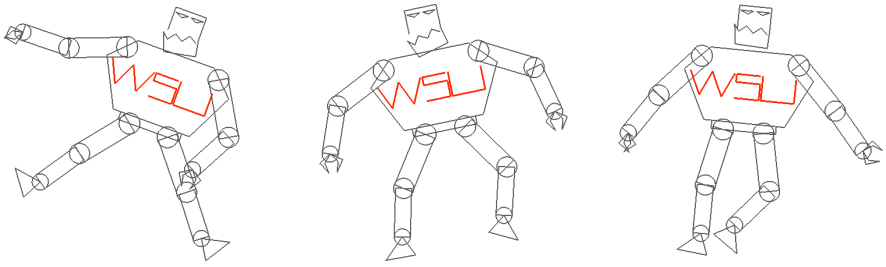
Programming Project #1
Dancing Robot

CS 442/542

Due Friday, September 14, 2012 by Midnight



1 Overview

This project is designed to give you some experience with *hierarchical modeling* and *object instancing* in OpenGL. You will design a 2D "robot" (or 3D if you dare) that consists of several moving components connected by joints. Each joint rotation should be parameterized so that the robot can be placed into different poses. You will then animate your robot by continually updating these rotation angles and re-rendering the robot in its new pose.

2 The Robot Model

The first step is to design your robot as a hierarchy of connected components as described in class. You robot should consist of at least the following components and joints:

- torso,
- head,
- left and right arms with shoulder, elbow, and wrist joints, and
- left and right legs with hip, knee, and ankle joints.

Feel free to add other components and joints (jaws, necks, fingers, etc. . .). Design each component as you like, but be creative.

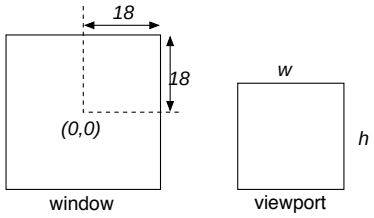
I would start with a "paper design" of your model (graph paper is useful for this) and draw the associated *scene graph* containing the appropriate transformation and primitive references as shown in class. When designing each individual component use a local coordinate system with the origin at the appropriate "pivot points." Use your design to create the appropriate OpenGL source code that traverses the scene graph to render your robot.

3 Displaying Your Robot

I suggest implementing your program incrementally. Before animating your robot, simply render it in some static pose (choose some reasonable angle for each joint). Once you are satisfied with how your program displays the robot, then animate it.

3.1 Window to Viewport Transformation

In order to display your robot, you will need to set up the appropriate *window to viewport* transformation. For example, the following *reshape callback function* defines a 36×36 *window* (with adjusted aspect ratio) with the *world coordinate* origin in the center. My robot fits nicely within this window.




```
void reshape(int w, int h) {
#define HALF_SIZE 18.0   /* window half-size */
  glViewport(0,0, w,h);  /* viewport is entire application window */

  matrixIdentity(Projection); /* set window into world coord system */
  if (w > h)
    matrixOrtho(Projection,
                -HALF_SIZE, HALF_SIZE, -HALF_SIZE*h/w, HALF_SIZE*h/w, -1,+1);
  else
    matrixOrtho(Projection,
                 -HALF_SIZE*w/h, HALF_SIZE*w/h, -HALF_SIZE, HALF_SIZE, -1,+1);

  matrixIdentity(ModeView); /* modelview matrix = identity */
}
```

3.2 Double Buffering

Double buffering is a standard technique to reduce "animation flicker" and is easy to implement with GLUT. When the application window is created, make sure the `GLUT_DOUBLE` option is specified. Then each time your `display()` callback is invoked, all drawing is performed in an "offscreen frame buffer." When a new frame has been completely rendered offscreen, you invoke `glutSwapBuffers()` which swaps the offscreen buffer with the display buffer.

```
int main(...) {
 ...
 glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE); /* double buffering */
 ...
 glutCreateWindow("Dancing Robot");
 ...
 glutDisplayFunc(display);
 ...
}

void display(void) {
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT);  /* clear offscreen buffer to black */
  /* render robot in offscreen buffer */
  glutSwapBuffers();  /* swap offscreen/display buffers */
}
```

3.3 Idle Events

GLUT allows you to install an *idle event callback* with the `glutIdleFunc()` function. The purpose of such a callback is to allow a program to continuously invoke some function whenever window system events are not being received. In this case, your callback alters the various joint rotations of your robot and requests a display event via the `glutPostRedisplay()` function. Note that the idle callback should return quickly for reasonable performance.

The `idle()` callback should perform *no* rendering; Instead, it should modify the animated model's parameters and request a redisplay.

```
static void idle(void) {
  /* update object's parameters */
  glutPostRedisplay();
}
```

3.4 Timing

In order for the animation to have a consistent behavior from system to system, some measurement of elapsed time is necessary. In order to determine the number of milliseconds that have passed since GLUT was initialized, call `glutGet()` with an argument of `GLUT_ELAPSED_TIME`. Your model can then be updated based on the amount of time that has elapsed:

```
static void idle(void) {
  GLfloat seconds = glutGet(GLUT_ELAPSED_TIME)/1000.0;
  /* update object based on current time */
  glutPostRedisplay();
}
```

I define constants for the minimum and maximum angles (θ_{min} and θ_{max}) allowed for each joint, and the frequency f which controls how rapidly each joint pivots. I then use the following formula to determine the current joint angle at time t :

$$\theta = (\theta_{max} - \theta_{min}) \sin(2f\pi t) + (\theta_{min} + \theta_{max})/2 \tag{1}$$

I defined the following function to evaluate this equation (I specify all angles in degrees since this is what `glRotate()` uses):

```
float getAngle(float freq, float min, float max, float t) {
  return (max - min)*sin(freq*2*M_PI*t) + 0.5*(min + max);
}
```

For example, in my `idle()` function, I update the robot's left elbow angle (stored in the global variable `leftElbowAngle`) as follows:

```
leftElbowAngle = getAngle(LEFT_ELBOW_FREQ,
                          LEFT_ELBOW_MIN, LEFT_ELBOW_MAX,
                          seconds);
```

4 Use Modern OpenGL

Make sure you are using a moderm dialect of OpenGL which requires you to install your own vertex and fragment shaders and roll your own matrix manipulation routines; you are free to use my code (or another author's code) – just make sure you give proper attribute to the author(s).

5 Submitting your solution

All of your source code, documentation, executables, etc. . . will be archived in to a single compressed archive file and submitted electronically. With every project submission you are to include a text file named `README` that includes

1. The name of all authors (which hopefully includes you), your student ID number, and an email address you can be contacted at.
2. A brief description of what you are submitting – enough, so a novice can understand what you have done. This description should avoid implementation details, but just give the user an idea what the program is.
3. A description of how to build and use your executable(s). For example, how to properly invoke make on a Unix system.
4. A list of all files that should be in the archive, and a one line description of the file.

Follow the instructions at the following site to submit your solution:

`http://ezekiel.vancouver.wsu.edu/~cs442/submit/submit.html`

Your project is due at midnight on the due date. Have fun!