02564, REAL-TIME GRAPHICS - Exercice 1 Getting to know shaders

OLIVIER ROUILLER - S090842
Department of Informatics and Mathematical Modelling
Technical University of Denemark

February 7, 2010

1 Part 1

In this part of the exercise, we modify the vertex and fragment programs `object.vert` and `object.frag` in order to implement a per fragment shading.

object.vert

```
varying vec3 norm, position;

void main()
{
 //  Compute eyespace normal and position
 norm = normalize(gl_NormalMatrix*gl_Normal);
 vec4 pos = gl_ModelViewMatrix*gl_Vertex;

 vec3 position = (gl_ModelViewMatrix*gl_Vertex).xyz;

 gl_TexCoord[0] = gl_MultiTexCoord0;
 gl_Position = ftransform();
}
```

object.frag

```
uniform sampler2D tex;
varying vec3 norm, position;

void main()
{
 vec4 color = gl_LightSource[0].ambient*vec4(0.2,0.2,0.2,0);

 vec3 normal = normalize(norm);
 //  If the normal points away from the viewer, flip it.
 if(dot(normal,-position)<0.0)
     normal = -normal;

 //  Compute dot product of normal and light source direction
 float d = dot(normal,normalize(gl_LightSource[0].position.xyz));
 vec3 h = normalize(gl_LightSource[0].halfVector.xyz);

 if(d>0.0)
 {
     color += d*gl_LightSource[0].diffuse*gl_FrontMaterial.diffuse;
     float s = dot(normal, h);
     if(s>0.0)
     {
         color += pow(s, gl_FrontMaterial.shininess)*
         gl_LightSource[0].specular*gl_FrontMaterial.specular;
     }
 }
 //  Simply multiply color by texture.
 gl_FragColor = color* texture2D(tex, gl_TexCoord[0].xy);
}
```

Below in figure 1.1 the scene rendered with a per vertex lighting and in figure 1.2 the same scene with a per fragment lighting.

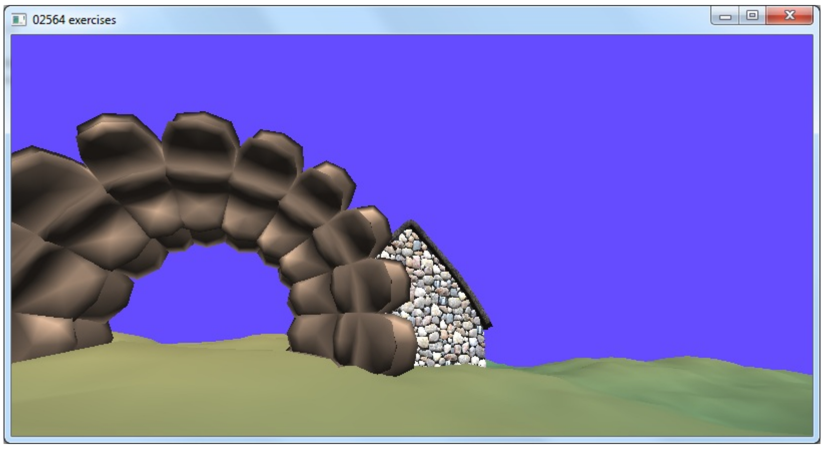

Figure 1.1: Per vertex shading

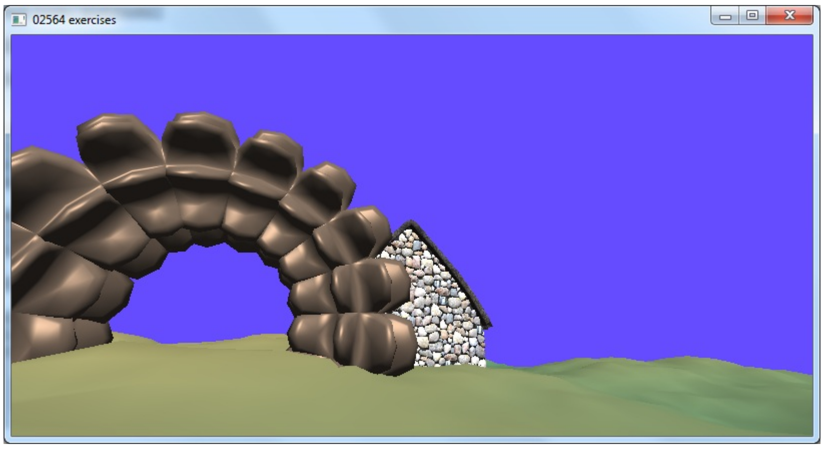

Figure 1.2: Per fragment shading

2 Part 2

In this exercise we implement a very simple steep mesurement to change the color of the areas of the terrain where the slope is high. The idea is to use the z value of the normal of the terrain in world coordinate. The lowest it is, the steepest the terrain is. In `terrain.vert` we pass the normal in world coordinate to the fragment shader and in `terrain.frag` we simply check if the z coordinate of the normal is under a threshold.

terrain.vert

```
varying float h;
varying vec3 norm, worldnorm;

void main()
{
 h = gl_Vertex.z/10.0;
 norm = gl_NormalMatrix * gl_Normal;
 worldnorm = gl_Normal;

 gl_Position = ftransform();
}
```

terrain.frag

```
varying float h;
varying vec3 norm, worldnorm;

const vec3 col_low = vec3(0.2,0.5,0.4);
const vec3 col_high = vec3(1,0.7,0.4);
const vec3 col_steep = vec3(0.3,0.3,0.3);

void main()
{
 float diff = dot(norm, gl_LightSource[0].position.xyz);

 if(normalize(worldnorm).z<0.99)
     gl_FragColor.rgb = diff*col_steep;
 else
     gl_FragColor.rgb = diff*((1.0-h)*col_low + h * col_high);
}
```

Below the terrain rendered without (figure 2.1) and with (figure 2.2) slope detection.

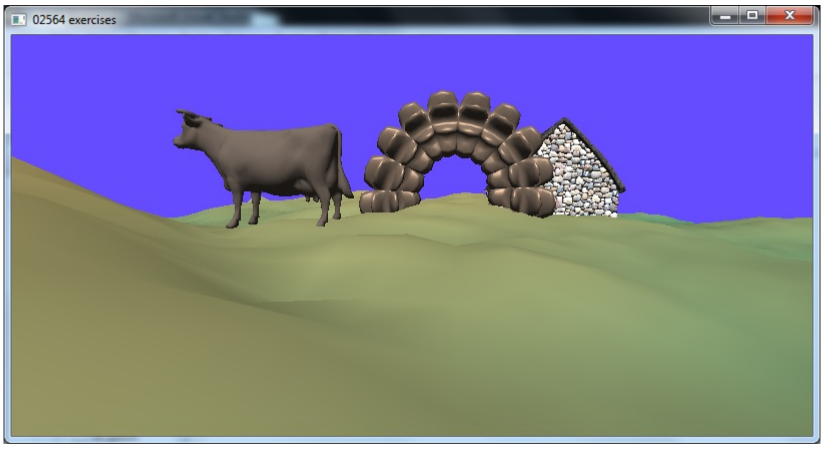

Figure 2.1: The terrain rendered without slope detection

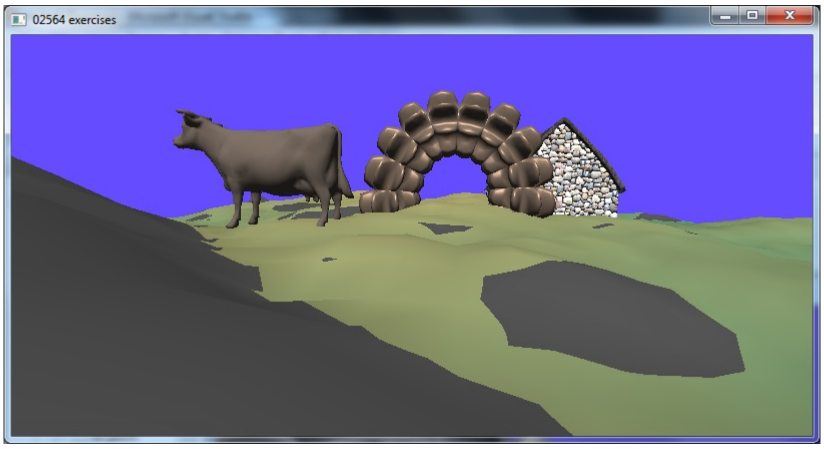

Figure 2.2: The terrain rendered with slope detection

3 Part 3

As a first application of the geometry shader we implement a way to render both lines and faces of the triangles in a single pass. Basicaly, a fragment whose position is inside a triangle is rendered white and a fragment on a line is rendered red. In the fragment shader we dispose of the distances of the fragment to each of the edges of the triangle and the color is computed according to the minimal distance to the edges with an exponential filtering function. The distances are computed per vertex and in screen coordinates in the geometry shader using the formula $d(p0, p1p2) = \frac{\overrightarrow{p0p1} \times \overrightarrow{p1p2}}{\|\overrightarrow{p1p2}\|}$.

tri.geom

```
//  ------------------ Geometry Shader --------------------------------
#version 150
#extension GL_EXT_gpu_shader4 : enable
#extension GL_EXT_geometry_shader4 : enable

uniform vec2 WIN_SCALE;
noperspective varying vec3 dist;

void main(void)
{
  vec2 p0 = gl_PositionIn[0].xy * WIN_SCALE / gl_PositionIn[0].w;
  vec2 p1 = gl_PositionIn[1].xy * WIN_SCALE / gl_PositionIn[1].w;
  vec2 p2 = gl_PositionIn[2].xy * WIN_SCALE / gl_PositionIn[2].w;

  // Vertex 0
  float cross = abs((p0.x-p1.x)*(p2.y-p1.y) - (p0.y-p1.y)*(p2.x-p1.x));
  float d =  cross/length( p2-p1 );
  dist = vec3(d,0,0);
  gl_Position = gl_PositionIn[0];
  EmitVertex();

  // Vertex 1
  cross = abs((p1.x-p2.x)*(p2.y-p0.y) - (p1.y-p2.y)*(p2.x-p0.x));
  d =  cross/length( p2-p0 );
  dist = vec3(0,d,0);
  gl_Position = gl_PositionIn[1];
  EmitVertex();

  // Vertex 2
  cross = abs((p2.x-p1.x)*(p0.y-p1.y) - (p2.y-p1.y)*(p0.x-p1.x));
  d =  cross/length( p0-p1 );
  dist = vec3(0,0,d);
  gl_Position = gl_PositionIn[2];
  EmitVertex();

  EndPrimitive();
}
```

Below the result (figure 3.2) of this technique when the distances are corectly computed.

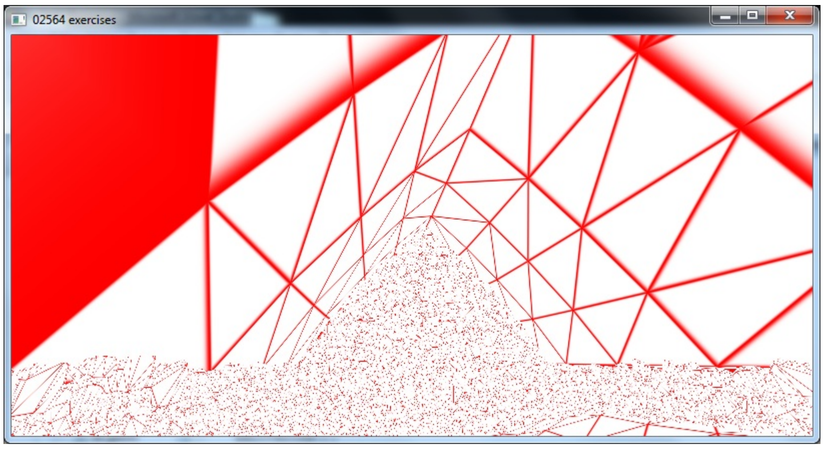

Figure 3.1: Wrong distances

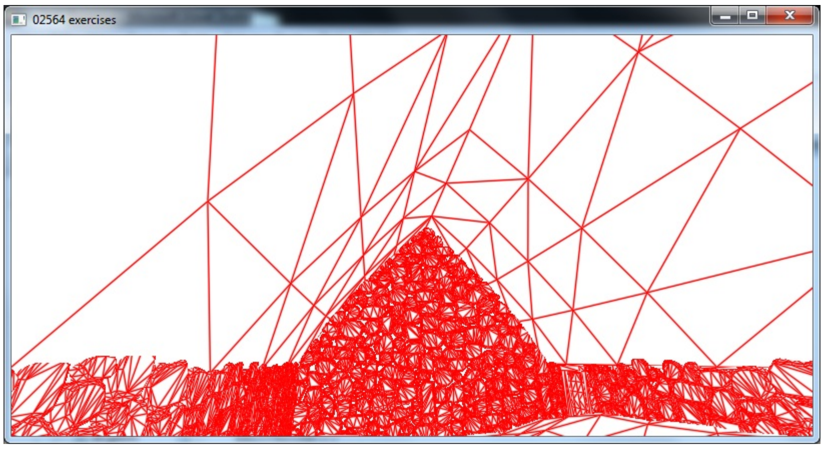

Figure 3.2: Right distances

4 Part 4

1. `gl\_NormalMatrix` is the transpose of the inverse of the `gl\_ModelViewlMatrix`.
2. To specify the material we call `glMaterial` wich exist in several versions depending on wether we pass direct values or an array and on the type of datas. To set a lightsource we call `glLight` wich also exists in several versions.
3. The value of a varying variable is computed for each primitive and is interpolated along the pipeline whereas an uniform variable is set in the application program and is constant as long as the application program does not change it.
4. Geometry shaders can be used for mesh refinement, level of detail, frustum culling, or to render several cube maps in a single pass to allow real time global illumination.