24-783 Problem Set 9

Before starting update your public, data, and course_files directories by typing:

svn update ~/24783/src/public

svn update ~/24783/data

svn update ~/24783/src/course_files

Deadline: Two-week assignment. Please see Canvas for the deadline.

Preparation: Set up CMake projects

You first create projects for the two problem sets.

1. In the command line window, change directory to:
   ~/24783/src/*yourAndrewId*
2. Update the course_files, data, and public repositories.
3. Use "svn copy" to copy base code to your directory:
   `svn copy ~/24783/src/course_files/ps9 .`
4. Add ps9/ps9_1 and ps9/ps9_2 sub-directories to your top-level CMakeLists.txt
5. Run CMake, compile, and run ps9 executable.
6. Commit to the SVN server.

Optional: Check out your directory in a different location and see if all the files are in the server.

PS9-1 Plain 2D Renderer

The purpose of this assignment is to give you an experience of writing GLSL programs.

In this assignment, you write a GLSL program (a pair of vertex and fragment shaders) called Plain2DRenderer. In this practice you write a GLSL code so that the program renders 2D graphics correctly.

(1) The vertex shader in the base code takes a 2D vector as an attribute and passes it to gl_Position with no transformation. Modify the vertex shader so that incoming (0,0), (800,0), (0,600), (800,600) are transformed to (-1,1), (1,1), (-1,-1), (1,-1), respectively. (Leave z,w components 0 and 1).
(2) The fragment shade is always output (0,0,0,1). Modify vertex shader and fragment shader so that incoming color is written to the pixels.

Once (1) is completed, the program works as the cannon-ball game that we have been using, but everything is rendered black and white. When (2) is done you will see the colors.

In PS9-1 you do not have to touch the C++ cod and CMakeLists.txt. Only GLSL programs. In fact you only need to add/modify less than five lines.

In Mac and Linux environment, unless a C++ file is updated GLSL programs won't be copied to the runtime directory. Use 'touch' command to force the time stamp to be updated when you modify GLSL programs.

PS9-2 Drawing Cubic Bezier Surface with a GLSL program

Cubic Bezier surfaces are very often used in CAD packages. It defines a surface with 16 control points. A point on the surface is a function of two parameters (s,t).

The vertex shade in the base code passes the incoming parameter (s,t) with no transformation to gl_Position, and the fragment shader always writes (0,0,1,1).

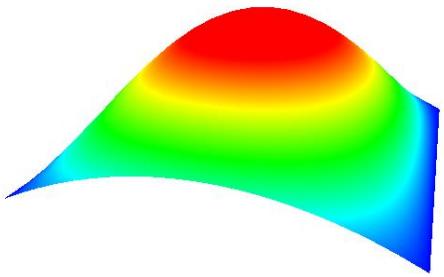
(1) Modify the vertex-shader so that the output gl_Position is a point on the Bezier surface defined by the 16 control points (already given as ctp[16] in the vertex shader) and the incoming parameter (param.x and param.y), transformed by the modelview and projection matrices (already given in the vertex shader). Cubic Bezier calculation cod is shown in C++ in the next page. You can get a point on the Bezier surface by giving 16 control points and parameter to CubicBezierSurface function. You need to translate this code into GLSL code. (Or, write your own.)

(2) Modify the vertex shader and the fragment shader so that the output color is a rainbow color calculated based on the Y coordinate of the point BEFORE transformed by projection and modelview. You can use RainbowColor function already given in the base code.

Note: Color calculation must be done in the fragment shader. Don't calculate color in the vertex shader and give as a varying to the fragment shader, nor don't calculate color in the C++ program and give to the vertex shader as an attribute.

If you successfully implement the renderer and send the input uniforms and attributes correctly, you will see an image like the following.

You also don't have to modify C++ code and CMakeLists.txt in PS9-2 unless you go for 10 extra points.



10 Points Bonus: Animate the Bezier surface by making Y coordinate of the four corner control points as 2.0*sin(currentTime), and four center control points as 1.0+cos(currentTime)*2.0, where currentTime is (double)FsSubSecondTimer()/1000.0.

(Cubic-Bezier Calculation)

```
YsVec3 CubicBezierCurve(const YsVec3 ctp[4],double s)
{
    // Recursive form of Cubic Bezier Curve.
    // (1) Interpolate ctp[0-1], ctp[1-2], and ctp[2-3] with parameter s -> a[0], a[1], a[2]
    // (2) Interpolate a[0-1], a[1-2] with parameter s -> b[0], b[1]
    // (3) Interpolate b[0-1]
    // Easy to remember.  You don't have to remember the Bezier parameter equation.
    const YsVec3 a[3]=
    {
        ctp[0]*(1.0-s)+ctp[1]*s,
        ctp[1]*(1.0-s)+ctp[2]*s,
        ctp[2]*(1.0-s)+ctp[3]*s,
    };
    const YsVec3 b[2]=
    {
        a[0]*(1.0-s)+a[1]*s,
        a[1]*(1.0-s)+a[2]*s,
    };
    return b[0]*(1.0-s)+b[1]*s;
}

YsVec3 CubicBezierSurface(const YsVec3 ctp[16],double s,double t)
{
    // Calculating a point on a Cubic Bezier Patch.
    // (1) First, calculate a point on four Bezier curves defined by ctp[0-3], [4-7], [8-11], [12-15], and
parameter s -> a[0], a[1], a[2], a[3]
    // (2) Calculate on a point on the Bezier curve defined by a[0-4] and parameter t.
    // Easy to remember.  You don't have to remember the Bezier parameter equation.
    const YsVec3 a[4]=
    {
        CubicBezierCurve(ctp   ,s),
        CubicBezierCurve(ctp +4,s),
        CubicBezierCurve(ctp +8,s),
        CubicBezierCurve(ctp+12,s),
    };
    return CubicBezierCurve(a,t);
}
```