The Anima Animation System, Version 2.0

Danielle Sauer, Jared Gabruch, Cara Gibbs,
Howard Hamilton, Jason Selzer,
Chris Sembaluk, and David Thue
Technical Report CS-2003-04
October, 2003

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0453-4

Table of Contents

The Anima Animation System, Version 2.0

Danielle Sauer, Jared Gabruch, Cara Gibbs, Howard Hamilton, Jason Selzer, Chris Sembaluk, and David Thue

Department of Computer Science
University of Regina
Regina, SK, Canada S4S 0A2
{sauer1da, hamilton}@cs.uregina.ca

Abstract

Anima is a software system, written in Java, for creating animations. It is designed for research into the Computer Animation, Computer Graphics, and Knowledge Discovery fields. This document outlines key aspects of the system. Major features include the ability to create and animate scenes that can be rendered into an AVI movie, to shape objects, to add visual effects such as lighting, to keyframe object movement, and to change an object's parameters and attributes. Some significant features of Anima are (1) the ability to do precision picking, (2) allowing the user to define his/her own control points for curves, (3) the manipulation of objects using handles, (4) merging and smoothing surfaces, (5) path following for both objects and the camera, (6) particle systems, and (7) dynamic animation. This document provides detailed information on object picking, textures, object manipulation, and particle systems. In each case, the main concept, the crucial aspects of the implementation, and any problems discovered are described.

1. Introduction

The use of animation is growing in the film and game industries. Animation software has become an important Canadian industry, due to companies such as Alias-Wavefront, Discreet, and Softimage [4]. These three companies controlled 65% of the US$146-million world market for professional 3-D animation software packages in 2001. Their systems are also being used in the architectural and industrial design sectors [4]. Animation has created technical jobs and excellent research opportunities, bringing some of the best people in this field to Canada, while enticing Canadians to remain.

The University of Regina's Anima animation software package was a result of the effort of six students, under the supervision of Howard Hamilton, working together to create a framework for doing animation. This framework will also serve as a basis for future research. The initial version, Anima 1.0, was designed by Cara Gibbs and Howard Hamilton and written in Java by Cara Gibbs in January-April 2003. This version used 2-D graphics and had object scaling and translating functionality, curves, a gravity force, and inherent object motion. The second version, Anima 2.0, was designed and written by Jared Gabruch, Danielle Sauer, Jason Selzer, Chris Sembaluk, and David Thue using the Java 3D libraries to create the scenes. A large majority of the code from Anima 1.0 was replaced or changed when it was converted to 3-D.

The remainder of this document is organized into three sections. Section 2 discusses the animation software, the purpose of Anima, some features, plus well-known examples that are used in the industry today. Section 3 covers the overall design of the Anima software, as well as the detailed implementation design of four features. Lastly, the conclusion addresses significant achievements of the project and possible future directions.

2. Animation Software

An ***animation software package*** is a collection of tools used to create scenes and computer animations. Its purpose is to extend an artist's capabilities and to make it easier to create digital animation. Some main features found in existing systems include modeling, visual effects (dynamics and particles), fluid effects, hardware and software rendering, and curves. Animation software packages are utilized extensively in the film and game industries. Some existing commercial systems are Alias-Wavefront's Maya, Discreet's 3d Studio Max, Softimage's Creative Environment, and Curious Lab's Poser. Maya, 3d Studio Max, and Softimage are the top three general-purpose animation systems in the world today and all three are Canadian based! Poser is well known for its ability to create and animate human figures. All four systems, while similar in nature, have areas in which they excel over the competition. They are often used together in the same projects and even the same scenes. Movies such as Spider-Man, Harry Potter, the Star Wars prequels, and the Lord of the Rings trilogy used 3-D animation to enhance the visual appeal and lifelikeness of the picture.

The general purpose of the AniMiner project is to create tools to aid in the process of computer animation. Anima is a part of this project. Other tools created for the AniMiner project will be able to mine data, such as pictures of real objects, and set the initial properties of the system automatically to create realistic looking animations. This is unlike many of the large and widely used animation systems of today where the users have to configure these properties themselves to set up their scene. Many of the features of the project were inspired by larger systems already in circulation, though they may have been given a different twist or used in a different way. These features include curves, lighting, textures, meshes, surfaces, particle systems, and path following.

3. System Design

This section describes the overall design of the Anima software package. It is organized into seven subsections. The first subsection provides an overall look at how the major pieces of the system are arranged with respect to their class structure, as well as how they work together. The second subsection tells how the system is set up from a programming perspective. The third subsection discusses the main features found in the system, while the remaining four subsections, object picking, textures, object manipulation, and particle systems, cover specific details.

3.1 General Design

The Anima system consists of four major components, the Interface, the SceneGraph, the Canvas3D, and the Objects, as shown in Figure 1. Each of these components is now described.

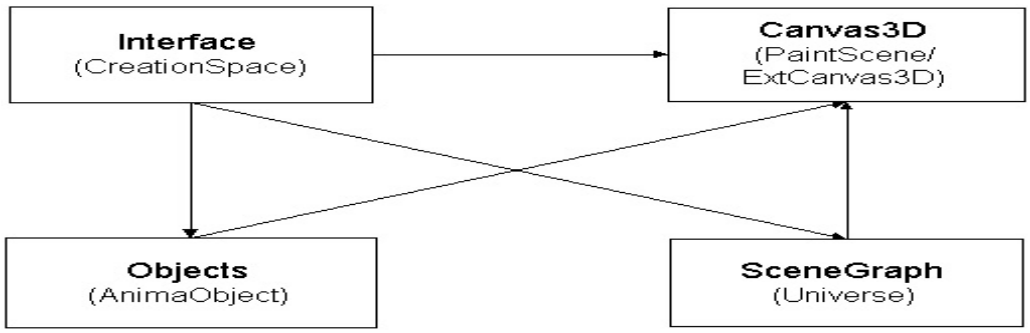

Figure 1: General Design

The first component is the Interface, which deals mainly with the CreationSpace class in the code. This is where all the pieces of the system are brought together and through which the user accesses the other major components. All the tab buttons and menu options are implemented in this section. The interface also includes the Attribute Editor element, which allows the user to manipulate objects in the scene by typing in values.

The second component is the SceneGraph, also known as the Universe class. Next to the CreationSpace class, this is the most important class in the system. Objects are added to the scene through the SceneGraph component and almost everything in the system must deal with it. It works as a go-between for the other classes and it contains information such as the list of currently chosen objects and the handle being used. It also contains the Object Picking element and the Object Handles element. The Object Picking element allows for the user to choose an object and the Object Handles element allows for the user to manipulate the given object by rotating, scaling, or translating it.

The third component is the Canvas3D, which deals with the PaintScene and ExtCanvas3D classes. Four different instances of this component are created, one for each view: top, side, front, and perspective. The major element of this component is mouse handling. The PaintScene class contains all of the mouse functionality, aside from object picking. It is used to manipulate both objects and the camera based on where and how the mouse is moved. It distinguishes the difference between a mouse click, the mouse being dragged, and various other mouse functions and will behave according to the implementation in them. The ExtCanvas3D class is used to "paint" objects onto the screen and deals with the timeline. It is also used to render movies based on the scenes. These canvases are used throughout the system, especially in the classes that deal with mouse functionality.

The fourth and final component is the Objects, which deals entirely with the items the user can work with in the system. These items include primitives (shapes), meshes, lights, curves, and surfaces. Also found in this component are the attributes and appearances of these objects, along with the methods used to generate them. Many of the classes used to create these objects inherit from the AnimaObject interface or use its functions through the AnimaObjectMethods class. All of the items can be generally classified as AnimaObjects, which makes this component easier to use throughout the system.

As shown in Figure 1, all components are connected to both the Interface and the Canvas3D. The Interface calls the other three components because all commands from the user must pass through it to access all other parts of the software. The Canvas3D is used by the three other classes, either due to mouse handling or because it contains the canvases that objects are placed on. The SceneGraph and Objects component are not connected because the SceneGraph does not deal with the objects themselves, except when adding them to the scene. It never creates a new instance of an object, because the object must already exist in the system for the SceneGraph to be able to use it.

3.2 Implementation Design

The underlying design of the system, as seen in Figure 2, is a tree structure, which is common in Java 3D[1]. Generally only one Virtual Universe (VU) is used in a Java program and the Locale object underneath it acts as a marker to determine the location of objects in the virtual universe. The two main branches of the tree are the View Branch and the Content Branch. The View branch, which is shown in more detail in Figure 3, specifies the viewing parameters, such as the viewing direction and location. The Content branch, shown in Figure 4, specifies the contents of the virtual universe, such as geometry and appearance. A BranchGroup is used to place an object in the scene and the TransformGroup below it holds geometric transformations such as rotation and scaling, which are represented by Transform3Ds. Typically when adding a new feature to the software, the Transform3D is constructed and then added to the TransformGroup, which is then added to the BranchGroup. Both the BranchGroup and the TransformGroup can have more than one child underneath them. With respect to the TransformGroup, any manipulation done to it affects all of its children in the same way. Each branch in the tree must contain a BranchGroup and a TransformGroup.

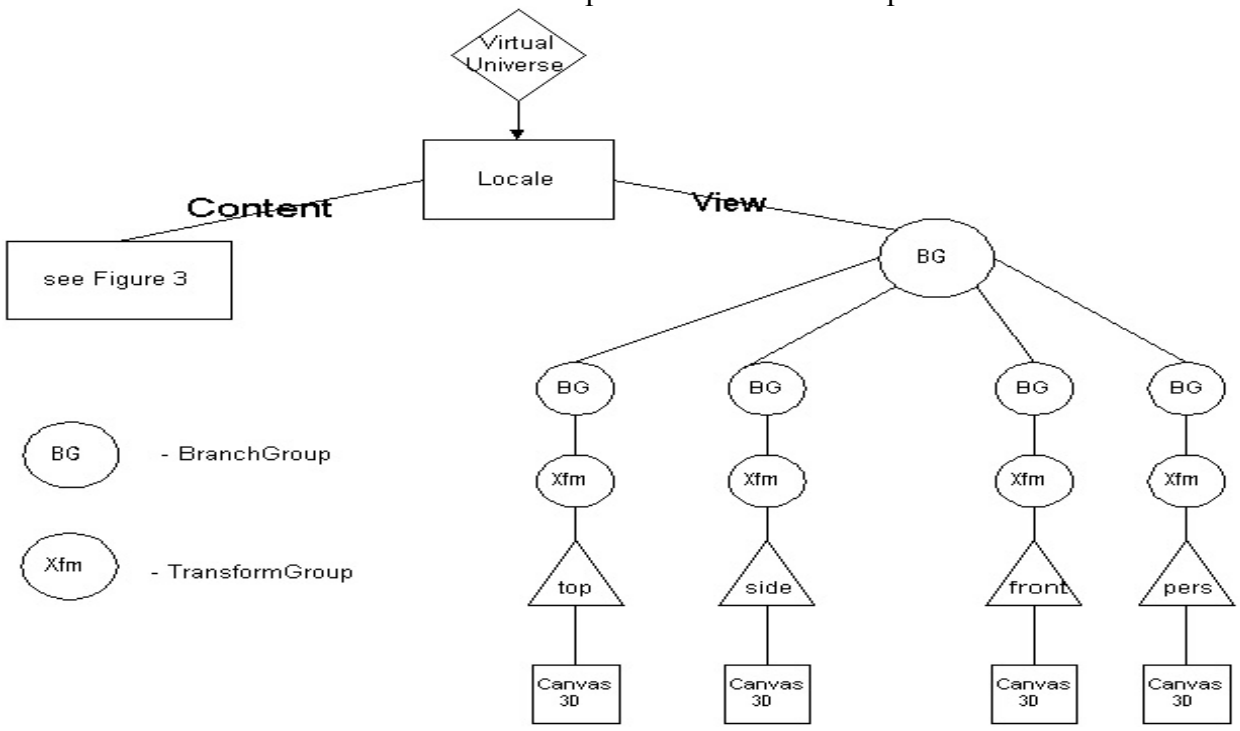

Figure 2: Overall Design of Implementation

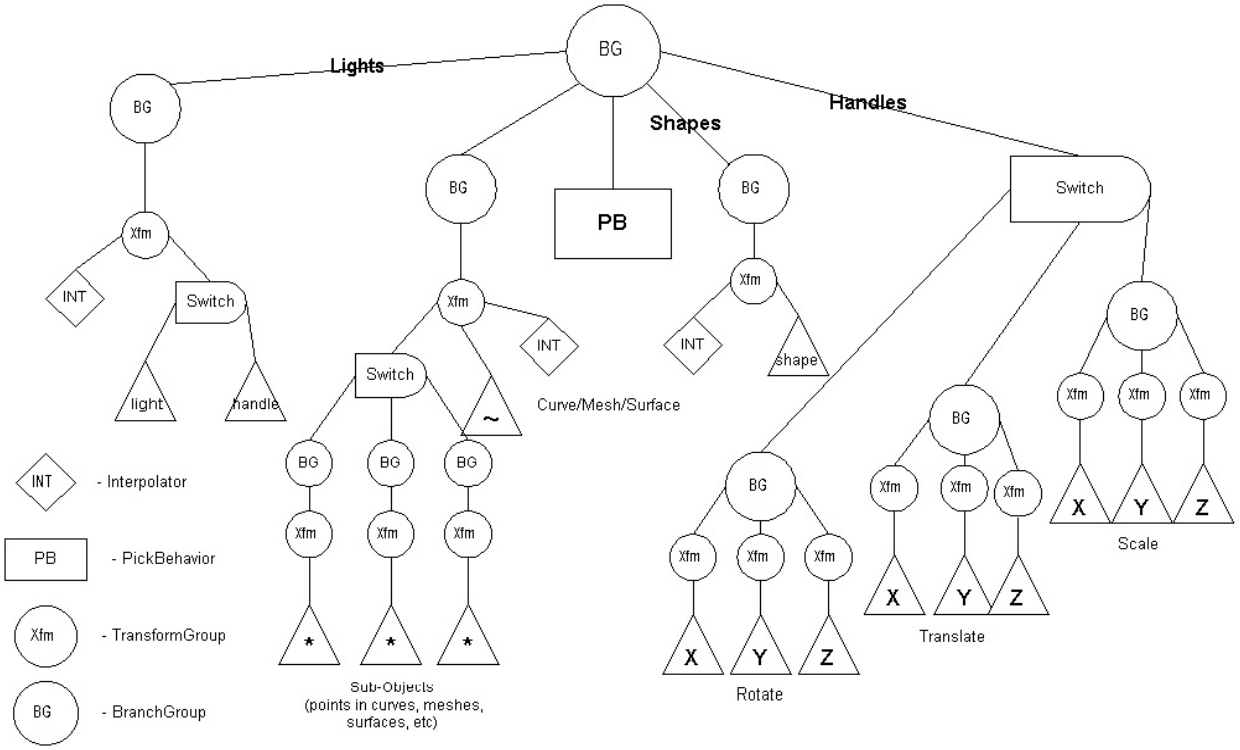

Figure 3: Content Branch

Within the Content Branch are various other branches representing objects, such as lights, curves/meshes, shapes, and handles. Within the branches for lights, curves/meshes, and shapes there is an interpolator. This interpolator is used to manipulate the timeline and to run an animation. It interpolates between the keyframes by filling in values. It is necessary only for objects that can be keyframed, which is why the handles do not have an interpolator object in their branch. The PickBehaviour object is used for object picking and is created only once, when the Content Branch is created.

Other than the interpolator, the Lights branch contains a Switch node that allows for turning on and off the objects below it (the lights and the light handles). The light handle is necessary so that the user can manipulate the lights in the scene. It is a visual object that the user can see and therefore it is easier to comprehend how to place the lights in the scene. The lights object is the lighting itself, which is moved and manipulated according to the light handles. The Curve/Mesh branch contains both the curve/mesh object as well as a Switch branch that includes branches for each of the different control points used in the curve/mesh object. These control points are denoted as sub-objects in both Figure 3 and the program's implementation. The Shapes branches are used to denote the objects in the system other than those already mentioned. This includes cubes, spheres, cylinders, and cones. Each new object has its own branch so that it can be manipulated separately from the other items in the scene. The last branch is the Handles branch, which is used for manipulating objects through the use of three handles. Unlike the other branches, the Switch node is first in the branch, with three BranchGroups following, one for each set of handles (rotation, translation, and scaling). Within these BranchGroups are separate

TransformGroups and objects for the X, Y, and Z handles for each respective set. This is because each handle will need to be changed separate of the other handles. The Switch node is used for turning on and off sets of handles, as only one set should be seen and utilized at a time.

Within the View Branch are four BranchGroups and four TransformGroups, one for each canvas in the scene. Three of the canvases (top, side, front) are orthogonal, meaning that the third dimension is not shown, and therefore they appear to be 2-D. The canvases dictate where the objects show up and within what bounds they can be manipulated.

3.3 Main Features

An overall look at the running system is shown in Figure 4. The tab menu at the top has labels for Shapes, Curves, Meshes, Patches, Lights, and Particles. The major objects of the system can be found through these tabs. The timeline is found at the bottom, and the three buttons on the left hand side of the screen are used to switch between handle sets. The attribute editor is on the right hand side, with the object picking combo box just above it.

Four major shapes that can be manipulated to create scenes are cubes, spheres, cylinders, and cones (see Figure 4). Two specific types of curves can be created: B-Spline and Catmull-Rom. The creation of these curves involves the user specifying the control points by clicking on the screen for each point and then pressing enter when the curve is complete. Curves can be used as paths that objects or cameras are keyframed to follow. There are four different types of lighting used: ambient, directional, point, and spotlight. Figure 5 shows an example of a Catmull-Rom curve and three different types of lighting (as shown on the sphere). The left light is a point light, the bottom one is a directional light and the light at the top is a spot light. The meshes are shapes that are made up of many triangles, with a control point at each vertex. By moving the control points, the user can manipulate the appearance of the mesh. Patches can be changed in the same way, but can also be merged together using more than one patch to create an entirely new shape. From there they can be smoothed so that the connection is less apparent (see Figure 6 for an example of this).

The attribute editor is useful when precision is needed. By typing in coordinates for translation, rotation angles, or scaling amounts, the object can be manipulated directly, without the need for the mouse. The object selection combo box allows the user to pick an object on the scene, even if it cannot be seen. The currently selected object will always show up first in the list, with the remaining objects found below it. This object will be highlighted by way of mouse handles, which will appear on it once it has been selected. The camera itself can be rotated/translated/zoomed using the mouse, as well as keyframed and set on a path in the same way as an object. Pressing Shift and the left mouse button rotates the camera, Shift and the right mouse button translates it, and Shift with the middle mouse button zooms it. The user is also provided with a feature which turns on and off the lights, light handles (triangular shaped objects which move the lights around in the scene), and control points. This feature is found in the menu system under ‘View’ and uses checkboxes to determine whether one of the above mentioned is on or off.

This completes the list of the major features that are found in the system. The following four subsections will discuss in detail how object handles, object picking, placing textures onto

objects, and particle systems work. In each case, the main concept, the crucial aspects of the implementation, and any problems discovered are described.

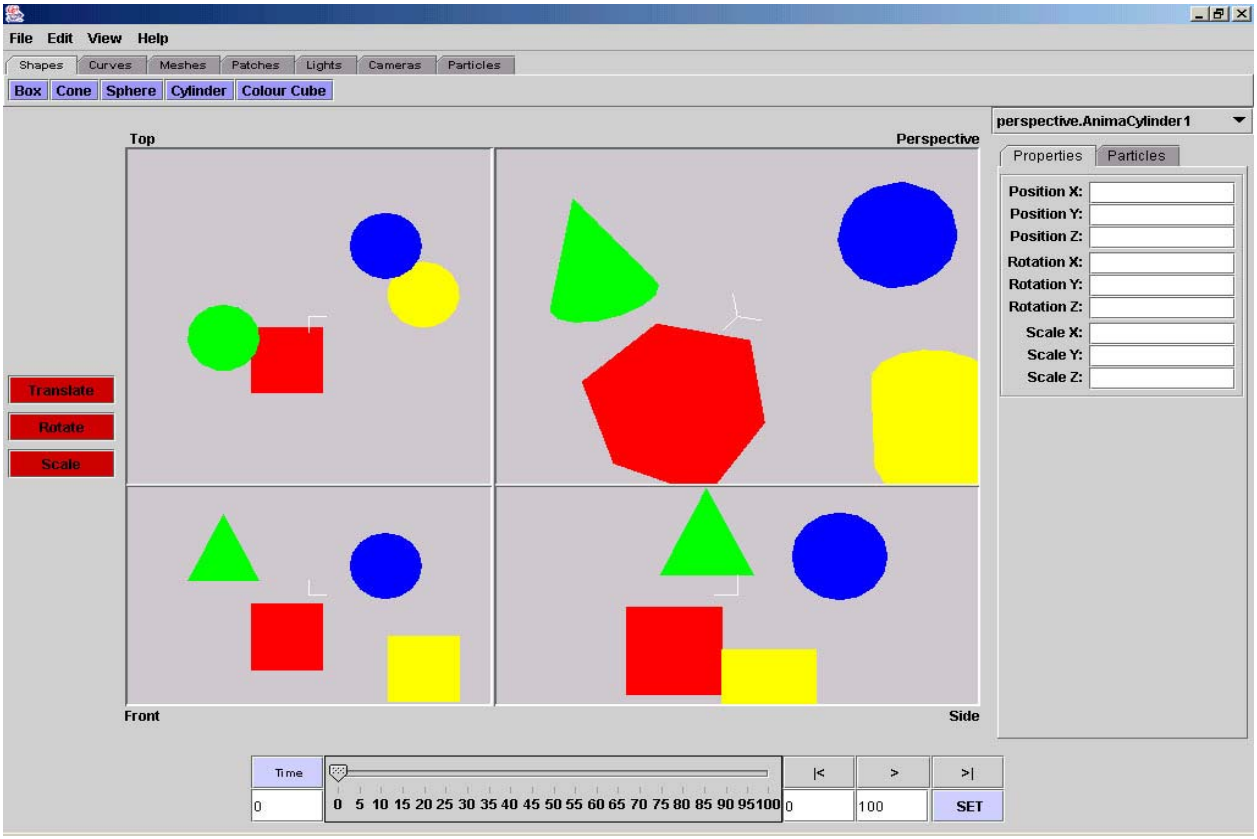

Figure 4: Overall Display of System

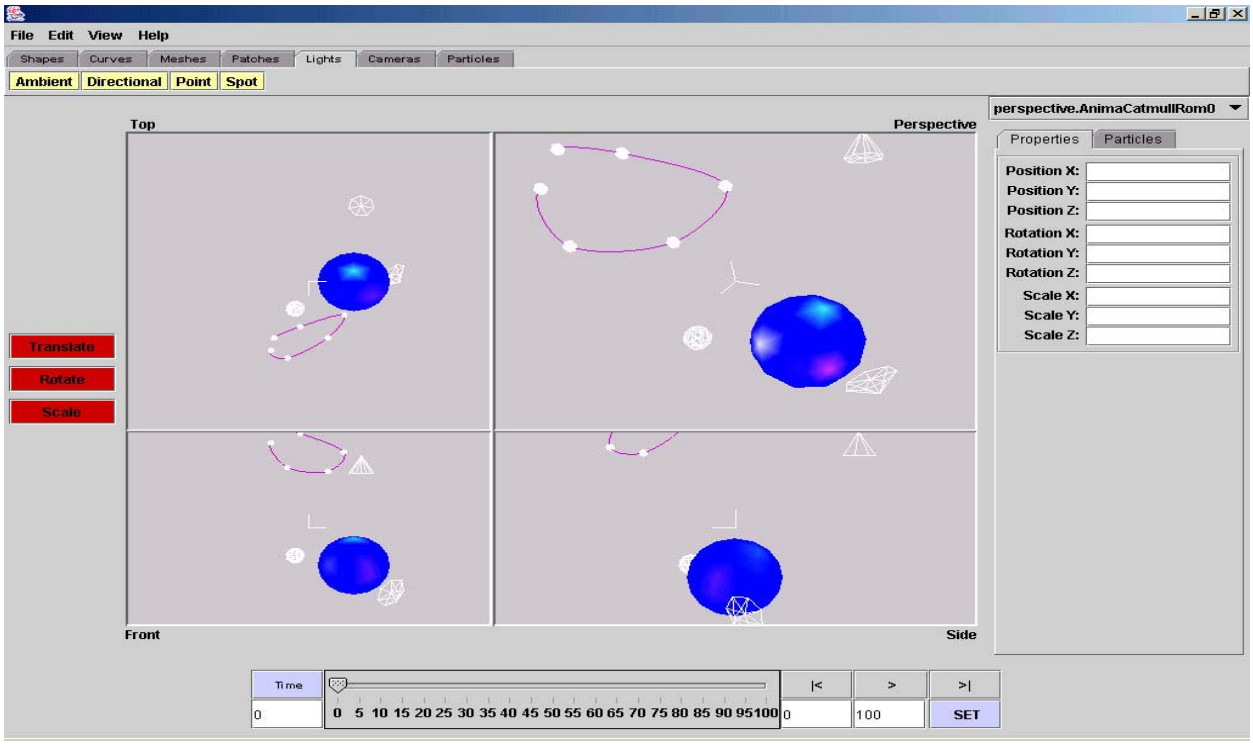

Figure 5: Curves and Lighting

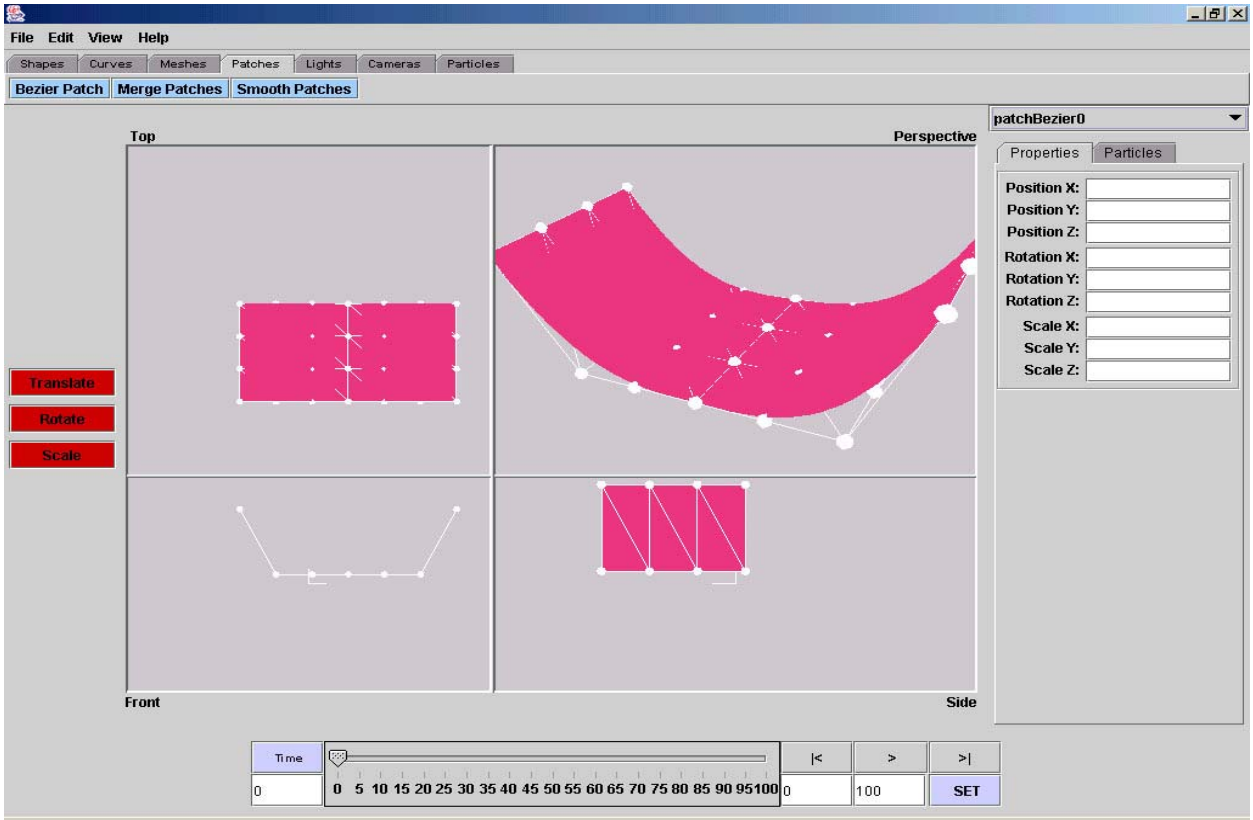

Figure 6: Merging and Smoothing Patches

3.4 Object Picking

Picking objects on the screen was one of the most time consuming aspects of the project. The first solution that was attempted involved Java's built in PickBehaviour functions. These functions allowed the user to interactively pick and manipulate visual objects in the scene. By using the left mouse button for rotating an object, the right button for translating an object, and the scroll or middle button for scaling an object, it appeared to be an easy and efficient way to pick and manipulate objects. However, once we got into more detailed testing of these functions, we discovered that they were not as efficient as we needed for the system. After placing one object so that it overlapped another, this method of picking chose both objects, instead of the one closest to the user. The problem with these functions lay in the precision of the picking. They used "bounds picking" to determine which object had been chosen. Bounds picking chooses an object by checking the area of the object's bounds, and not the area of the actual object. An object's bounds is usually larger than the size of the object itself, and covers an area around the object. When using bounds picking, the precision of the picking decreases because when the bounds of the object is encountered it is considered a hit. Thus, the accuracy of the selection is not always high because the user may select a point just off the object, but it will still be considered a hit if the bounds encompasses that area.

To combat this precision problem, a new approach was taken. It involved building a Java 3D PickRay and determining what, if anything, the pickray intersected with. This was done in two passes:

1. Select the Shape3D nodes that may be picked by seeing if their bounds intersect the ray. This can be done by using Java 3D's pickAllSorted() function, which returns an array of the objects that were hit by the pickray. The array is sorted from the closest object to the furthest one. This pass uses the type of picking discussed above by creating a list of objects that the pickray intersects with according to their bounds.
2. Determine if the geometry of the shape intersects with the ray. This pass is needed for more precise picking. We used the intersect() function of the Shape3D class, with the shape to be checked, the ray, and an empty distance variable (whose value is returned from the function) as parameters. The purpose of this function is to check if the given ray has intersected any of the geometries of the shape itself. If so, it returns a boolean value of true, and fills in the distance parameter with the distance the object is from the ray (along the z-axis). If not, it returns a boolean value of false and we know that the object bounds were picked and not the object itself.

This method of object picking was found to be much more accurate than the bounds picking, and passed the test of placing one object in front of another. It correctly chose the object that was closer in the scene to the user. It also recognized that nothing was being chosen when the user picked just outside of an object. With this method, the pickAllSorted() function does not find the entire shape if the shape is of Java type Primitive. The function finds the leaf node of the path in which the object is placed, which in this special case will be the object's face and not the object itself (because a Shape3D, or the face of the object, falls below the Primitive object in the tree). It is then up to the programmer to determine which shape this face belongs to. This can easily be done by determining the number of nodes in this particular path of the tree and then retrieving the node found one level above the leaf node.

PickBehaviour does all of the picking and is instantiated in the Universe class. The PickBehaviour class extends Java's PickMouseBehavior class, and therefore needs to pass into its superclass a Canvas3D, the main BranchGroup that contains all the objects in the scene, and the bounds it should be looking in for picking. An instance of the Universe class is also passed into PickBehaviour, but this does not go to the superclass. This instance is used to retrieve the active canvas in the scene, which is the canvas that the mouse is currently in. This is necessary in order to determine which of the four views the user is trying to pick an object in, as the orthogonal views (top, front, side) are handled differently than the perspective view when determining the attributes of the pickray. The mouse function called mouseEntered(), found in the PaintScene class, sets the active canvas variable to the canvas that the mouse just entered, which is then used in the picking class.

Figure 7 displays the system when an object is first placed onto the screen. The cube is "picked", which means that the current set of handles (translation handles) is displayed. Its attribute values are shown in the Attribute Editor and its name is listed first in the combo box found just above the Attribute Editor. This same process happens each time a new object is chosen, with the exception of when multiple objects are chosen. In this case, the values in the Attribute Editor and the first name in the combo box are those of the object that was selected last.

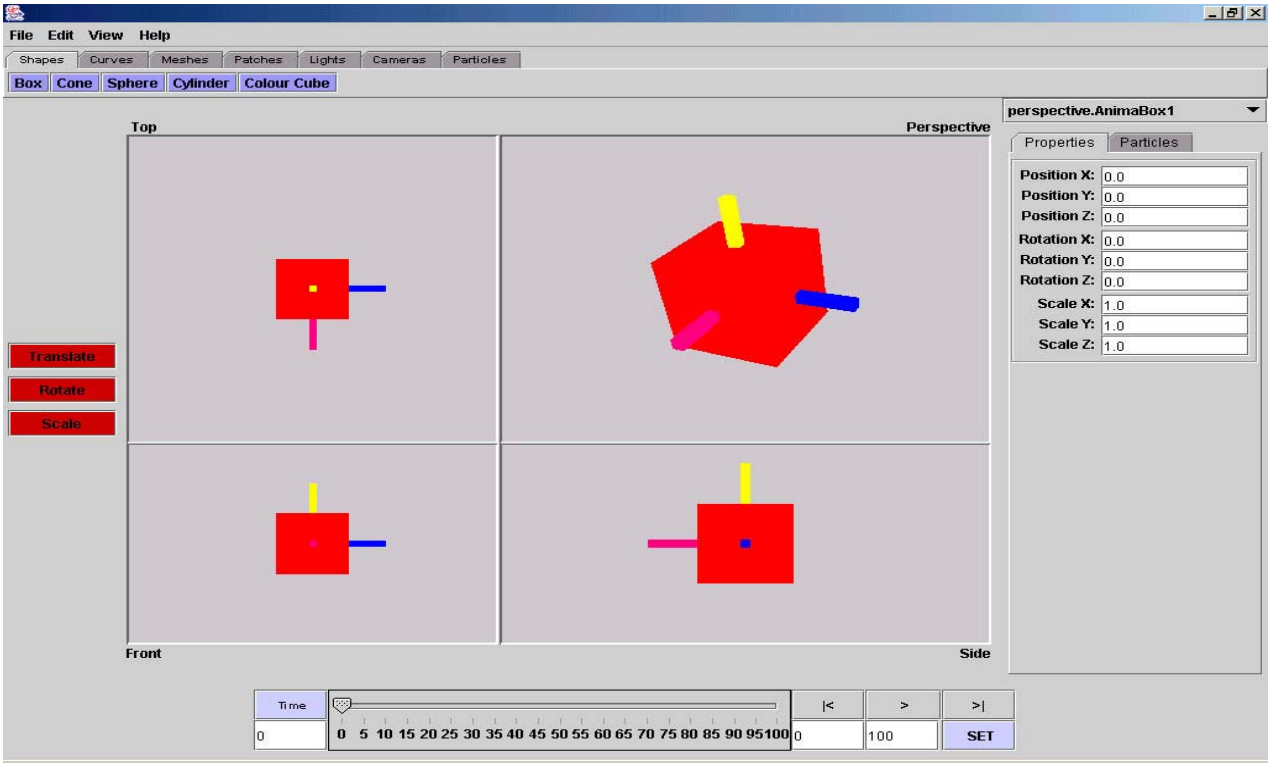

Figure 7: Object Picking

3.5 Textures

To place a texture on an object, the texture image and the capability bits must first be specified. According to Java standards, the size of the texture must be a power of two in each dimension. An error will occur at runtime if this rule is not satisfied and the texture will not be placed on the object. The following capability bits must be set in the appearance of the object:

```
app.setCapability(Appearance.ALLOW_TEXTURE_WRITE);
app.setCapability(Appearance.ALLOW_TEXTURE_READ);
```

These bits ensure that the texture of the object can be both read from and written to the object's appearance. Figure 8 below gives three examples of textures being placed on objects in the system.

There are four major steps in placing a texture on an object.

1. Prepare texture images (ensuring the size of the texture follows the Java standards).

2a. Load the texture.

```
TextureLoader loader = new TextureLoader("stripe.gif", this);
ImageComponent2D image = loader.getImage();
```

The libraries 'com.sun.j3d.utils.image.TextureLoader' and 'java.awt.image' must be imported for the TextureLoader and ImageComponent2D classes to be recognized.

2b. Set the texture in the Appearance bundle (by retrieving the object's appearance and setting the texture of it).

```
Texture2D texture = new Texture2D();
texture.setImage(0, image);
Appearance appearance = object.getAppearance();
appearance.setTexture(texture);
```

3. Specify the texture coordinates of the object's geometry. Luckily, Java provides us an easy way to do this. When creating the object for the first time, a flag that reads Primitive.GENERATE_TEXTURE_COORDS must be added into the constructor parameters. An example is the following:

```
Sphere sphere = new Sphere(1.0f, Primitive.GENERATE_TEXTURE_COORDS, app);
```

where the first parameter is the radius of the sphere, the second parameter is the flag which tells Java to generate texture coordinates for the object, and the third parameter is the appearance that will be set to the object.

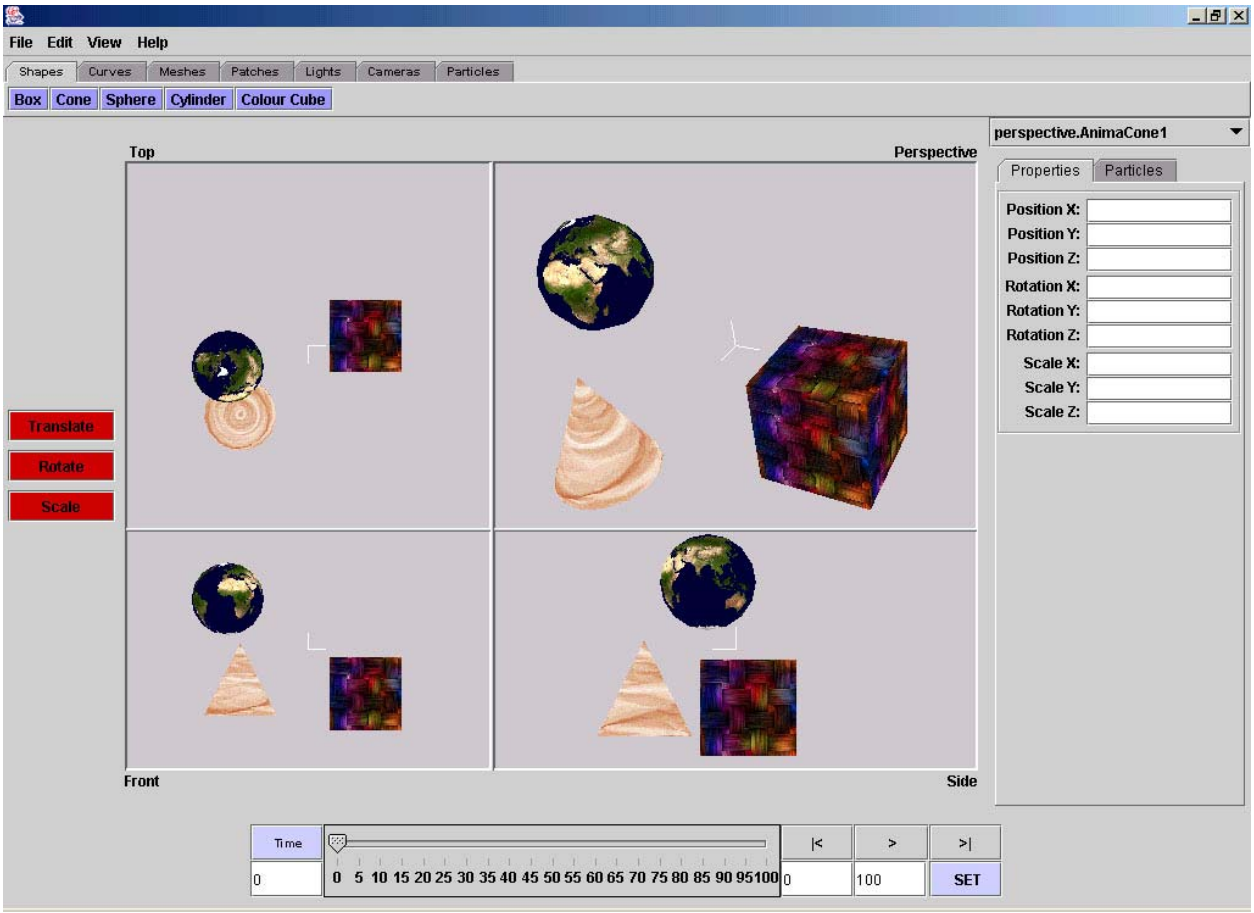

Figure 8: Textures

3.6 Object Manipulation

Determining the best and most efficient way to manipulate objects required experimentation. This task could have been done in three ways. The first used the PickMouseBehavior classes of Java 3D, including PickRotateBehavior, PickTranslateBehavior, and PickZoomBehavior. These classes allowed for easy manipulation of the object because they internally determined which object was chosen and then, based on the mouse button clicked, either rotated, translated, or scaled the object. There was a problem with this because the object picking was bounds based and not geometry based, and there was no precision with this type of picking.

The design of the object manipulation was changed to give the programmers more control. Within the mouseDragged function of the PaintScene class checks were placed to determine which button was selected, and then the object was rotated, translated, or scaled. The majority of the object manipulation code was placed into this class, including code for moving the camera. Although this code appeared to work, it too had difficulties. Since none of the three manipulations were restricted, the user could not rotate/move/scale in one direction without fear of changing the other directions. For example, a user could move an object with freedom

anywhere on the screen, but it was very difficult to only move in the y-direction without accidentally moving in the other directions as well.

The current version of Anima uses manipulation handles that place restrictions on movement. As in Maya or 3d Studio Max, a set of handles is shown on an object once it has been chosen. After a specific handle has been picked, movement of the object is restricted to the direction the handle points in. For example, if the translation handle pointing in the direction of the x-axis is chosen, the object can only be moved in a horizontal direction. If the rotation handle pointing along the y-axis is chosen, the object can only be rotated upwards (so it goes around the x-axis). Similarly, if the scaling handle pointing in the direction of the z-axis is chosen, then the object can only be scaled along the z-axis. Handles allow for easier and more controlled manipulation by the user. However, if the user chooses the object itself and not a handle, then the object can be moved in any direction with all capable freedom. We experimented with having a different mouse button activate each set of handles, but found it hard to use. Instead, only the left mouse button can be used, and pressing one of the three buttons on the left hand side of the screen changes the set of handles currently being used in the scene. Each button corresponds to the set of handles the user can apply at any given time: Rotate, Translate, and Scale. When picking for multiple objects was implemented, the design of the manipulation handles needed to be altered. The midpoint of all the objects was determined and a set of handles was placed at that point. Whatever was done to one object was done to all of the chosen objects.

The last major design change that took place was to the number of handle instances that could be created. At first, an instance of a set of handles was set up with each new object that was produced. Therefore, each object had its own set of handles, independent of all other object handles. This was an unnecessary use of space, so the number of handle instances was decreased to only one. No matter how many objects were created, only one handle instance ever existed. This meant that whenever a new object was picked, the position, size, and rotation of the handles had to be determined based on the new object, as these would be different for each object on the screen. Figures 9 and 10 display the set of rotation handles and the set of scaling handles respectively. The values in the attribute editor are altered to reflect the changes caused by the handles to the object.

Although the design and concepts behind object manipulation got increasingly more complex, the progression from the built-in functionality to the object handles was extremely beneficial in making the code behind this area stronger and more sound. The current design works efficiently and our experiments with other designs shows that its complexity is justified. It is also the most user-friendly of all the attempted designs.

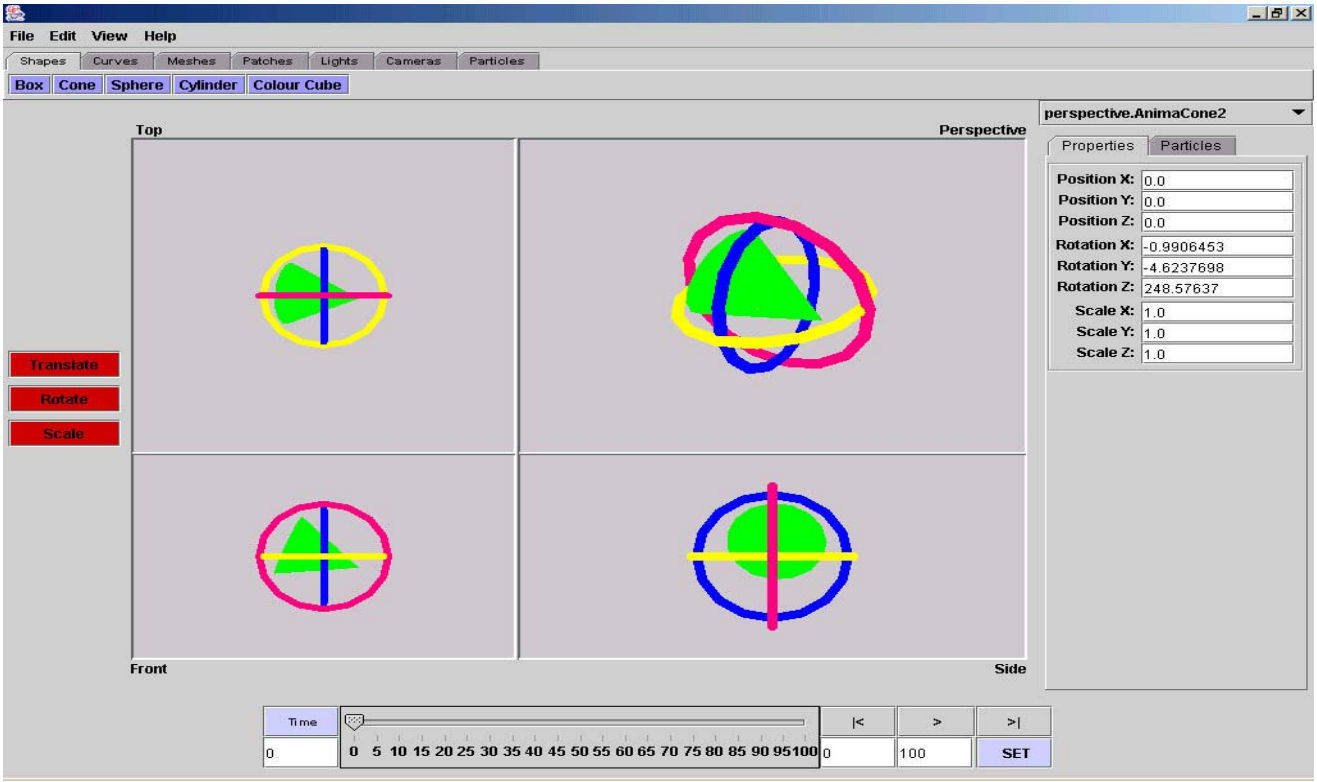

Figure 9: Object Rotation Handles

Figure 10: Object Scaling Handles

3.7 Particle Systems

The particle emitter is used as the starting point for the particles in a particle system. The particles travel out from the emitter and move around the screen. When the emitter object is moved by the user, its corresponding particles are moved along with it. The emitter is used in other features such as snow, fireworks, and the individual bursts. As is consistent with most particle systems [2], the particle emitter uses stochastic processes to initialize its member variables. These variables include colour, velocity, lifetime, and the number of particles generated per frame. The velocity, along with any forces working on the particle system, determines the position of each particle at a point in time. The shape of the particles in the system can be boxes, spheres, cylinders, and cones, and can be changed by the user. The colour of the particles can be consistent, or can be various shades of a user chosen colour. There is also the option for the particles to change colour at each frame, slowing moving from one end of the spectra to the other. Particles can also have textures generated on them. The mean velocity of each particle will be the same (as defined by the user in the Attribute Editor), but due to the stochastic processes, the end result will be slightly different from particle to particle. The same idea applies to lifetime and colour. For each frame, a stochastic process determines how many particles will be generated into the frame, with the higher the frame number, the more particles being generated in. Just like any of the other shapes in the system, the emitter can be keyframed to move around on the screen. All the attributes of the particles can be switched at any time during the animation (but not keyframed). This means that you can run the animation for a few frames using spheres, stop the animation, change the shape to cylinders, and then cylinders will start emitting, mixing with the spheres that have already been emitted in the system.

Collision detection was implemented into the system with the use of a function that is called when the user turns on collision detection for a specific particle system. The collision detection function then proceeds to send a pickray from the particle in the direction that it is currently moving. A pickray is a Java 3D class that selects objects along the path supplied. The particle's current position is the origin of this ray and the velocity of the particle is the direction it is moving. The pickray then returns a list of objects in the scene that the ray intersects. This list is sorted by distance with the first object being the closest object to the origin of the ray. These objects are the first group of potential collisions.

The collision detection function then filters out any objects that do not have their geometry intersect with the pickray. The first pass checks to see if the ray intersects with an object's bounds, then a second check is performed to see if the ray intersects with the actual geometry of the object. When an object is found that intersects geometries, the distance to this object is then computed. This distance is then compared to the distance that the particle will move in the frame. If the object is within this distance a collision has occurred, and the position and velocity of the particle must be updated. The position is set to the point on the edge of the object. The velocity is calculated using the kinematic method of rigid body simulation. The normal is calculated for the surface that the particle would hit. Then the normal, the particle's velocity, and a dampening factor called k are used to calculate the new velocity with the formula $v(t) = v(t_i) - (1 + k) \cdot v(t_i) \cdot N$. The new velocity is updated in the system and a true flag is returned to signal that a collision has occurred. If a collision has not occurred then a false flag is returned and the system updates the particle's position as normal.

Exploding objects are one of the most technically challenging of the particle features discussed here. They take an object in the scene, break it up into numerous different particles, and cause them to "explode". This is done by taking the given object and breaking it up into its respective faces. From there, Java 3D defines the 'geometry' of each face, which consists of numerous triangles that put together each side of the object. This is where the particles are created. The coordinates are divided into sets of 3's, and each set becomes a particle in the particle system. When the sets are all put together, they look exactly like the original shape. However, the original shape actually has been removed from the system, leaving in its place many particles put together. To cause the appearance of the explosion, the midpoint is calculated for the three points in each particle. By comparing this midpoint to the origin of the object, it is determined which direction is 'away' from the center. This is the direction the particle will travel in, giving the object the appearance of exploding.

In more technical terms, this feature was accomplished by looping through each side (face) of an object and retrieving its coordinates. From there, the coordinates were put into a linked list so they could be easily removed. Depending on the type of shape being used, either a TriangleStripArray (for spheres, boxes, and cylinders) or a TriangleFanArray (for cones) was used to divide up the coordinates and place them in particle systems. These types are defined by Java 3D and are used to put together the geometry pieces of an object. If a TriangleStripArray was going to be used, then the coordinates were put into sets of 3's in the following way: 012, 123, 234, etc. If a TriangleFanArray was going to be used, then the coordinates were put together as follows: 012, 023, 034, etc. Using a special instance of the particle class that takes a GeometryArray (which is the class that TriangleStripArray and TriangleFanArray inherit from) as a parameter, these arrays were passed into the class and a particle was created from them.

Snow is created by particles emitting along a plane horizontally and then falling vertically. The snow is created at a certain height above the emitter and spreads out along the x-plane. The snow then begins to move down in the y-plane, creating the effect of falling snowflakes. Snow was designed in the system by extending the AnimaParticleSystem class to provide more precise functions and methods.

A particle burst is an emitter that produces all of its particles at one frame and emits them at the same time. This creates an explosion of particles that travel in all directions. Particle bursts were implemented in the system by creating special functions in the AnimaParticleSystem class that behave differently than the original functions. These functions deal mainly with the process of creating the particles, as this happens at only one frame instead of during all the frames. The process of moving the particles around the scene is the same as a normal particle system and uses the same functions.

A firework particle system consists of a special firework particle emitter and a series of burst emitters. The firework particle emitter is used to fire a single "trail" particle that travels from the firework emitter to the burst emitters. The burst emitters then burst sequentially every few frames in random colors. Fireworks are created by selecting the firework button and choosing the number of bursts required from a dialog box. The user then clicks two points on the screen where the emitters will be positioned. The firework emitter is placed at the first click and the burst emitters are placed at random locations around the second click. The fireworks

system is implemented in its own class and inherits most of its functions from the AnimaParticleSystem class. There are a number of additional attributes, such as a linked list that stores the burst emitters for reference by the firework emitter. Special functions are used to calculate the velocity of the trail particle and update that particle's position every frame.

One of the other particle features was the implementation of gravity as a force that acts upon the particles in a particle system. A particle system is selected and then the gravity button pressed to apply gravity to that particular system. Gravity was implemented in the system by creating an acceleration vector that is used in conjunction with the velocity, the initial position of the particle, and the time that the force has been applied. The acceleration is set to a constant value that produces the desired effect for gravity. The velocity is the value specified by the user. The initial position is the point where the gravity is first applied to the particle. The time variable is the length of time the force has been applied to that particle (measured in frames). These values are then used in a formula, $P = ½(F * t^2) + (V*t) + P_0$, that calculates the particles position at the current time frame.

The attraction force pulls particles towards a point on the screen. A special attraction flag is used to alert the system that attraction should be applied to the particles when the attraction button is selected from the interface. This force pulls particles towards a point specified by the user. Attraction is implemented much like gravity. The force that attracts the particle towards a point is determined by formulas and is dependent on the distance between the two points. This force is then used to calculate the position of the particle at a given time with respect to velocity, the initial position the particle was at when the attraction button was selected, and the length of time the force has been applied. The formula used to calculate the particle's position is $P = ½(F * t^2) + (V*t) + P_0$. The path of the particles changes with respect to where the force point is; if the point moves then the path of the particles alters to match that. The particles resume their original movement pattern if the force is cancelled by clicking the attract button a second time.

Repulsion was put into the system as an opposite of attraction. It is implemented very similarly to attraction, with a small difference in the way the force is calculated for the particle. Instead of making the particles move in a direction towards the force point, repulsion makes them move in the opposite direction. If the force is cancelled then the particles will resume their normal movement pattern. Instead of creating a new class as some of the other features do, both the attraction and repulsion forces are added to the original AnimaParticleSystem class, with special functions and flags that alert the system to handle them.

The drag force has the effect of slowing down the particle's speed and decreasing the amount it moves each frame. It is accomplished in a slightly different fashion from the other forces. It is applied to a particle by subtracting small amounts from the particle's velocity. When drag is cancelled, the small amounts are added back to the velocity until it reaches the original rate, which causes the particles to speed back up. Drag is created by following the same steps required to apply other forces to a particle system, such as selecting the corresponding emitter and then pressing the drag button. To cancel the drag, simple press the drag button a second time.

4. Conclusion

Anima is a software package that allows the user to create and animate scenes that can be rendered into an AVI movie. It gives them the freedom to shape objects, add visual effects such as lighting, keyframe object movement, and change an object's parameters and attributes. It is part of the AniMiner project for research into discovery-aided Computer Animation. Anima is flexible, user-friendly, and easily sustainable. It has laid the groundwork for future research in this area. Some significant features of Anima are its ability to do precision picking, allowing the user to define their own control points for curves, the manipulation of objects using handles, merging and smoothing surfaces, path following for both objects and the camera, particle systems, and dynamic animation. Although Java was not the best language to use with respect to the interface, it provided functionality that made it much easier to implement some features that would otherwise have been quite difficult to do.

The following changes could be made to the Anima software to increase its functionality and to improve its usability.

1. The addition of a message bar that gives instructions on the feature being used.
2. The modification of object handles so that they stay the same size, no matter the size of the object (they appear to be inside the object).
3. The ability to remove an object from a group of currently selected objects.
4. The addition of automatic removal at render time of all objects the user wouldn't want in the rendered animation (e.g.: axis, light handles).
5. The addition of icons in the tab menu to denote the features and conserve space.
6. The ability to save and load the animation the user is working on.
7. The improvement of the AVI file saving to avoid losing frames.
8. The ability to convert shape objects to meshes so as to deform them.
9. The ability to add/remove points for curves after they have been created.
10. The ability to change the time settings for an object following a curve.
11. The ability to detach objects from a path.
12. The addition of an undo option.
13. The ability to have objects move backwards around a curve.
14. The ability to view and change the properties of a light object.
15. The ability to do object tracking when moving a camera on a curve (currently the camera does not always point to where the object is).
16. The ability to make animation movies from any view (currently this only works for the perspective view).
17. The addition of a camera object that can be manipulated, instead of using the mouse and Shift button.
18. The ability to change if the handles are relative to an object or absolute according to the world.
19. The improvement of the timeline so the number of frames can be increased or decreased by the user.

References

[1] Palmer, I. *Essential Java 3D Fast.* Germany: Springer Verlag Pub., 2001.

[2] Reeves, W., Particle Systems – A Technique for Modeling Fuzzy Objects, Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).

[3] Sun Microsystems Java 3D Engineering Team. *Java 3D API Tutorial.* Sun Microsystems. September 8, 2003 <http://developer.java.sun.com/developer/onlineTraining/java3d/>.

[4] Wahl, A., Battle of the 3-D Killer Apps, *Canadian Business*, pp. 41-44, February, 2003.