Advanced Computer Graphics Assignment 2: Point Based Rendering

Due date: check nestor

1 Assignment and grading

The complete assignment will be graded on a 0-100 scale, and the score will be broken down as follows

- 20 points: Questions
- 10 points: Rendering points
- 20 points: Point scaling vertex shader
- 20 points: Ray-casting the splats
- 30 points: Gouraud shading

It is important that you understand what you are doing, so we want you to answer some questions that are interleaved in the assignments. Answers should not be too long, just convey the basic idea. The answers to the questions should be saved to a file called questions2.txt (Note: questions2.txt is supplied, and has the questions written out). You will want to answer them as you complete the assignment, since many of them require experimentation.

What to hand in: Please hand in C and Cg sources for your code for each of the assignments. Also, after you filled in the answers to the questions in file `questions2.txt`, archive everything and upload your archive to your group's file exchange.

2 Overview

Old graphics hardware was restricted to rendering meshes consisting of triangles and other simple primitives. This changed when GPUs with a programmable pipeline were introduced, as it made it possible to use other rendering paradigms, such as Point Based Rendering. In this lab session, you will implement various techniques to do Point Based Rendering using point splatting.

3 Reading material

- The course slides (can be found on nestor)

- Cg User's Manual and Reference Manual
- OpenGL reference pages http://www.opengl.org/sdk/docs/man/, for documentation on GL extensions (such as GL_EXT_framebuffer_object) see the extension registry http://www.opengl.org/registry/.
- Optional: the book Point-Based Graphics, by Markus Gross

4 Getting started

To not waste time on getting the initialization working, you'll want to start with a skeleton program that initializes Cg, GLUT and GLEW. Download the skeleton project from Nestor to get started. Once compiled, it will show a colorful rotating quad.

5 Rendering points

A Surfel structure has already been defined,

```
struct Surfel
{
    float pos[3];
    float color[4];
    float uvec[3];
    float vvec[3];
};
```

and data for 1000 points (surfels) distributed around the unit sphere is provided in this format in the include file `points2.h`. You can use these throughout the assignment to test your point based rendering algorithms. Start by changing your program to render the surfels as plain colored points. Use Vertex Arrays (`glDrawArrays`). Your result should look like Figure 1.

Question 1: What is the advantage of using Vertex Arrays above immediate mode rendering (using `glBegin`/`glEnd`)?

Question 2: What are `uvec` and `vvec` in the `Surfel` structure? What properties (name at least two) of the splat do they represent?

6 Point scaling vertex shader

By default, OpenGL renders points of size 1, which results in big holes in the surface when viewed from nearby. To close these, we need to make the points bigger depending on the distance to the viewer, which is accomplished by applying perspective transformation to the point size. Implement a vertex program that does this. A basic one is given in the course slides. In the last lab session you were shown how to load and how to bind shader programs, so that they are applied when rendering.

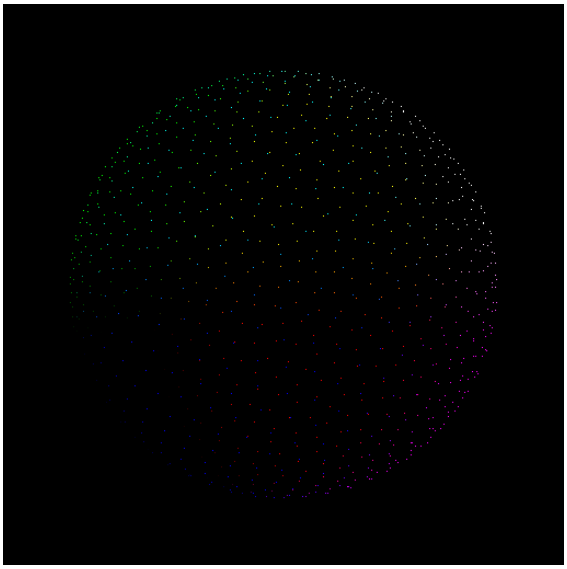

Figure 1: *Colorful points distributed on a sphere (look at digital version and zoom in if necessary).*

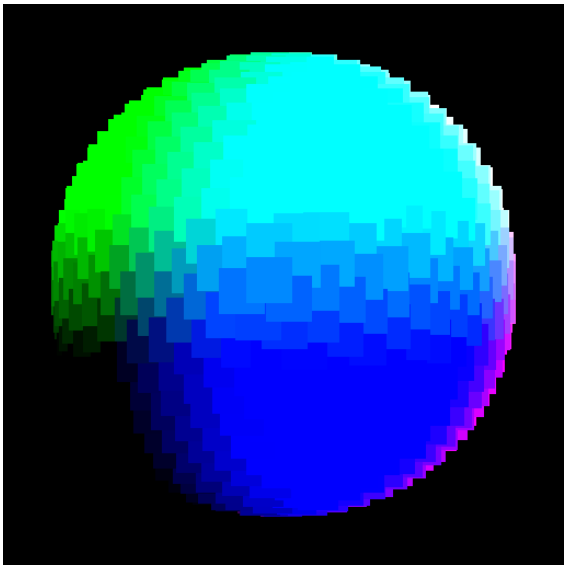

Figure 2: *Colorful blocks distributed on a sphere.*

Question 3: On the slide, it is not shown how to compute `peye`. This is the position in of the point in eye (view)-space. What matrix should be used to transform a position from object-space to eye-space?

- When you use a vertex shader it is easiest to pass the surfel attributes in texture coordinates, so that they come into the shader as Cg binding semantic `TEXCOORD0`, `TEXCOORD1`, ...
- You have to re-implement $gluPerspective$ using $glFrustum$ to get hold of the various frustum parameters that are needed in the equations: far plane distance f, near plane distance n, left l, right r, bottom b and top t. These can then be passed to your Cg shader as uniform parameters (or used to form a single multiplication factor to pass to your shader).

Question 4: Does it matter whether we pass the surfel color to the vertex shader in a texture coordinate (glTexCoord...) or as the vertex color (glColor...)? Why (not)?

Your result should look like Figure 2.

7 Ray-casting the splats

The resulting surface will be blocky and ugly, as the square points are clearly noticeable. In this section you will apply a ray-casting fragment shader to the splats so that they are rasterized in a perspectively correct way. Start with the one given in the course slides.

- The $\mathbf{u}$ and $\mathbf{v}$ vectors that are passed to the vertex shader are in object-space. To simplify ray-casting, the eye must be in the point of origin.

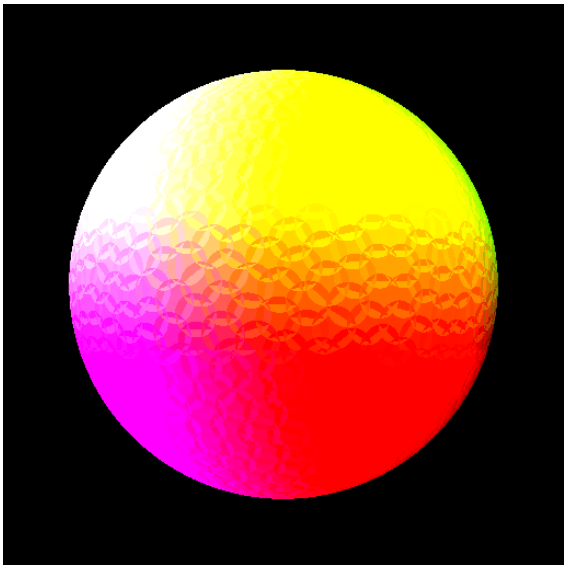

Figure 3: *Sphere rendered with splats using a ray-casting fragment shader.*

Figure 4: *Gouraud shaded sphere.*

- The current pixel position of the fragment enters the fragment program through the `WPOS` binding semantic. Use this to compute the position on the near plane, $\mathbf{q}_n$.
- Solve u, v and λ for the intersection point between ray and the splat tangent plane. Try to move as many computations as possible to the vertex shader.
- Reject fragments if they hit the tangent plane too far from the splat center (distance > 1).
- Return the surfel color if the fragment passes.
- Compute the depth for correct overlap of splats, and return this in the `DEPTH` binding semantic of the fragment shader.

Question 5: What matrix should you use to transform the $\mathbf{u}$ and $\mathbf{v}$ vectors? Is it different from the matrix you would use to transform a normal? Why (not)?

Question 6: Why would we want to move as much computations as possible from the fragment program to the vertex program, or even from the vertex program to the application, in case they are constant over the entire rendering pass?

Question 7: Does it matter which values the outputs have when a fragment is discarded?

Your result should now look like Figure 3.

8 Gouraud shading

At least the sphere should be round now, but there is no smooth surface in the areas where splats overlap. In this section you will implement Gouraud shading, which interpolates the surfel color in screen-space. With Gouraud shading you will need to do two passes over the points: a visibility splatting pass, and a blending pass. These will be rendered to a floating point off-screen render target for more precision. Finally, the result will be rendered to the screen in the normalization pass.

- In the `reshape` function, create a FBO (Frame Buffer Object) as intermediate render target, and attach a 16 bit RGBA floating point texture (format `GL_RGBA16F_ARB`), and a render buffer for the depth (format `GL_DEPTH_COMPONENT`):

```
glDeleteFramebuffersEXT(1, &fbo);
glDeleteRenderbuffersEXT(1, &depthbuffer);
glDeleteTextures(1, &color_tex);

glGenTextures(1, &color_tex);
glBindTexture(GL_TEXTURE_2D, color_tex);
glTexImage2D(GL_TEXTURE_2D, 0, ..., w_width, w_height, 0, GL_RGBA,
  GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glBindTexture(GL_TEXTURE_2D, 0);

glGenRenderbuffersEXT(1, &depthbuffer);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depthbuffer);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, ...,
  w_width, w_height);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);

glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, ...,
  GL_TEXTURE_2D, color_tex, 0);
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, ...,
  GL_RENDERBUFFER_EXT, depthbuffer);

assert(glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT) ==
  GL_FRAMEBUFFER_COMPLETE_EXT);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

  Note that the `glDelete*` functions ignore zero values, so the above is safe if you initialize the handles to zero.

  As you can see, texture filtering for the intermediate texture is set to `GL_NEAREST` for efficiency, because no interpolation is necessary when rendering to the equally-sized screen. Fill in the areas denoted by ... yourself.

- Make sure that the rendering of splats happens to this FBO.
- Implement the visibility splatting pass. Rasterize the splats in the depth buffer only, so that blending only happens within a band around the nearest surface. To accomplish this, add an offset to the depth in the fragment shader. Before rendering, disable color rendering (`glColorMask`) and enable depth write (`glDepthMask`).
- Implement the blending pass. First set up alpha blending for accumulating the kernels, re-enable color rendering, and disable depth write (do *not* forget to re-enable depth write again, as otherwise `glClear` will also not be able to clear the depth buffer for the next frame).
- In the fragment shader use the `smoothstep` function to compute a weight that is 1 at the center of the splat and 0 on the border.
- Implement the normalization pass, which renders a full screen quad using a normalizing fragment shader, sourcing from the intermediate texture.

Question 8: What is the difference between a texture and a render buffer?

Question 9: Why must depth write be disabled when blending the splats?

Your result should now look like Figure 4.