of abstraction and supporting rapid programming.

The following sections describe the library's features in greater detail and illustrate through examples how Act may be used by both expert and occasional graphics programmers.

## Scene graph

Graphics scenes are naturally composed of sets of objects arranged into a hierarchy to organize dependencies as well as enhance performance by seeking to minimize the amount of data passed between CPU and frame buffer. These models can be traced at least back to PHIGS [Foley et al., 1996], and Inventor, ALICE, and the VRML Spec structure their scenes hierarchically. Our Act library works in a similar way, offering the programmer different objects which may be combined to compose a hierarchical model of the scene. Figure 2 illustrates the main branches of the C++ class hierarchy used to implement the library.

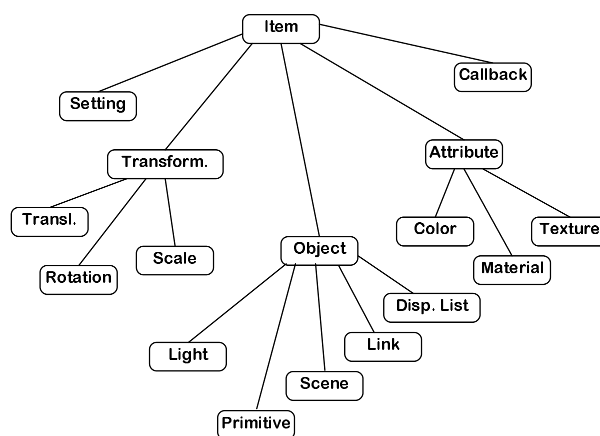


Figure 2: Main branches of class hierachy.

At the topmost level, the *item* represents the base class of any other graphic object, representing anything that can logically be part of a graphic scene. *Setting*, *transformation*, *object*, *attribute*, and *callback* derive directly from *item*. The *setting* class represents global scene rendering settings, such as whether to include fog. The *transformation* class represents geometrical transformations establishing the initial positions of renderable objects in the scene. Transformation objects are cumulative and adopt the same convention of specifying transformations to an object's coordinate system as OpenGL.

The *object* class represents objects having physical meaning or serving as a container for other objects. Each object has implicitly defined a local coordinate system and therefore can be freely transformed without affecting others. When specifying an object transformation, the programmer may choose the most appropriate system to specify the parameters. For example, defining the movement of the hands of a watch

on the wrist of a moving robot may be easier if we choose the watch (not the robot or the hands themselves) as the reference system for our transformation parameters.

The *light* and *primitive* classes have their natural meanings and have their usual derived classes.

The *scene* class represents a container for other items. It can also contain other scenes, which allows the creation of hierarchical models. Except *settings* and *lights*, which have global effect, no item in a scene affects objects outside of that scene, and the previous rendering state is restored after rendering a scene.

Instead of directly creating multiple instances of a single object, the programmer uses the *link* class. Defining a link creates a reference to the linked object, but the link has its own coordinate system. Both the original object and the link object can thereafter be manipulated independently. The use of links (in the place of direct multiple instances) simplifies both the implementation and use of the library.

With links, the data structure needed to implement the scene structure is therefore a conventional tree, allowing backward traversal to obtain a specific rendering state and minimizing the need for using paths to identify an item in the scene. From the programmer's point of view, however, the scene is represented by a graph since links allow multiple instances of items. Following Inventor, we generally call the scene structure a *scene graph.*

The *display lis*t class allows the user to gain performance by combining non-volatile objects into a display list (a concept implemented in OpenGL) as a single unit.

The *attribute* class represents rendering attributes assigned to the renderable objects in the scene. Currently, it may be a simple *color*, a *material*, or a *texture*. An attribute sets the current state value; thus, a new attribute replaces the previous one within any scene in the hierarchical scene graph.

Finally, the *callback* class allows a program to attach a function at an appropriate point to the scene graph to be invoked whenever the scene graph is traversed. Therefore, callbacks introduce procedural mechanisms to change the description of the scene.

## Event handling

Act uses an event-driven approach to define object behaviors. There are two kinds of events: *editing events* and *interaction events*. As would be expected, editing events are generated by editing operations, while interaction events result from operations which do not directly change the state of the scene graph.

The library itself generates editing events corresponding to changes in the scene graph or changes in the objects themselves. For instance, setting an object field (e.g., the radius of a sphere), applying a transformation to an object, and inserting an item in or removing an item from a scene will each generate a corresponding editing event. Any application built on top of the Act library (a client) can then grab these events to create and update dependencies among objects. As an example, animating the movement of a train may be achieved by initially programming each car to follow its leader, and then animating only the lead car.

Each editing event will also generate an entry into the undo buffer, which is automatically managed by the library and made available to client applications. To extend the library, clients can also register additional editing and interaction events and notify Act of their occurrence. These events are then handled by Act's internal event dispatching mechanism. Clients may also post events to the undo buffer if appropriate.

The ability to register new events not only allows for extensibility but also helps to guarantee portability across platforms. For example, the library itself does not know about external interaction events (such as mouse or keyboard events); it is the binding GUI's responsibility to register and notify such events. Once bindings are established for a GUI library, the binding becomes a reusable extension of that library. If the client chooses the external events' bindings with care, one can easily interchange one GUI library with another one, as long as the new GUI library is capable of generating the same sort of events.

The client binds an event to a resulting action by registering a callback function to be called whenever the event occurs on an object. For editing events, this callback has two purposes: validating the action and creating dependency among objects. The library itself does not have knowledge about the semantic meanings invested in objects by the client; turning event validation over to the client allows greater control and flexibility for applications [Celes, 1995]. For instance, applying a transformation to an object may be prohibited if the client does not allow objects to interpenetrate.

The way the client creates an event binding between an object and a resulting action adapts the Tk approach [Ousterhout, 1994]. The object to which the binding applies can be a single instance or a class name such as *sphere* or *primitive*, in which case the binding applies to all instances of that class. An instance-based binding is more specific than a class-based binding, and a derived class binding is more specific then a base class binding. Whenever an event occurs, the most specifically matched binding is triggered first. Then, depending on the callback return value, the same event is sent up the affected object's class hierarchy to the base class bindings. The client can choose whether all

matched bindings should be triggered. As recognized by Tk developers, sending the same event upward may be important for allowing clients to create instance-specific behavior without interfering with general class behavior.

We also extend this concept of binding to deal with our hierarchical model. If an object has no binding for an interaction event, that event is transferred to its parent object in the scene graph. A client program can therefore manipulate complex objects by creating bindings to their root scenes, instead of creating bindings for that event on each of the different objects that compose the scene.

Event bindings and undoable actions are stored in buffers selectable by the client. The client application can create separate event buffers for each of several interactive modes. Each application mode defines the event bindings it needs, and event bindings applicable to other modes do not interfere when that mode is active. While in a walkthrough mode, for example, there is no need to know which events are bound by other modes. Hence, interactive modes can be reused across applications.

Being able to create multiple undo buffers also allows the client to edit different scenes independently, or to use the undo/redo resources to handle temporary actions (such as interactive tasks) without interfering with the main undo/redo resources. For example, to provide feedback while moving or rotating an object interactively, it may be useful to activate a temporary undo buffer to transform the object, display it, and then undo the transformation each time feedback is requested.

## Three-dimensional canvas

To be able to interact with the objects in the scene, the Act library provides three-dimensional canvases, which the client program binds to underlying two-dimensional windows that support OpenGL. The Act three-dimensional canvas transforms mouse positions from the GUI into a three-dimensional cursor.

The three-dimensional cursor is defined by its 3D position and orientation. When unconstrained, the 3D cursor position “maps” to a point on the surface of the closest object in the scene, with its orientation corresponding to that surface normal. To automatically find this point on an object surface, the library traces a ray from the raster cursor position through the scene. The tracing operation considers only objects in the scene that can respond to the event being dispatched. For instance, while moving the mouse, only objects bound to the "moving mouse" event will be considered. This is done automatically so that the client does not need to enable/disable objects to gain efficiency in tracing rays through the scene.

The 3D cursor may also be constrained, which greatly facilitates programming interactive tasks. Using the library core classes, the cursor can be constrained to move on a virtual plane, in a virtual line, on a virtual sphere, or along a virtual circumference. Constraints may be created to apply to individual objects, and when activated restrict cursor motion according to the geometry defined with the constraint. For example, a client may use constraints to restrict moving an object on a plane, rotate it about a center point, or to create specialized manipulators.

When the client specifies a 3D canvas, it must also create a *camera* to view the scene. Each 3D canvas can have only one camera attached, but a scene may be displayed simultaneously in several canvases.

## Interpreted language binding

The Act graphics library is a C++ library and any C++ application can access the library features described above. For programmers comfortable in C++, any extension can be implemented by creating new derived classes conventionally. Direct calls to OpenGL functions may also be integrated if necessary.

However, occasional graphics programmers need easy access to (sometimes sophisticated) graphics features to create front-end interfaces for their domain-specific applications, or to illustrate concepts by graphically simulating simple models (e.g., for educational purposes). The vast range of applications that such a library could benefit highlights the biggest challenge of its design: how do we create a graphics library that at the same time meets the needs of both expert and occasional graphics programmers?

The Act library meets this challenge through the use of Lua, an interpreted language that combines data description facilities and conventional procedural features, using a clear and simple syntax. Lua also provides mechanisms to support object oriented programming and to extend its own semantics, being smoothly integrated with C++. As an example, the code below represents a valid Lua construction that, with the appropriate binding to the graphics library, creates a scene and stores it in the variable `myScene`. Then, any *scene* method may be called through `myScene`.

```
myScene = Scene {
   PositionalLight {position={0,10,0}},
   Material {ambient={1,1,1},
             diffuse={1,0,0}
   },
   Sphere {radius=2.5}
}
...
myScene:realize()
```

In addition, Lua supports all the conventional control structures, including expressions, loops,

conditional statements, and function calls, and all of them can be combined in defining the scene graph.

Even more important is the ability to easily create auxiliary data structures, besides the scene graph itself. The scene graph with its hierarchical structure is adequate to describe the scene and support rendering. There may be clearer and more efficient ways to traverse a data structure for other purposes, however. For example, suppose we are creating and animating a model of the solar system, with its planets and corresponding moons. We create a scene graph to represent the model, but, instead of traversing the entire scene graph looking for objects that should be animated, it is much easier to create an auxiliary data structure to manage when, how, and which objects should be animated. For each time step (perhaps triggered by a separate simulation process), we traverse the auxiliary data structure to position the planets and moons, and then issue a single "redraw canvas" command to the Act library, which renders the scene in its current state.

Lua provides associative arrays that can be used to implement not only ordinary arrays but also symbol tables, sets, records, etc. This makes the creation of auxiliary data structures very natural, even for occasional programmers. In fact, any graphic object in Lua is represented by an associative array (a *table* in Lua) and thus can store any other field, besides those used by the graphics library.

From Lua, one can access any feature of the C++ library. We have also bound almost all OpenGL functions to Lua. Expert programmers can combine direct calls to OpenGL with calls to the graphics library, as in C++. One can also fully dynamically extend the library. Using Lua, programmers can create a derived class from any class in the C++ library and access its base methods or redefine them. Because these extensions are done using an interpreted code, they can be dynamically loaded by any other client.

We clarify Act's features for novice, intermediate and expert programmers by describing alternative implementations for a chess game using the interpreted language. In the following code fragments, we illustrate different ways to represent pawns, which for the sake of simplicity we build from a cone with a sphere on the top (Figure 3).

A novice could collect a cone and a translated sphere into a scene named "pawn", and then create 8 instances of each color piece via links to the original.

```
-- Create one original pre-defined scene
pawn = Scene {
   Cone {radius = RADIUS, height = HEIGHT},
   Translation {0,HEIGHT,0},
   Sphere {radius = RADIUS/2}
}

-- Instances are created using links
one_instance = Link{pawn}
```

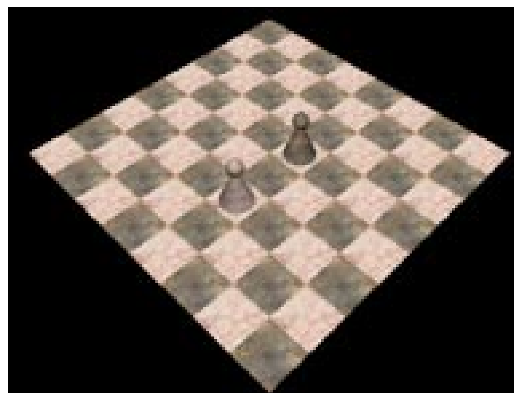

Figure 3: Chess game interface.

This approach works fine for rendering the pieces and can support interactive movement, but more advanced game features such as movement validation would have to be created and managed for each instance of each game piece.

A more experienced programmer would gain several advantages by creating a new class for each type of piece. A new pawn class could be derived from the scene class, with a cone and a translated sphere automatically inserted into it by the constructor.

```
-- Create a pawn class deriving from scene
classPawn = ActClass {name = "Pawn"}

-- Class constructor
function Pawn (self)
   -- create corresponding C++ object
   local obj = LuaActScene:new(classPawn)
   -- automatically add children
   Cone {radius=obj.radius,
         height=obj.height,
         scene=obj
        }
   Translation {0,obj.height,0; scene=obj}
   Sphere {radius=obj.radius/2, scene=obj}
   -- add any additional initialization
   ...
   return obj
end

-- Instances are created using the new class
one_instance = Pawn {radius=0.4, height=1.0}
```

Therefore, because the pawn has its own class, pawn behavior (for example, movement validation) can be defined through class-based event bindings, promoting modularity and allowing reuse.

To gain performance and minimize storage, an expert programmer could take one further step and create a new primitive class, directly calling OpenGL functions, instead of using the existing cone and sphere primitives. At this point, after prototyping each piece, it may also be worth converting to C++ for improved performance, although the interpreted language will support all three approaches.

## Applications and extensions

To illustrate the versatility and applicability of the library, we will now describe some of the current research and education projects using Act. The applications presented here access conventional widgets such as menus using TkLua, a library to access the Tk toolkit from Lua [Figueiredo et al., 1996]. Some of the applications are C++ programs accessing the C++ graphics library. Others are purely Lua code, accessing the Act graphics library through an interpreter that simply initializes the library and executes the interpreted code.

### Medical visualization application

The first example is a C++ application that only accesses the C++ Act library and does not use the Lua extension. The application reconstructs a 3D model of human arteries based on x-ray images (Figure 4). The 3D artery model is reconstructed using data from x-ray images, which must first be correctly positioned in 3D space.

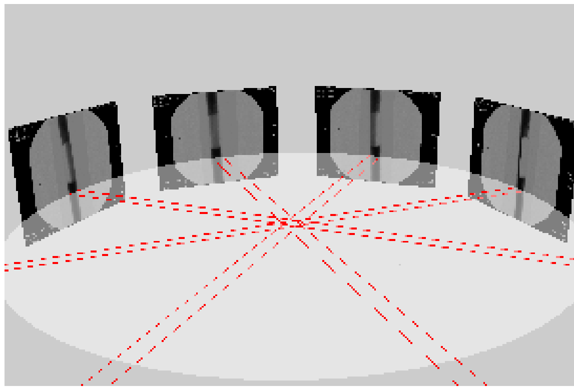

Figure 4: Application to reconstruct a 3D artery model.

Once fixed in 3D space, x-ray (raster) values are mapped to a global 3D coordinate system using an appropriate hierarchical model to represent the scene. In addition to providing objects and event handling for a 3D interface, the Act graphics library makes it very easy to deal with different coordinate systems using the library resources, freeing the application itself from algebraic transformations.

### Animated solar system model

We have developed an animated model of the solar system with its planets and major moons for educational purposes, written entirely in Lua so as to be accessible for modification and extension by high school teachers. Act's hierarchical model for describing a scene is very useful for this application, which gains simplicity from dependency among planets and their corresponding moons. The following code illustrates the definition of the scene graph to represent the earth and its moon.

```
sun = Scene {
  ...,
  Scene { -- Earth and its moon subscene
    Texture {image=Image{"earth.bmp"}},
    Sphere {radius=1.0, name="globe"},
    Scene {  -- moon
      Texture {image=Image{"moon.bmp"}},
      Sphere {radius=0.2725};
      name = "moon",
      eccen = 0.055, smAxis = 385/AU,
      inclination = 5.0, period = 27.322
    };
    name = "earth",
    eccen = 0.017, smAxis = 1,
    inclination = 0, period = 365.256
  },
  ...
}
```

With Act's hierarchical model, we animate each moon in relation to its planet in the same way that we animate each planet in relation to the sun. From the moon we can consider the earth as a fixed reference, so when animating the earth's orbit about the sun, the moon (hierarchically below the earth) will automatically follow the earth movement. This dependency does not preclude us from revolving the earth sphere (rather than the whole earth scene) independently about its own axis without affecting the moon.

Note that we can store any desired field in an object in addition to the library's graphics attributes. For the solar system model we store data to compute the position of the object (period, orbit major axis, eccentricity, and orbit inclination). As discussed earlier, we constructed an auxiliary data structure storing a list of all planets and moons for purposes of animation.

### A simple 3D modeler

We created a simple 3D modeler entirely written in Lua to demonstrate the use of the library and allow scenes to be created for educational projects (Figure 5).

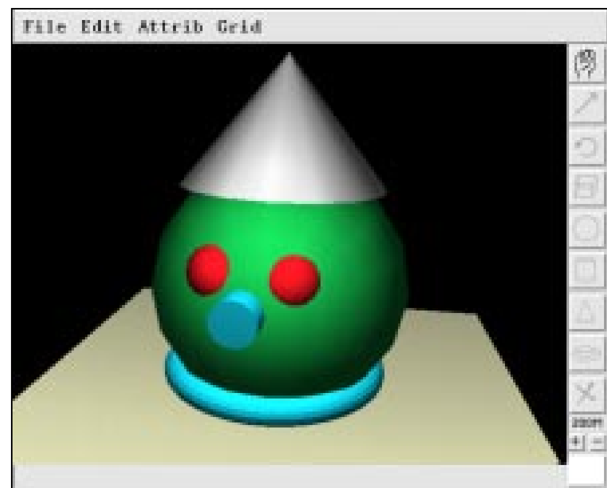


Figure 5: A simple 3D modeler.

The program allows the user to interactively create primitive shapes (cuboid, sphere, cylinder, cone, and torus) with a color or pre-defined texture, translate and rotate shapes, and undo or redo any action. Including all Tk widgets, the modeler requires only 1100 lines of Lua code.

The following code illustrates how the modeler implements the interactive mode to rotate objects about their origins. Whenever the mouse button is pressed on an object, the cursor is constrained to move on a virtual sphere centered at the object's local system origin. When the user moves the mouse, the program provides feedback displaying the object at its new position. However, the rotation is only really performed when the user releases the button. The main code calls the function to create the event buffer and turns it active whenever required. Note that this code can be loaded and used by any other applicacation to provide the same interactive mode.

```
-- Creates constraint and temporary undo
rot_constr = OnSphere{center=Triple{0,0,0}}
rot_tmpUndo = Undo {10}

-- Press button
-- Constrains the cursor.
function rot_bpress (obj,data,cv)
 rot_constr:constrainer(obj)
 rot_constr:point(data:point())
 cv:constraint(rot_constr)
 return Act.OK
end

-- Button motion
-- Rotates object, redraws scene for
-- feedback, and undoes the rotation.
function rot_bmotion (obj,data,cv)
 local prev = ActUndo:current(rot_tmpUndo)
 obj:rotate(data:angle(),data:normal())
 cv:redraw()
 rot_tmpUndo:undo()
 ActUndo:current(prev)
 return Act.OK
end

-- Button release
-- Really rotates the object and redraws the
-- scene. Then it removes the constraint.
function rot_brelease (obj,data,cv)
 obj:rotate(data:angle(),data:normal())
 cv:redraw()
 cv:constraint(nil)
 return Act.OK
end

-- Creates and sets event buffer
-- The given canvas is passed as an extra
-- parameter to the callbacks.
function rot_createMode (cv)
 local event = Event {}
 event:bindInteraction(
  "ButtonPress1","Object",rot_bpress,cv)
 event:bindInteraction(
  "ButtonMotion1","Object",rot_bmotion,cv)
 event:bindInteraction(
  "ButtonRelease1","Object",rot_brelease,cv)
 return event
end
```

Act supports saving screen images to a file and saving and reloading scenes; the metafile used to store a saved scene is simply Lua code describing the scene. For the program to support reloading stored scenes, it is enough to include a command to execute the saved metafile. This is the same approach already used by the EDG system [Celes et al., 1995].

### Collision detection

We have demonstrated Act's support for handling external events through an extension for collision detection using the RAPID library [Gottschalk et al., 1996]. We first register a new 'collision' event with Act. Each time a new primitive is inserted in the scene, we then grab the resulting event, build a corresponding collision detection structure (an OBB-Tree with the RAPID library), and associate it with the primitive. Whenever a primitive is later edited or transformed, we grab the event and check if performing such operations would result in object inter-penetrations. If our extension library detects a collision, we notify Act of this 'collision' event, which Act makes available to the client application. The application can then determine whether to allow object inter-penetration, using the event's returned value to report its validation in the same way any other editing event can be handled.

### Rigid body physical simulation

We are currently integrating the Act graphics library with a rigid body physical simulation library. The goal is to create a simple virtual laboratory where teachers and students can experiment with physical concepts. Aside from improving the accuracy and efficiency of the physical simulation itself, the challenge here is to obtain an easy-to-use interface so that teachers and students can build their own models for simulations.

While a 3D interactive environment described in Lua or created using our simple modeler will appeal to both teachers and students, the ability to establish appropriate constraints to focus interactivity may be the most critical factor for successful educational applications. To introduce the concept of trajectory, for example, a teacher can construct an experimental game where students try to score a basket. Because Act supports controls constraining the students to change only the direction of the shot, not the starting point or initial velocity, the game can focus on the effect of the initial angle on the ball trajectory. Without low-level library support for constraints, it would be a very challenging task to program appropriate interactivity.

## Conclusion

Developers of 3D graphics applications face several programming challenges, and 3D interfaces are often omitted because of the skill and effort required to build them. A tool at a high abstraction level is necessary to provide the appropriate framework for 3D graphics application development.

The Act library is very small when compared to Inventor or VRML applications, yet is quite versatile. Using Inventor, achieving the level of extensibility we have described would require an expertise in C++ as well as “a serious commitment and a reasonable effort to master” [Shekhar and McGinley, 1994].

ALICE and several VRML modelers provide another way to develop interactive 3D environments at a good level of abstraction for non-experts, and offer better support for creating virtual reality applications. However, as they are not general graphics libraries, programmers cannot easily use or extend these tools to create a 3D interface to embed within their own existing applications.

The Act library combines power and versatility. It is a useful tool for anyone with a minimum of programming knowledge who wants to create 3D interactive applications, while offering rapid prototyping in an interpreted language and the capability for extensions to meet any domain-specific need. Act can serve as a toolkit for introductory computer graphics programming, facilitate the development of interactive 3D educational applications, and to meet a range of needs for developing OpenGL applications.

### Future work

We still have much to do in improving the efficiency with which scenes are rendered, and in adding features to fully cover the OpenGL library. We also plan to use the Act framework and Lua to implement a visual programming environment to create 3D applications.

## Acknowledgments

During the development of this research, the first author held a post-doctoral fellowship from the Brazilian Council for Scientific and Technological Development (CNPq). Support for both authors was provided at the Cornell Program of Computer Graphics through the National Science Foundation Directorate of Educational Human Resources and the Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219). Much of the research was performed on computers generously provided by the Hewlett Packard Corporation.

## References

W. Celes, L. H. de Figueiredo, and M. Gattass, "EDG: a tool to easily create interactive graphic interface" (in Portuguese), VIII SIBGRAPI, 1995.

W. Celes, "Customizable modeling of hierarchical planar subdivision", Ph. D. Thesis (in Portuguese), Computer Science Dept., PUC-Rio, 1995.

L. H. de Figueiredo, R. Ierusalimschy, and W. Celes, “Lua: an Extensible Embedded Language”, *Dr. Dobb’s Journal*, #254, pp. 26-33, December, 1996.

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics, principles and practice*, 2nd edition in C, Addison-Wesley, 1996.

S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: A Hierachical Structure for Rapid Interference Detection", *Computer Graphics*, ACM SIGGRAPH, pp. 171-180, 1996.

R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua: an extensible extension language", *Software: Practice & Experience*, 26 (6), pp. 635-652, 1996.

J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.

J. K. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, 1994.

R. Pausch, T. Burnette, A.C. Capeheart,M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, J. White, “Alice: Rapid Prototyping System for Virtual Reality”, *IEEE Computer Graphics and Applications*, May 1995.

R. Shekhar and B. McGinley, "Open Inventor 2.0", *Computer*, v. 27, July, pp. 100-102, 1994.

P. Strauss and R. Carey, “An Object-Oriented 3D Graphics Toolkit”, *Computer Graphics*, ACM SIGGRAPH, 26 (2), July 1992.

T. Thompson, “An inside look at the most popular 3D environments: OpenGL, QuickDraw 3D, and Direct3D”, *Byte*, June 1996.

VRML 2.0, *The Virtual Reality Modeling Language Specification*, Version 2.0, ISO/IEC CD 14772, 1996.

J. Wernecke, *The Inventor Mentor*, Addison-Wesley, 1994.

J. Wernecke, *The Inventor Toolmaker*, Addison-Wesley, 1994.