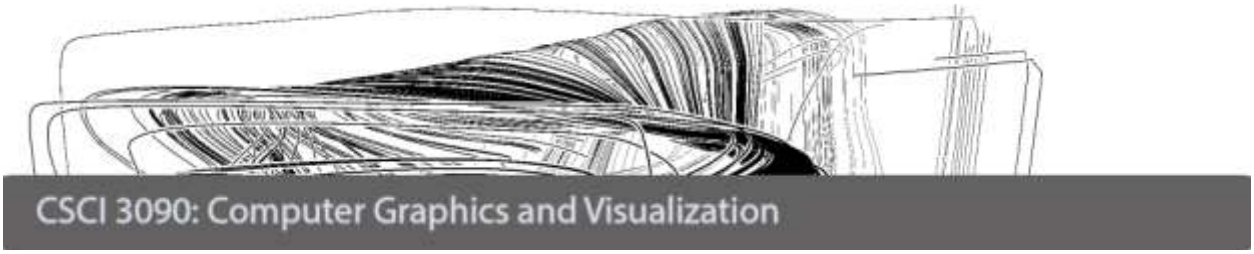


Laboratory Nine: Point Sprites

Introduction

Point sprites are an interesting graphics primitive which are constructed from points. A point sprite is basically a point that has a size. When OpenGL normally draws a point it makes it one pixel in size. But, we can change the size either in our C++ program or in a shader program. When we do this the point is replaced by a square, where the lengths of the sides of the square are equal to the point size in pixels. The square is drawn in screen space so it is always normal to the vector from the eye to projection plane.

When we draw a point sprite, the fragment shader is called for each pixel that is covered by the point sprite. The variable gl_PointCoord is a 2D vector whose components vary from 0 to 1 over the surface of the point sprite. We can use this variable to compute a different colour for each pixels covered by the point sprite.

Start this lab by making a copy of the example 8 program. This program has the basic framework that we need. The part of the program that processes the OBJ file will be deleted since we will generate a random set of points

Modifications to the C++ code

As noted above you can delete all the parts of the program that deal with OBJ files and you will not need to include the OBJ loader files in your project. This mainly impacts the init() procedure. We will use the random functions from the C++ library so add the following to the list of include files at the start of the program:

```
#include <random>
```

We will use the random functions to generate a number of random points inside a cube bounded by -2.0 to 2.0 in the x, y and z directions. The number of points is given by the variable np, which is declared at the start of the program:

```
const int np = 200;
```

The code for our new init() procedure is:

```
void init() {
	GLuint vbuffer;
```

```
GLuint ibuffer;
GLint vPosition;
int vs;
int fs;
GLfloat *vertices;
GLuint *indices;
int i;
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(-2.0,2.0);

textureStruct *texture;

glGenVertexArrays(1, &objVAO);
glBindVertexArray(objVAO);

vertices = new GLfloat[4*np];
for(i=0; i<np; i++) {
	vertices[4*i] = distribution(generator);
	vertices[4*i+1] = distribution(generator);
	vertices[4*i+2] = distribution(generator);
	vertices[4*i+3] = 1.0;
}

indices = new GLuint[np];
for(i=0; i<np; i++)
	indices[i] = i;

/*
 *  load the vertex coordinate data
 */
glGenBuffers(1, &vbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
glBufferData(GL_ARRAY_BUFFER, 4*np*sizeof(GLfloat), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, 4*np*sizeof(GLfloat), vertices);

/*
 *  load the vertex indexes
 */
glGenBuffers(1, &ibuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, np*sizeof(GLuint), indices, GL_STATIC_DRAW);

/*
 *  compile and build the shader program
 */

vs = buildShader(GL_VERTEX_SHADER, "lab9.vs");
fs = buildShader(GL_FRAGMENT_SHADER, "lab9.fs");
program = buildProgram(vs,fs,0);

/*
 *  link the vertex coordinates to the vPosition
 *  variable in the vertex program.
 */
glUseProgram(program);
vPosition = glGetAttribLocation(program,"vPosition");
glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(vPosition);
```

```
        /*
         *  Create the texture.
         */

        texture = loadImage("star.jpg");

        glGenTextures(1, &texName);
        glBindTexture(GL_TEXTURE_2D, texName);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture->width, texture->height,
               0, GL_RGB,GL_UNSIGNED_BYTE,  texture->data);
        glGenerateMipmap(GL_TEXTURE_2D);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

}
```

Note that the majority of the changes are at the start of this procedure where we generate the random coordinate values.

There are also a few changes to the display function. Most of these changes are deleting code that we no longer need. Note that the second parameter to lookat() has been changed.

```
void displayFunc() {
        glm::mat4 view;
        int modelViewLoc;
        int projectionLoc;
        int normalLoc;
        int eyeLoc;

        view = glm::lookAt(glm::vec3(eyex, eyey, eyez),
                                glm::vec3(0.0,0.0,0.0),
                                glm::vec3(0.0f, 1.0f, 0.0f));


        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glUseProgram(program);
        modelViewLoc = glGetUniformLocation(program,"modelView");
        glUniformMatrix4fv(modelViewLoc, 1, 0, glm::value_ptr(view));
        projectionLoc = glGetUniformLocation(program,"projection");
        glUniformMatrix4fv(projectionLoc, 1, 0, glm::value_ptr(projection));

        glBindTexture(GL_TEXTURE_2D, texName);

        glBindVertexArray(objVAO);
        glEnable(GL_POINT_SPRITE);
        glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);

        glDrawElements(GL_POINTS, np, GL_UNSIGNED_INT, NULL);

        glutSwapBuffers();

}
```

We need to enable the point sprites and the ability to change the point size in our vertex program. These two lines are not required on the most recent graphics cards, but it is a good idea to include them (I needed them for my laptop).

Since the points are randomly distributed in a cube close to the origin the initial eye position should be changed so that z=5.0 and r=5.0. These changes are made close to the end of the main() procedure.

Exercise One

The first thing we will do is texture map our sprite. The file star.jpg is available on Blackboard in the folder for this lab. Our program already reads this texture in and sets it up for the fragment shader. We now need to write the vertex and fragment shaders that will draw our pixel sprites. The vertex shader that we will use is:

```
*
 *  Simple vertex shader for Lab 9
 */

#version 330 core

in vec4 vPosition;

uniform mat4 modelView;
uniform mat4 projection;

void main(void) {

	vec4 pos = projection * modelView * vPosition;
	gl_PointSize = (1.0 - pos.z/pos.w) * 64.0;
	gl_Position = pos;
}
```

It starts by computing the screen space position of the point using the standard technique. It then computes the size of the pixel sprite based on its distance from the near clipping plane. The maximum size of the sprite will be 64 pixels if it lies on the near clipping plane. It will be zero if it is on the far clipping plane.

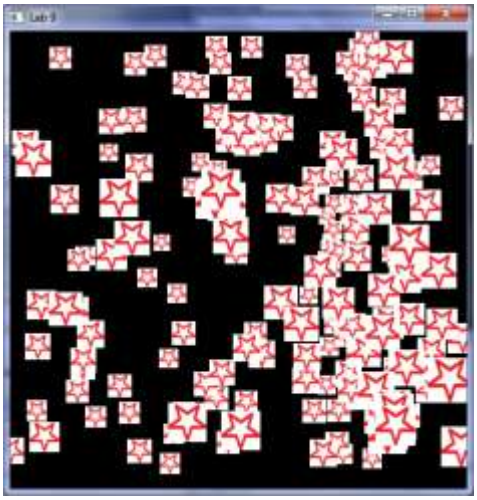
The fragment shader for this experiment is:

```
*
 *  Simple fragment shader for lab 9
 */

#version 330 core
```

```
uniform sampler2D tex;

void main(void) {

        gl_FragColor = texture(tex, gl_PointCoord);

}
```

This program uses the coordinates within the sprite as texture coordinates and extracts the texture values at that point as the colour of the pixel.

Put all the pieces together and run the program. You should get a result similar to the following:



Exercise Two

In the next experiment we will use the fragment program to compute the pixel colours instead of using a texture map. The vertex shader for this program is the same as the previous one. Most of the changes are in the fragment shader, which has the following code:

```
/*
 *  Simple fragment shader for lab 9
 */

#version 330 core

uniform sampler2D tex;

void main(void) {
        const vec4 colour1 = vec4(0.6, 0.0, 0.0, 1.0);
        const vec4 colour2 = vec4(0.9, 0.7, 1.0, 0.0);
```

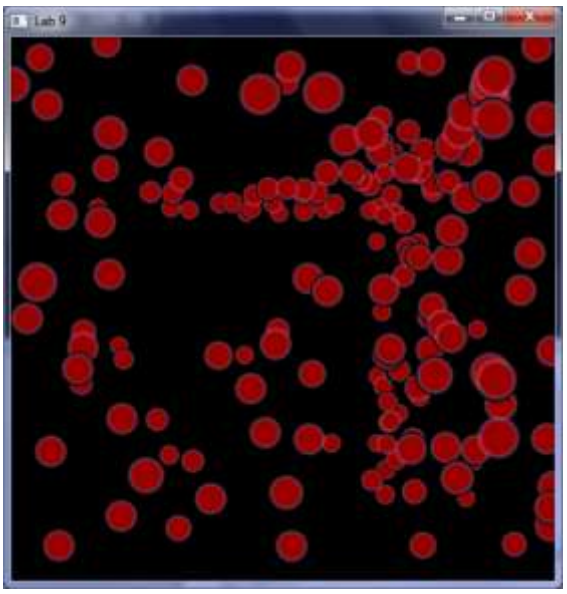
```
        vec2 temp = gl_PointCoord - vec2(0.5);
        float f = dot(temp,temp);

        if(f > 0.25)
                discard;

        gl_FragColor = mix(colour1, colour2, smoothstep(0.1, 0.25, f));
}
```

This shader draws the point sprite as a circle with the colour varying based on the distance of the pixel from the center of the circle. The shader starts by computing the distance from the center of the point sprite to the current pixel. It then compares this to r^2 for the sprite (remember our coordinates go from 0.0 to 1.0). If the distance is greater than r^2 the pixel is discarded, that is it is not drawn. Otherwise the distance is used to interpolate between the two colours. This is done by the mix function. The smoothstep function is used to construct the value used to control the interpolation. If the value the third parameter is less than the first parameter 0.0 is returned. If the value is greater than the second parameter 1.0 is returned. Otherwise a Hermite curve is used to smoothly interpolate the value of the third parameter.

Since one of the colours has an alpha component we need to turn on alpha blending, which is done by adding the following two lines before the call to glDrawElements:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

After doing this you should get a result that looks like the following:



Laboratory Report

Submit screenshots of your results in the Blackboard dropbox.