“GLHangings”: A Tool for Creating Hierarchical 3-D Mobile Animations in OpenGL

Abhishek Venkatesh (venky@gatech.edu)
Jason Sirichoke (jason.sirichoke@gatech.edu)
Mayur Bhosle (mbhosle3@mail.gatech.edu)

CS 6491, Fall 2007, College of Computing, Georgia Institute of Technology, Atlanta GA

Abstract

In this paper we describe the details of our tool “GLHangings” which can be used to quickly create hierarchical mobiles animations which we call as hangings. The user can create a tree structure consisting of a mobile at each node with a few mouse clicks and hang 3D Objects (triangle meshes) under the leaf nodes in the tree structure. Each mobile rotates around the y-axis in clockwise or anti-clockwise direction with different speed which the user can vary. The user can choose the animation type (Minkowski Morphing and Twister) for each 3-D Object from. Graphical User interface is provided to user to interact (camera, light, wireframe view, mouse picking) and modify the mobile structure. Finally the user has the option to save the scene he has created or load a previously saved scene. Our tool is capable of creating multiple complex mobile hangings including very large triangle meshes with textures.

Keywords

Animation, Minkowski, Morphing, Twister, Opengl, Glut, picking, triangle mesh, corner table, center of mass.

Problem statement

The aim of the project is to create a mobile design studio for animation. Each mobile can have multiple legs below which more child mobiles could be hanging forming a hierarchy of mobiles. Each mobile rotates about its local y-axis in clockwise or anti-clockwise direction along with its children hanging below the respective leg of the mobile. The leaf mobiles may have 3D-shapes (triangle meshes) hanging below them. The wire with which the 3D-shapes are hung should be attached to the center of mass of the 3D-shape from the parent mobile leg to show realistic hanging. Each of these solids changes its shape over time using morphing or twisting. The Studio should provide graphical user interface to interact with the animation and should allow the user to save his creations and load them back.

Introduction

“GLHangings” is a tool based on Opengl which provides the user a mobile design studio to create animations consisting of mobile tree structures with 3D-shapes hanging below the leaf mobiles.

Section I describes the components of the hierarchical mobile animation and the representation of the hierarchy, Section II describes the animation of the 3-D solid shapes, Section III describes the file handling, Section IV describes the user interface menus, Section V shows some sample screenshot of the animation created using this tool and Section VI gives the conclusion and future work.

Section I

1.1 Components of the Hierarchical Mobile Animation

Each hanging animation consists of the following components (See figure 1 for visualization)

1) **Mobile**: Mobiles are the building blocks of the hierarchical animation. Each mobile consists of a snake (consists of 5 balls and skin) to hang it and a number (>=0) of legs (again a set of balls and skin). Each mobile rotates about y-axis around the snake with user specified angular velocity and direction. Properties associated with each Mobile:
   a) Location in World Space: The coordinates of the mobile is computed from the tip of the leg of the parent mobile. An offset is added to the y-component to show that it is hanging from the parent mobile's leg with a string connecting the parent leg to the snake.
   b) The number of legs
   c) The center and radius of the balls of which the Snake and Legs are made of in local coordinates.
   d) Colour of the Mobile.
   e) Angular velocity about the vertical axis and the direction of rotation
   f) A Boolean variable "leaf" to identify that the current mobile is a leaf in the hierarchy or not. Based on this variable we choose traverse down the tree further or just display the 3D-shape hanging below it.
   g) Each Mobile has a set of flies which hover around it randomly.
   h) Parent mobile pointer.

2) **3D-Shapes**: The 3D shapes are triangle meshes which are loaded from files. The file contains the coordinates of the vertices of the mesh and a triangulation of these vertices which represents the 3D shape. Optionally it may have texture coordinates and an accompanying texture file usually in the form of a .bmp image. The 3d-shape's center of mass is computed and the point of hanging is directly above the center of mass of the solid. This gives a feeling of **realistic hanging(See Appendix A for algorithm)**.
   a) Filename of the mesh and its texture file.
   b) Corner table representing the triangle mesh.
   c) Texture Coordinates.
   d) Center of mass of the solid.(See Appendix to see how to compute this).
   e) Vertex-Normals.
   f) Face-Normals.
   g) Animation type(Minkowski or twister)

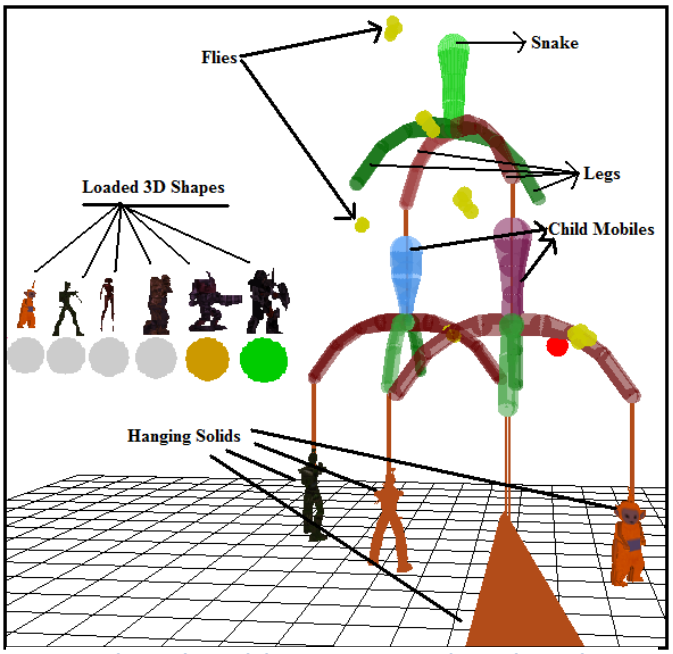

Figure 1 : Shows the mobile components and sample 3D shapes and a hanging structure with parent and child mobiles.

1.2 Representation of the Hierarchical Mobile Animation

The Animation is stored as a simple tree structure. Each node of the tree is represented by a mobile. The number of children at each node is denoted by the legs the mobile node has.

The tool is capable of creating multiple trees. To avoid keeping track of individual trees and to ease the loading and saving of the animations all trees have a common

root. This results in no loss of generality. Choosing a common root node for all trees has the following advantages:

- Avoids keeping track of individual trees which simplifies the data structure.
- The display routine just has to call a function to display the root. It provides an abstraction for rendering the complete animation.
- Simplifies the file format for saving and loading the complete animations consisting of numerous trees.
- There is no loss of generality.
- All trees can be translated together.

The underlying tree algorithm manages memory dynamically using pointers and updates the nodes based on the user interaction dynamically. The tree nodes can be altered by addition or deletion of mobiles and legs.

1) **Adding children ("Add Leg")**: If a user wishes to add more legs to a mobile he can simply choose the mobile and click on "Add Leg" button. This will add a leg to the selected mobile node. The user can then select the added leg and either a solid or hang further mobiles below it.
2) **Deleting a node ("Delete Leg"):** For deleting a node the user has to select the parent mobile's leg and delete it. The user can click "Add Leg" again to add the leg back to the mobile. If a particular leg is deleted the entire sub-tree is deleted and the memory collected back.
3) **Rendering the tree:** The tree is rendered in "depth first traversal". At each node the rotation and translation of that particular node is applied and the components of the mobiles are displayed. OpengGL provides a convenient way to push and pop the current transformation matrices. Each mobile renders itself in the following structure:

```
RenderMobile(){
glPushMatrix();// save the current state
  glTranslated();// apply the translation specific to the node.
  glRotated();// apply the rotation specific to the node.
  renderSnake();
  renderLegs();
  for each child
    RenderMobile() // recursive call
if(leaf node)
RenderSolidMesh();
glPopMatrix();// restore the saved state
```

At each leaf node we also store the type of animation to performed for the 3D-shape and render accordingly. Our tool is capable to complex meshes (3D shapes) and their morphing which consist of large number of triangles. We use glDrawElements to avoid rendering individual triangles but render them in one go. This reduces the number of calls and gives the complete triangle mesh in a single call to the rendering pipeline. We realized almost 5-6 time performance gain as compared to rendering individual triangles.

Section II

Animating the 3D-Shapes

Our tool provides two types of methods for animating the hanging solids: Vertical Twister and Minkowski Morphing.

2.1 Minkowski Morphing

3D morphing is the process of gradual transformation between 3D bodies. We refer to the 3D bodies which are represented as polyhedra, i.e. by vertices, straight edges connecting them, and faces, each consisting of a planar vertex set with the connecting edges. This is the most common representation for 3D bodies in computer graphics, and especially in modellers for 3D animation. For morphing we need to know the initial and final shapes i.e A and B.

Minkowski Sum: Minkowski sum of sets A and B is denoted by (A xor B) where

(A xor B) = {a +b | a E A, b E B }.

Important Properties of Minkowski sums:

1. The shape of the Minkowski sum of two polyhedra is independent of the choice of the coordinate systems and is not affected by any translation applied to either of argument polyhedra.
2. The final shape is dependent of the orientation of the two initial polyhedra
3. Given any two arbitrary polyhedra A and B, the faces (A xor B) are a subset of the union of faces that are written as (Va xor Fb) , (Vb xor Fa) or (Ea xor Eb) where Ea, Va, Fa are edges , vertices , faces of polyhedra A.
4. The morphing of non-convex shapes produces multiple branches and self-intersection.

Parameterized Interpolating Polyhedron (PIP)

The PIP interpolating the polyhedra A and B, denoted by PIP(A,B) is a graphical representation of the family of shapes, C(t) t belongs to (0,1) such that,

$$C(t) = (1-t)*A \text{ xor } t*B.$$

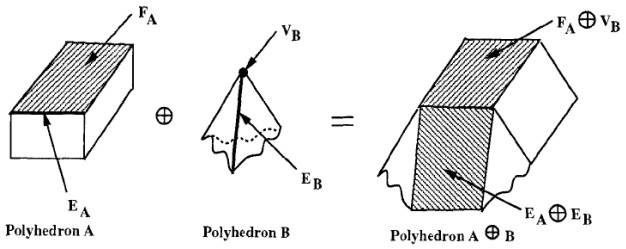

Figure 2: Shows the vertices and edges which will be mapped with each other for interpolation

Algorithm:

1. For each vertex in B(VB) find the matching face of A(FA). (Figure 3). The matching is done when each blue tangent has a negative dot product with the red normal. The set of blue normal can be approximated as the outward vertex normal. The red normal is simply the cross product of the 2 edges of the face.
2. Similarly for each vertex VA in A find the matching face FB in B.
3. For each edge EA in A and EB in B find the matching edges. The matching is done when each blue tangent has a negative dot product with the red normal.(Figure 4). Here the red normal is the cross product of EA and EB. The blue vectors are in face vectors for each face which EA and EB are part of.
4. Once we have the mappings for all the vertices using the above 3 steps, we build a triangle list, whose vertices are each represented by a reference to a vertex of A and a reference to a vertex of B
5. Since computing the vertices during animation would be compute intensive we pre-compute the interpolated values for t=0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1. Each vertex of M(t)=(1-t)A+tB linearly interpolates a vertex of A and a vertex of B where M(t) is the intermediate morphed shape.

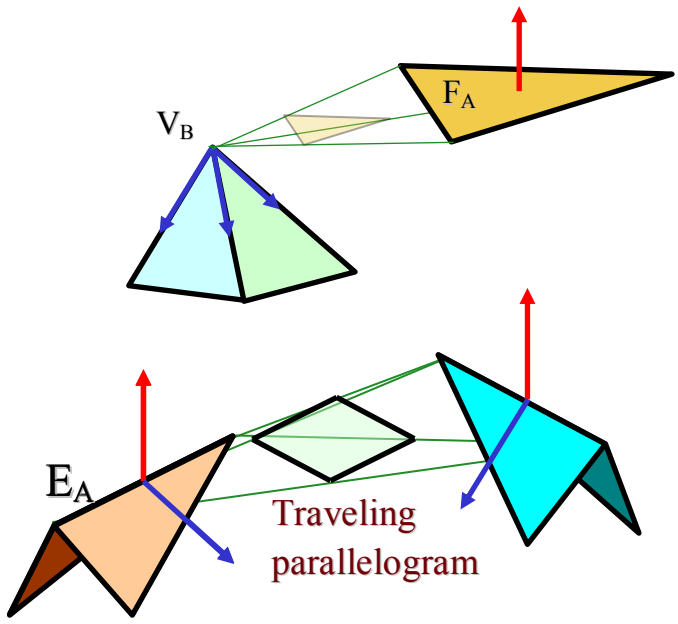

Figure 3: Vertex-Face mappings and Edge to Edge mapping

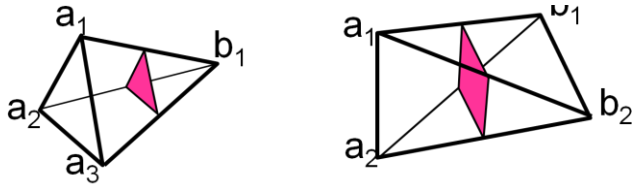

Figure 4: Intermediate Triangle for vertex-face mappings and edge-edge mappings.

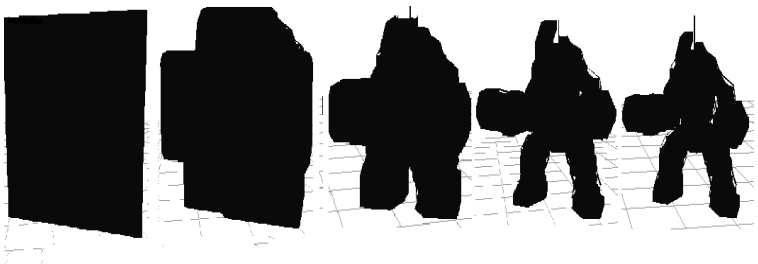

Figure 5: Morphing of cube to a robot using Minkowski Morphing

2.2 Twister

This is form of deformation where a warp is applied to the initial shape. The warp is applied progressively to show the animation. For each application of warp we generate a shape which is rendered on the screen.

The twister in our tool does deformation by rotating the vertices of the mesh around the vertical axis. The amount of rotation is interpolated from top to bottom which gives an appearance of twisting the shape:

Algorithm:

1) Compute the minimum and maximum y (miny,maxy) coordinate of the triangle mesh.

2) Each vertex is rotated around the vertical axis by 'R' degrees. But a scaling 's' is applied to this R based on the y-coordinate of the vertex. The scaling factor 's' is computed as follows:

`s=(G[i].y-miny)/(maxy-miny);` //where G[i].y is the y-coordinate of the vertex.

3) The rotation is applied as follows:

```
x = G[i].x* cosr - G[i].z *sinr;
z = G[i].z* cosr + G[i].x *sinr;
```

4) The rotation angle 'R' is also varied based on a time parameter and step 2) and 3) is done before rendering each frame.

Sample screenshot of this deformation is shown in figure 6. Note that the legs do not move because the rotation at the bottom most part of the mesh will be calculated as zero in step 2) for y=maxy

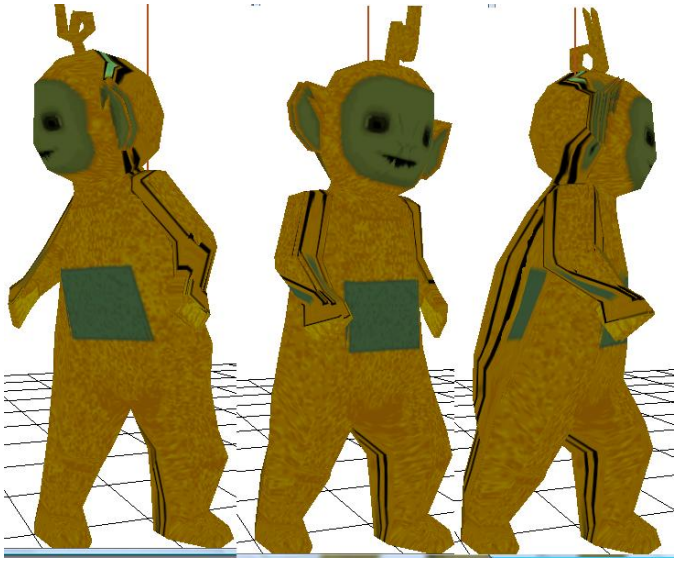

Figure 3: Deformation using Twister

Section III

File Format:

The "Save Scene" button gives the user the ability to save entire scenes in a proprietary file format and load them without being concerned about the underlying implementation. Each scene file is composed of the loaded mesh files, texture mappings, mobile configuration and scene layout. Since our tool is capable of creating complex hierarchies with complex meshes it will be redundant a waste space to save the meshes inside the animation file. It is possible to explicitly save the details of each loaded mesh and texture, but this is undesirable since it results in a very large monolithic file and the load process would have to read and process each data point individually. A single mesh can potentially contain on the order of thousands of data points and requiring an I/O operation to rebuild each data point exponentially increases the load time. Instead, the stored mesh and text files are passed to the initialization function of the mesh class which rebuilds the meshes from scratch directly using file information. This approach requires that the original mesh and texture files must be stored in addition to the scenario file. This also reduces the memory footprint while the tool is running. It takes less than a minute even for loading a scene with a dozen of meshes and different combinations of morphing.

Unlike meshes which only require the original filenames to be stored, the configuration and composition of the mobiles must be stored explicitly in order to be reloaded. This involves the following:

- Mobile
  - Visibility
  - Rotation Direction
  - The leg number of the parent mobile(-1 for root)
  - Location in World Space.
  - Rotation velocity.
  - Number of legs
- Mobile Base (Snake): We do not save any information for Snake as that can be re-

constructed from the Position of the Mobile stored earlier.

- Mobile Base (Snake)
  - global position
  - position, radius of individual internal balls
- Mobile Legs (Body)
  - For each leg if there is a child node.
  - Storing the information about leaf for each leg.
  - The center and radius of the balls of which the Legs are made of in local coordinates.
  - The chosen animation (Minkowski or twister) for the 3D-shape hanging below
  - The filenames of the meshes based on the type of animation.

It should be noted that color information is not stored in the scene file. Since colors are chosen at random and color manipulation is not a feature offered to the user, color choosing can be done at random when loading a scene. However, if necessary it would be trivial to add this feature in the future.

Mobile saving takes advantage of the natural tree properties of the implemented data structures by calling the save function of each node (mobile) in the tree (scene) then recursively calling the save function for each of its attached objects (children). The chain of function calls results in a depth-first traversal of the tree structure which is implicitly encoded in the saved scene file. Each save call returns when its respective mobile has been written to the file and as well as all of its children. Children of a mobile can be a combination of other mobiles or attached solid meshes. If no child exists for a particular leg in the mobile this can be represented by a null pointer in the code and saving is not invoked. A mobile is labelled as a leaf when it has only null pointers and/or solid meshes attached to its legs. When a leaf is reached in the chain of save function calls, the mobile and any of its attached meshes are written to the scenario file and control is immediately returned to the parent.

Save Implementation - Mobile

```
Mobile::Save(FILE f)

{
        f.write ( parameters listed above for each mobile)
        Snake.save();// save info for snake
        for(Leg C in Mobile.Legs)// save info for legs
                C.save(f);
        for(Leg C in Mobile.Legs)// save info for each child node
                if (C.isVisible AND C.hasChild)
                        C.child.save(f);// recursive call, results in depth-first traversal
}
```

The actual implementation for saving the base of the mobile does not explicitly write the base information described above. Location of the base snake can be derived from the global position information of the mobile and the radius of each ball can be chosen at random to duplicate the behavior of the initialization function.

Load Implementation – Mobile

```
Mobile::Load(FILE f)
        f.read(visibility, rotation, position, speed, parent)
        f.read(legcount)
        for (i:0 to legcount)
                Mobile.Legs[i].initialize();
                Mobile.Legs[i].child = null;
        f.read(children)
        for (i:0 to children)
                f.read(childindex)
                Mobile.Legs[childindex].child = new Mobile();
                Mobile.Legs[childindex].load(f)
```

The implementation for loading a mobile is focused on reconstructing the legs properly. Loading the base information is a simple copy of the data from the scene

file into the proper locations. The correct number of legs must be initialized and the child mobiles must be reattached to the proper legs. This call is recursive in nature since it can result in load calls for multiple mobile objects.

Load Implementation – Leg

```
Leg::Load(FILE f)
        f.read(bodysize)
        for(i:0 to bodysize)
                f.read (visibility, leaf, morphtype, position, radius)
        if(leaf)
                if morphtype != Minkowski
                        f.read(filename)
                        solidmesh.initialize(filename)
                else
                        f.read(filename1,filename2)
                        solidmesh.initialize(filename1, filename2)
```

If the leg contains a mesh, it is important to distinguish what kind of mesh it is in order to know how many filenames to pass to the mesh initialization function. Since meshes are actually preloaded prior to rebuilding the mobiles, it is possible to just pass a pointer to the loaded meshes rather than the filename. This is an implementation preference and the outcome is the same.

Implementation Notes

The use of initialization functions is important when saving and loading scenes in the mobile application. Without a proper initialization function, it would be necessary to store more the value for each variable in the mobile related classes then copy these variables individually when loading the object. Initialization functions provide the guarantee that they will initialize the object with the same values every time they are called. This allows for only minimal information to be stored that is necessary to invoke the respective initialization functions when loading the scene.

The file format can be visualized by means of figure 7

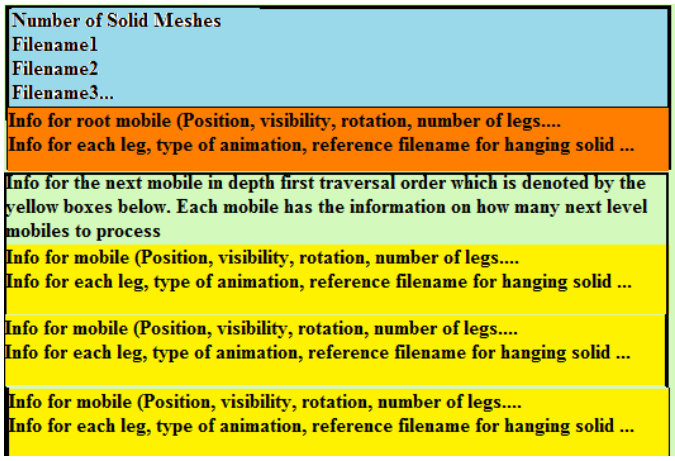

Figure 4: File Format for saving and loading the animation

Section IV

User Interface

The user can interact in the studio using the standard input devices: Keyboard and Mouse. A screenshot pf the UI is given in figure 8. An explanation of the buttons is given in figure 9. Also refer to figure 1 for few more information which can be inferred from the labelling in the figure.

Salient features of the UI:

- Ability to select objects in 3D using mouse clicks (Picking).See Appendix B to see how to implement picking in Opengl. Though this seems very trivial the situation becomes very complex when dealing with a hierarchy of models with each having its local coordinates and also having its own local transform.
- The components selected by the user (Snake, legs, solids) change colour when chosen by the user using the mouse.
- The gird lines are shown to make the directions and position of the viewer more clear.

- A generic camel model like a first person shooter game. The user can look using the mouse and move using the arrow keys. This allows the user to view the mobiles from any perspective.
- The loaded Meshes are displayed on the scene and the ability can chose any of them again by simply clicking using the mouse.
- Translation is allowed in all three (x,y,z)directions.
- Flies (small yellow spheres) are shown around the mobiles to make the scene livelier. These are small spheres which move around in random directions around a mobile. Some jitter is added to their movement to give a realistic effect.
- Colour Coded buttons to identify them quickly and become accustomed with.

Click This and move the camera like a first person shooter game. Change looking by mouse and translate using the arrow keys

Switch to wireframe mode. In this mode faces are not draw, only the edges are shown. Useful to see the Minkowski morphing actually working

Click this button and then click the Snake of the Mobile which you want to process

Select the leg of a previously selected Mobile(Snake). Should have selected a Mobile before clicking this

Add a leg to a previously selected Mobile(Snake)

Delete the selected Leg

Hang a new mobile below the selected leg(should have selected a leg before this operation

Hang the current chosen solid mesh under the selected leg. The animation type is set to twister mode

Hang the current chosen solid mesh under the selected leg. The animation type is set to Minkowski Morphing

To change the selected solid mesh to hang below a leg, click this and then click the sphere below the solid you want to use as current solid to hang below a leg

Load a triangle mesh(A dialog will be popped). Texture information is disregarded in this case

Load a triangle mesh with texture information(A dialogs will be popped). Another Dialog will be popped to get the texture bitmap file.

Save the current scene to file(A Dialog will be popped)

Load a previousle saved scene(A Dialog will be popped)

Flip the Direction of rotation of the selected leg. This will affect the mobile or solid which is hung below the selected leg.

Start rendering the animation. Use the key 'S' to come out of this mode and go back to edit mode.

Move the selected ball of the leg. Press 'x','y','z' to do translation along the corresponding axis. The x,y,z axis is shown in red, green and blue in the left top

Decrerase or Increase the velocity of the rotation of the selected mobile.

Disable the twister animation for the selected leg(will apply to the solid hung below the selected leg)

Figure 5: Explanation of user interface Buttons

Section V

Sample Result

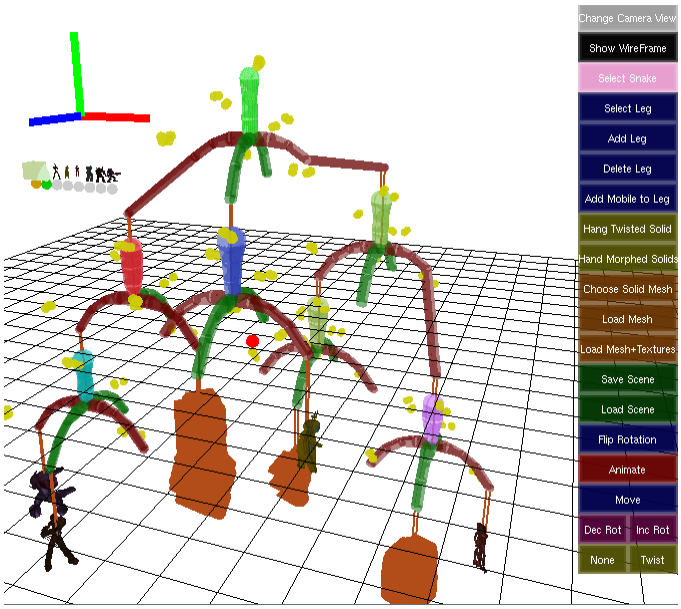

Figure 6: Sample Scene 1

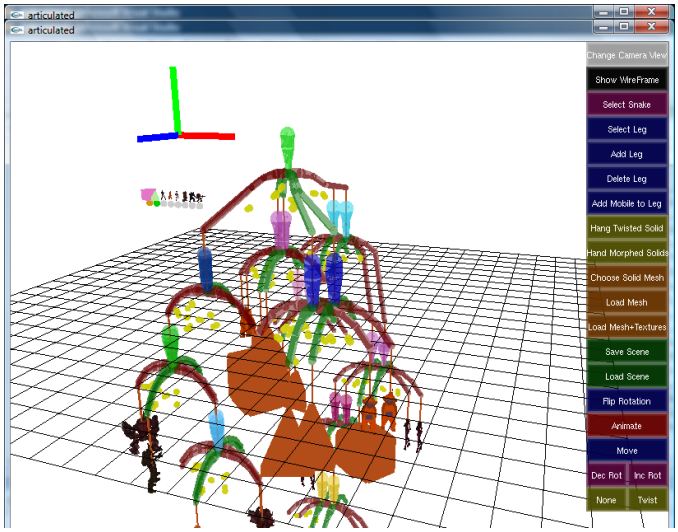

Figure 7: Sample Scene 2

Section VI

Conclusion:

We have provided a simple tool to the user to create complex hierarchical animations consisting of mobiles and triangle meshes. The solids can be animated using Minkowski morphing or twister. It takes only a few minutes to create a complex scene. The animation can be saved to a file and loaded back and improved upon. The tool is able to handle complex meshes and render them in real-time including the morphing which is compute-intensive. A generic camera model is incorporated to allow the user to view the model from any position.

Future Work:

- We can improve the rendering of the triangle meshes by using VBO (Vertex Buffer Object) in Opengl.
- Add shadow effects and better lighting effect for animation
- Add more types of animations for the solids
- A lot of backend parameters exist for which there is not User Interface. Example rotation about arbitrary axis, scaling the objects, copy paste sub-trees structures....
- Apply physics to compute various forces acting on an object and move it accordingly

References

[1] Rossignac, J. Retrieved from CS6491 Foundations of Computer Graphics, Modelling, and Animation: http://www.gvu.gatech.edu/~jarek/courses/6491/

[2] Jarek R. Rossignac. Solid and Physical Modelling.

[3]Kaul and Rossignac, Solid-interpolating deformations: Construction and Animation of PIPS.

[4]http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=32 (shows how to do picking in Opengl)

[5] Louis Bavoil, Rendering Huge Triangle Meshes with OpenGL

[6] OpenGL Programming Guide (Addison-Wesley

Publishing Company)

Extra Credit Features

We have implemented the following features for extra credits

1) Flies (small yellow spheres) are shown around the mobiles to make the scene livelier. These are small spheres which move around in random directions around a mobile. Some jitter is added to their movement to give a realistic effect.
2) A generic camera model similar to first person shooter. The user can look in any direction using the mouse and move using the arrow keys.
3) Texture mappings for solids which are hung from mobiles.
4) Implemented 2 kinds of animation for solids:
   a) Minkowski Morphing
   b) Twister
5) We are able to display multiple complex triangle meshes consisting of thousands of triangles at decent frame rate using specialized data structures and passing them to Opengl using glDrawElements.
6) Our tool can support any number of tree structures with multiple number of levels in each tree in one scene (limited by the memory of the computer). Thus we are able to create complex scenes with dozens of hangings and solids with animation.
7) Cool and easy ways to interact and edit with the animation. Examples include: wireframe mode, picking, translation along any axis by dragging the mouse, color coded buttons, grids to tell the user about the perspective view, view independent x,y,z axis is always show to the user for reference(see the red, green, blue axis in the UI screenshot).
8) Enable lights by right clicking the mouse.
9) Our mobiles can have as many legs as possible instead of the required 1,2,3,4. Thus it generalizes to a generic tree structure. Our file format is fully compatible with this feature.

Appendix A

Computing the center of mass for a triangle mesh

```
float meshvol=0;
centerofmass.setTo(0,0,0);
for(int i=0;i<3*nt;i+=3)
{
	CVec g =
	(G[V[i]]+G[V[i+1]]+G[V[i+2]]);
	g.scaled(1/4);
	float tetvol=(G[V[i+1]]-
	G[V[i]]).Cross(G[V[i+2]]-
	G[V[i]]).Dot(CVec(0,0,0)-G[V[i]]);
	centerofmass=centerofmass+g.scaled(
	tetvol);
	meshvol+=tetvol;
}
centerofmass.scale(1.0f/meshvol);
```

The idea is to compute the volume the tetrahedron composed of each triangle and origin. Compute the weighted position of the center of mass of each of the tetrahedron and divide the total by the total volume.

Appendix B

Picking in Opengl:

```
//pass the mouse coordinates and get the
corresponding 3D coordinate
CVec GetOGLPos(int x, int y) {
	GLint viewport[4];
	GLdouble modelview[16];
	GLdouble projection[16];
	GLfloat winX, winY, winZ;
	GLdouble posX, posY, posZ;

	glGetDoublev( GL_MODELVIEW_MATRIX,
modelview );
	glGetDoublev( GL_PROJECTION_MATRIX,
projection );
	glGetIntegerv( GL_VIEWPORT,
viewport );
	winX = (float)x;
	winY = (float)viewport[3] -
(float)y;
	glReadPixels( x, int(winY), 1, 1,
GL_DEPTH_COMPONENT, GL_FLOAT, &winZ );

	gluUnProject( winX, winY, winZ,
modelview, projection, viewport, &posX,
&posY, &posZ);

	return CVec(posX, posY, posZ);}
```