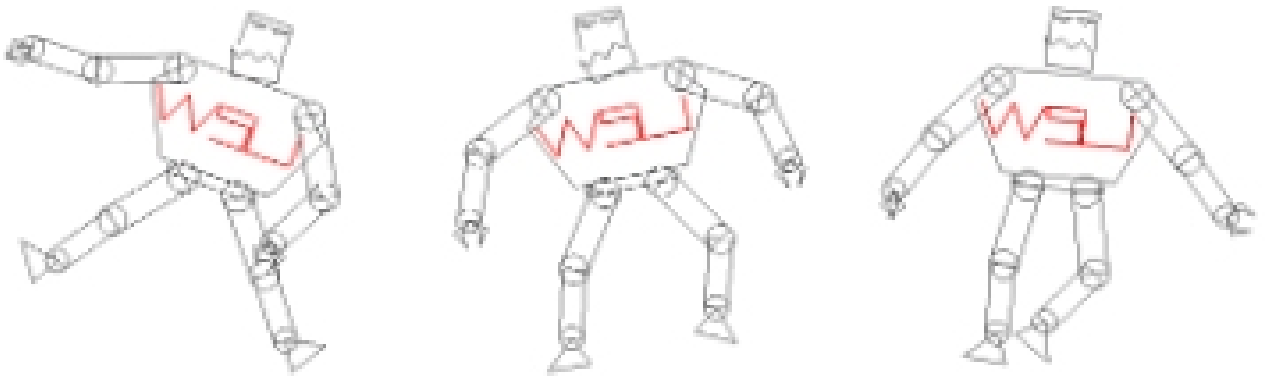
Programming Project #2
WebGL Dancing Robot

CS 442/542

Due 11:59 pm Tuesday, October 14, 2014



1 Overview

This project is designed to give you some experience with *hierarchical modeling* and *object instancing* in WebGL. 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. What to submit is described in Section 4.

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 Canvas and GL Context

WebGL renders to a HTML5 Canvas element

```
<canvas id="mycanvas" width=640" height="380"></canvas>
```

Which you can access from the HTML *Document Object Model* (DOM) from JavaScript and fetch a GL context to render to:

```
canvas = document.getElementById("mycanvas");
gl = canvas.getContext("experimental-webgl");
gl.viewport(0,0,canvas.width,canvas.height);
```

Here we set the viewport transformation to render to the entire canvas area.

3.2 Shaders

3.2.1 Vertex Shader

We use a very simple *vertex shader* that simply transforms the input vertex position with the currently loaded `ModelViewProjection` matrix:

```
<script id="vertex" type="x-shader">
  attribute vec2 vertexPosition;
  uniform mat4 ModelViewProjection;
  void main() {
     gl_Position = ModelViewProjection*vec4(vertexPosition, 0.0, 1.0);
  }
</script>
```

3.2.2 Fragment Shader

The *fragment shader* outputs a single color for each primitive:

```
<script id="fragment" type="x-shader">
   precision mediump float;
   uniform vec3 objectColor;
   void main() {
       gl_FragColor = vec4(objectColor, 1.0);
   }
</script>
```

3.2.3 Installing your shaders

The following boilerplate code loads, compiles, and links are shaders into a single `program` and tells WebGL to use it:

```
var v = document.getElementById("vertex").firstChild.nodeValue;
var vs = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vs,v);
gl.compileShader(vs);
if (!gl.getShaderParameter(vs,gl.COMPILE_STATUS))
    console.log(gl.getShaderInfoLog(vs));

var f = document.getElementById("fragment").firstChild.nodeValue;
var fs = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fs,f);
gl.compileShader(fs);
if (!gl.getShaderParameter(fs,gl.COMPILE_STATUS))
    console.log(gl.getShaderInfoLog(fs));

program = gl.createProgram();
gl.attachShader(program, vs);
gl.attachShader(program, fs);
gl.linkProgram(program);

gl.useProgram(program);
```

3.2.4 Fetch vertex attribute and uniform variable link locations

We need to fetch the locations in our shader program for the vertex attributes and the uniform variables:

```
program.ModelViewProjection = gl.getUniformLocation(program,
                                                    "ModelViewProjection");
program.objectColor = gl.getUniformLocation(program, "objectColor");
program.vertexPosition = gl.getAttribLocation(program, "vertexPosition");
```

3.3 Draw primitives

We can draw our primitives out of points, lines, or triangles using either `gl.drawArrays` or `gl.drawElements`. Before rendering, we need to create *Vertex Buffer Objects* (VBO's) and to store our geometry in GL's address space. Below the `joint` object represents the unit circle with a 20 sided polygon:

```
var joint = {
    numVerts : 20,
    vbuffer : -1,
    loadVBO : function() {
        var verts = new Array(2*this.numVerts);
```