3D PENGUIN

Author: Jin Wu
October 31, 2008

INTRODUCTION

This report describes the implementation of a 3D penguin (**Figure 1)** in complement to the code provided.

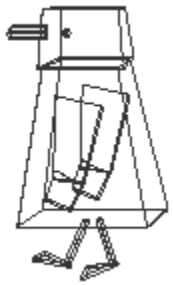

Figure 1: 3D Penguin

The penguin is a 12-part, 24 degree of freedom robot. There are nine joints that control **rotations** of different components, as well as two beaks that can **translate** up and down.

This report consists of 6 parts:

A. Modelling
B. Rendering Style
C. Material Properties
D. Light Source Control
E. Animation
F. Interpolation

A. MODELLING

To model the penguin, primitive convex 3D polygons were used. In particular, two functions were used to aid the creation of polygons, these are: `void drawCube()` and `void drawTrianglePrism()`.

Similar to the previous report, pushing and popping of matrices were done to ensure that transformations done to individual objects will not affect objects after it. On the other hand, hierarchy ensures that changes to the parental objects (i.e. movement) will affect children objects as well. This is demonstrated in the movement of the elbows, knees, as well as the beaks. This algorithm can be better understood in the form of a hierarchy tree as seen in **Figure 2** below.

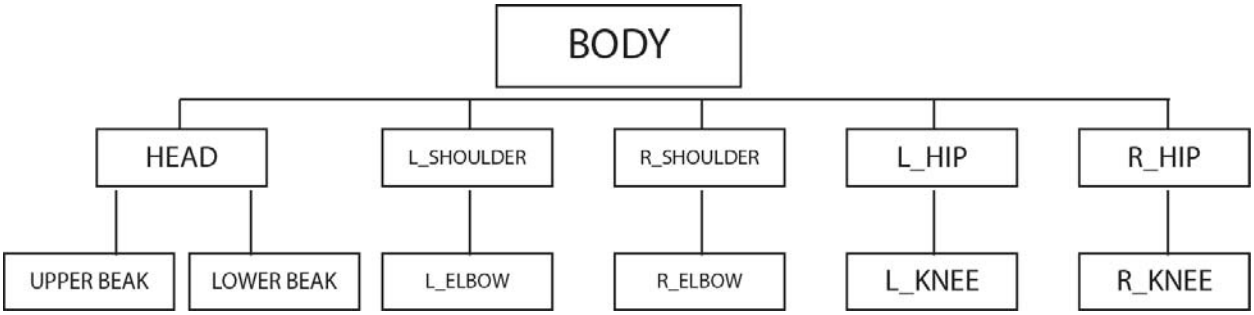

Figure 2: Kinematic Hierarchy Tree

Modelling is separated into three categories, with respect to the three different types of polygons.

I. The body is drawn in the following way:

1. **Draw** the 3D polygon
2. **Colour** it
3. **Scale** it
4. **Translate** and **Rotate** it along desired axis

II. Movable parts down the hierarchy (i.e. the head, shoulders, elbows, hips, and knees) are drawn in the following way:

1. **Draw** the 3D polygon
2. **Colour** it
3. **Translate** to centre location of object (i.e. define the location of the joint)
4. **Scale** the size of the polygon
5. **Set rotate** of the joint so it can rotate along the hinge, the head, elbows and knees will have 1 angle of rotation whereas the shoulders and hips will have 3 angles of rotation.
6. **Translate** the object to the joint hinge (i.e. define the local coordinates)

III. The beaks are drawn in the following way

1. **Draw** the beak
2. **Colour** it
3. **Translate** to centre location of object
4. **Scale** the size of the beak
5. **Set translation** of the joint so it can move up and down

All this has to be done in reverse order due to the nature of matrices.

For example:
The l_shoulder would be drawn in the following way
Push a new matrix
    Translate from body to joint hinge of shoulder
    Set rotation of shoulder around desired axis
    Push a new matrix
        Scale shoulder
        Translate to centre location of shoulder
        Colour shoulder a certain colour
        Draw the shoulder
    Pop the matrix
Pop the matrix // Note: this would only be done after the l_elbow has been drawn as well

This code can be found in the `void display(void)` function.

B. RENDERING STYLE

The first three rendering styles are as follows:

a. Wireframe
b. Solid
c. Outlined

All three are defined in the enum list in the initial variable list of the code. The renderStyle variable defines the selected render style when the program is first run.

The actual implementation of the three styles are in the `void display(void)` function while the primitive 3D polygons are being drawn.

The `glPolygonMode(FACE, MODE)` function is called to set the appropriate mode. FACE determines whether the function applies to front facing polygons, back facing, or both. MODE include GL_LINE and GL_FILL, which determines whether to draw WIREFRAME or SOLID, respectively.

Thus, the three modes can be defined as follows:

a. Wireframe:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

b. Solid

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

c. Outlined

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonOffset(10.0f,10.0f);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

In the outlined case, the polygon is drawn twice, once as a solid and once as a wireframe. The wireframe one is offset from the solid so that it can be visible.

A switch case is used to allow users to select between different modes using radio buttons.

The final code looks like this:

```
switch (renderStyle)
{
	case WIREFRAME:
		// Draw lines
		glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
		// Colour the lines black
		glColor3f(0.0f,0.0f,0.0f);
		drawCube();
		break;
	case SOLID:
		// Fill in solid
		glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
		// Colour the appropraite colour according to the colour vector above
		glColor3f(colour[0], colour[1], colour[2]);
		drawCube();
		break;
	case OUTLINED:
		// We need to draw the polygon TWICE, once as a wireframe, once as a time
		// as a solid
		glEnable(GL_POLYGON_OFFSET_FILL);
		// Draw the polygon as a wireframe
		glPolygonMode(GL_FRONT, GL_LINE);
		// Offset polygon
		glPolygonOffset(10.0f,10.0f);
		// Colour wireframe black
		glColor3f(0.0f,0.0f,0.0f);
		drawCube();
		glDisable(GL_POLYGON_OFFSET_FILL);
		// Draw the polygon as a solid
		glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
		// Colour it the appropriate colour of the penguin, according to the
		// colour vector above
```

```
                    glColor3f(colour[0], colour[1], colour[2]);
                    drawCube();
                    break;
}
```

Pseudo depth as well as backface culling were enabled so that objects that are backfacing as well as these that are blocked by other objects are not visible. Note: in the wireframe case, backface culling is enabled but pseudo depth is ignored.

C. MATERIAL PROPERTIES

In addition to WIREFRAME, SOLID, and OUTLINED, users are also able to select the following rendering styles:

d. METALLIC
e. MATTE

These are done by assigning `glMaterialfv(FACE, PROPERTY, VALUE)` to the 3D polygon after it has been drawn. Higher specular values were assigned to metallic objects to make it more shiny whereas higher ambient and diffusion values were assigned to the matte objects but very low specular values were given to give it an overall "dull" look.

Two movable lights were assigned to the scene, each with an identical ambient, diffuse, specular and position assigned to the light.

The final code looks like this:

```
// The following are MATTE and METALLIC parameters (diffusion, ambient, specular)
GLfloat matte_diff [4] = { 0.6f, 0.6f, 0.6f, 1.0f };
GLfloat matte_amb [4] = { 0.4f, 0.4f, 0.4f, 1.0f };
GLfloat matte_spec[4] = { 0.1f, 0.1f, 0.1f, 1.0f };
GLfloat metallic_diff [4] = { 0.4f, 0.4f, 0.4f, 1.0f };
GLfloat metallic_amb [4] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat metallic_spec[4] = { 0.9f, 0.9f, 0.9f, 1.0f };

// The following are light parameters (ambient, diffuse, specular, initial position)
GLfloat ambientLight[4] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat diffuseLight[4] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat specularLight[4] = { 0.4f, 0.4f, 0.4f, 1.0f };
GLfloat position0[4] = { 3.0f, 0.0f, 2.0f, 1.0f }; // initial position of light 0
GLfloat position1[4] = { -3.0f, 0.0f, -2.0f, 1.0f }; // initial position of light 1

// render according to render mode
switch (renderStyle)
{
          case METALLIC:
                    // Apply metallic materials
                    glMaterialfv(GL_FRONT, GL_DIFFUSE, metallic_diff);
                    glMaterialfv(GL_FRONT, GL_AMBIENT, metallic_amb);
                    glMaterialfv(GL_FRONT, GL_SPECULAR, metallic_spec);
                    // Render the polygon as a solid
                    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                    drawCube();
                    break;
          case MATTE:
                    // Apply matte materials
                    glMaterialfv(GL_FRONT, GL_DIFFUSE, matte_diff);
                    glMaterialfv(GL_FRONT, GL_AMBIENT, matte_amb);
                    glMaterialfv(GL_FRONT, GL_AMBIENT, matte_spec);
                    // render the polygon as a solid
                    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                    drawCube();
                    break;
}
```

D. LIGHT SOURCE CONTROL

As noted in section C, two lights were added to the scene - Light 0 and Light 1.

Two cubes were added to the scene at the exact same positions as the lights to help visualize the location of the lights. Both lights are able to move in circles from -pi to pi. These values are controlled by the user via the glui_spinner in a similar manner as the penguin's joints that will be described in section E.

E. ANIMATION

A set of sample keyframes are included in the **`sample_keyframes.txt`** file. Please rename the file to keyframes.txt in order to view it.

The animation is two parts:

I. Defining the position of the penguin

The position of the penguin is controlled by the user. The bahaviour is similar to Assignment 1b. Rotation and translation of the penguin (or parts of the penguin) are described in section A. These rotations and translations are controlled by the user through the `glui_spinner`, and are limited by MAX and MIN values specified in the beginning of the code.

II. Keyframing control.

The KEYFRAME CONTROL window controls the movement of penguin across keyframes. It consists of 6 functions:

1. Load Keyframes (`void loadKeyframeButton(int)`)
2. Load Keyframes From File (`void loadKeyframesFromFileButton(int)`)
3. Start/Stop Animation (`void animateButton(int)`)
4. Update Keyframe (`void updateKeyframeButton(int)`)
5. Save Keyframes to File (`void saveKeyframesToFileButton(int)`)
6. Render Frames to File (`void renderFramesToFileButton(int)`)

All six buttons are controlled by the six functions stated in the brackets above. Most of the code has already been written. However, some were added, in particular:

Written:

3. Update Keyframe (`void updateKeyframeButton(int)`)
   This function was implemented as follows:
   1. Obtain the current keyframe ID
   2. If the current key is bigger than the value of maxValidKeyframe, update the maxValidKeyframe value with the current key value
   3. Write the current penguin position into the keyframe array
   4. Update the user with a message stating the keyframe has been updated.
   The code is as follows:

```
void updateKeyframeButton(int)
{
          // Get the keyframe ID from the UI
          int keyframeID = joint_ui_data->getID();

          // Update the 'maxValidKeyframe' index variable
          // (it will be needed when doing the interpolation)
          if (keyframeID>maxValidKeyframe)
                    maxValidKeyframe = keyframeID; // otherwise leave it as it is

          // Update the appropriate entry in the 'keyframes' array
          // with the 'joint_ui_data' data
          keyframes[keyframeID] = *joint_ui_data;

          // Let the user know the values have been updated
          sprintf(msg, "Status: Keyframe %d updated successfully", keyframeID);
          status->set_text(msg);
}
```

F. ANIMATION

Catmull-Rom interpolation was used for this animation. This interpolation is a derivation of the cardinal spline that yields a smoother interpolation compared to linear interpolation.

The interpolation is written in the `Vector getInterpolatedJointDOFS(float time)` function. The `joint_ui_data` stores the interpolated joint DOF vector, which are called upon during the animation and rendering.

For example:

```
joint_ui_data->setDOFVector( getInterpolatedJointDOFS(frameNumber *
DUMP_SEC_PER_FRAME) );
```

This sets the interpolated joint DOF at a specific time. The time is obtained by multiplying the frame number with seconds per Frame to obtain the precise second value.

CONCLUSION

In conclusion, the above explanations are sufficient add-ons to the given code for modeling and animating the 3D penguin.