Lab 5
Geometry Shaders

CSE 4431/5331.03M

Advanced Topics in 3D Computer Graphics

TA: Margarita Vinnikov

mvinni@cse.yorku.ca

Moving to new Generation

Differences between OpenGL Shading Language versions

- http://en.wikipedia.org/wiki/OpenGL_Shading_Language

- Need to specify version

  #version 140

How to tell?

```
if (glewIsSupported("GL_VERSION_2_0"))
                printf("Ready for OpenGL 2.0\n");
else {
        printf("OpenGL 2.0 not supported\n");
        exit(1);
}

const GLubyte *renderer = glGetString( GL_RENDERER );
const GLubyte *vendor = glGetString( GL_VENDOR );
const GLubyte *version = glGetString( GL_VERSION );
const GLubyte *glslVersion = glGetString(
GL_SHADING_LANGUAGE_VERSION );
GLint major, minor;
glGetIntegerv(GL_MAJOR_VERSION, &major);
glGetIntegerv(GL_MINOR_VERSION, &minor);
printf("GL Vendor : %s\n", vendor);
printf("GL Renderer : %s\n", renderer);
printf("GL Version (string) : %s\n", version);
printf("GL Version (integer) : %d.%d\n", major, minor);
printf("GLSL Version : %s\n", glslVersion);
```

More Differences

- `ftransform()` is no longer available since GLSL 1.40 and GLSL ES 1.0.
  - Need to manage the projection and modelview matrices explicitly
- Use glm library
  - http://glm.g-truc.net/0.9.5/index.html

```
#include "glm\glm\glm.hpp"
#include "glm\glm\gtc\type_ptr.hpp"
#include "glm\glm\gtc\matrix_transform.hpp"
#include "glm\glm\/gtx/transform2.hpp“

glm::mat4 projView ;
glm::mat4 projProj;
glm::mat4 model;
glm::vec3 cameraPos;
GLint ModelMatrixLoc ;
GLuint ModelViewMatrixLoc;
GLuint NormalMatrixLoc;
GLuint MVPLoc;
GLuint projViewLoc;
GLuint WorldCameraPositionLoc;
```

```
#include "glm\glm\glm.hpp"
#include "glm\glm\gtc\type_ptr.hpp"
#include "glm\glm\gtc\matrix_transform.hpp"
#include "glm\glm\/gtx/transform2.hpp“

glm::mat4 projView ;
glm::mat4 projProj;
glm::mat4 model;
glm::vec3 cameraPos;
GLint ModelMatrixLoc ;
GLuint ModelViewMatrixLoc;
GLuint NormalMatrixLoc;
GLuint MVPLoc;
GLuint projViewLoc;
GLuint WorldCameraPositionLoc;

void setMatrices()
{
    glm::mat4 mv = projView * model;
    glm::mat3 normalM=  glm ::mat3( glm::vec3(mv[0]),
                  glm::vec3(mv[1]), glm::vec3(mv[2]) );
    glm::mat4 mvp = projProj * mv;

    glUniformMatrix4fv(ModelMatrixLoc, 1, GL_FALSE,
            &model[0][0]);
    glUniformMatrix4fv(ModelViewMatrixLoc, 1, GL_FALSE,
            &mv[0][0]);
    glUniformMatrix3fv(NormalMatrixLoc, 1, GL_FALSE,
            &normalM[0][0]);
    glUniformMatrix4fv(MVPLoc, 1, GL_FALSE,
            &mvp[0][0]);
    glUniformMatrix4fv(projViewLoc, 1, GL_FALSE,
            &projView[0][0]);
    glUniform3f(WorldCameraPositionLoc,cameraPos[0],
            cameraPos[1],cameraPos[2]);
    glUniform1f(explode_factorLoc, explode_factor);

}
```

```
void renderScene(void) {
..
     cx = radius * cos(angle1)*cos(angle2);
     cy = radius * sin(angle1)*cos(angle2);
     cz = radius * sin(angle2);

     cameraPos = glm::vec3(cx, cy, cz);
     glm::vec3 projAt = glm::vec3(lx,ly,lz);
     glm::vec3 projUp = glm::vec3(0.0f,0.0f,1.0f);
     projView = glm::lookAt(cameraPos, projAt, projUp);

     model = glm::mat4(1.0f);
     model *= glm::translate(glm::vec3(0.0f,0.3f,0.0f));
     model *= glm::rotate(radius,
            glm::vec3(0.0f,0.0f,1.0f));
     GL_FALSE,&model[0][0]);
     …
     setMatrices();
}
void changeSize(int w, int h)
{
     glViewport(0, 0, (GLsizei) w, (GLsizei) h);
     projProj = glm::perspective(50.0f, (float)w/h, 0.3f,
            1000.0f);
}
```

More Differences – Fragment Shader

- In GLSL 1.30 and later you can do
  - `glBindFragDataLocation( Program, 0, "MyFragColor");`
- where:
  - Program - your shader program's handle;
  - 0 - color buffer number, associated with the variable; if you are not using multiple render targets, you must write zero;
  - "MyFragColor" - name of the output variable in the shader program, which is associated with the given buffer.

```
#version 150
out vec4 MyFragColor;
void main(void) {
 MyFragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

More Differences - Layout Qualifier

- Indicates where the storage for a variable comes from
  - Defined as

```
layout(qualifier1, qualifier2 = value, ...) variable definition
```

- Setting Vertex Attributes (CreateVBO)
  - Recall glVertexAttribPointer provides OpenGL with all of the necessary metadata to use the block of raw data that uploaded to the GPU's using glBufferData.
  - The first parameter to glVertexAttribPointer is index
    - User-maintained number
    - used to identify those parameters marked with a layout qualifier in a GLSL shader program
    - For example :
      - if we changed the index parameter to 18, we must also update the GLSL shader's layout qualifier's location argument to 18:

```
glVertexAttribPointer(18, 4, GL_FLOAT, GL_FALSE, 0, 0);

layout(location=18) in vec4 in_Position;
```

- For more info and review on VBO:
  http://www.opengl.org/wiki/Layout_Qualifier_(GLSL)
  http://openglbook.com/the-book/chapter-2-vertices-and-shapes/

Vertex shader attribute index

- inputs can specify the attribute index that the particular input uses.

```
layout(location = attribute index) in vec3 position;
```

- With this syntax, you can forgo the use of glBindAttribLocation entirely.
  - If you try to combine the two and they conflict, the layout qualifier always wins.
- Attributes that take up multiple attribute slots will be given a sequential block of that number of attributes in order starting with the given attribute.
- For example:

```
layout(location = 2) in vec3 values[4];
```

  - This will allocate the attribute indices 2, 3, 4, and 5.

Fragment shader buffer output

- Specifies the buffer index that output writes to
  - This uses the same syntax as vertex shader attributes:
  - `layout(location = output index) out vec4 outColor;`
  - As with vertex shader inputs, this allows the user to forgo the use of glBindFragDataLocation. Similarly, the values in the shader override the values provided by this function.
- For dual source blending, the syntax includes a second qualifier:

```
layout(location = output index, index = dual output index) out vec4 outColor;
```

  - Again, this allows one to forgo the use of glBindFragDataLocationIndexed.

1nd Task

Setup

- Find out what is the latest version the machine you are working on is capable to accept
- Modify your current project to be compatible with that particular version
- Modify your vertex shader to utilize uniform matrices passed from CPU

Setting up geometry shader

- Vertex shader will run
  - once per vertex
- Geometry shader will run
  - once per primitive (point, line, or triangle),
- Fragment will run
  - once per fragment

Setting up geometry shader

- Use

```
glCreateShader(GL_GEOMETRY_SHADER);
```

- Same as the rest of shader programs
  - `glShaderSource()`
  - `glCompileShader()`
  - `glAttachShader()`
  - `glLinkProgram()`

- Draw geometry using a function like

```
glDrawArrays()
```

The primitives received by a geometry shader must match what it is expecting based in its own input primitive mode.

Allowed Draw Modes for Geometry Shader Input Modes

Allowed Draw Modes
GL_POINTS
GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP
GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP
GL_LINES_ADJACENCY
GL_TRIANGLES_ADJACENCY

Which mode is missing?

Vertex vs Geometry Shaders

Vertex

- built-in variables:

```
gl_Position, gl_PointSize,
and gl_ClipDistance[].
```

Or

```
out vec4 color;
out vec3 normal;
```

Or

```
out VertexData {
    vec4 color;
    vec3 normal;
} vertex;
```

Geometry

- The predefined inputs :
  - stored in a built-in array called gl_in

```
in gl_PerVertex {
        vec4 gl_Position;   float
gl_PointSize;       float
        gl_ClipDistance[];
} gl_in[];
```

Or

```
in vec4 color[];
in vec3 normal[];
```

Or

```
in VertexData {
 vec4 color;
 vec3 normal;
 // More per-vertex attributes can be
        inserted here
} vertex[];
```

Input/Output Arrays

```
layout (primitive_type) in;
layout (primitive_type) out;
```

Geometry Shader Input Mode	Allowed Draw Modes
points	GL_POINTS
lines	GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP
triangles	GL_TRIANGLES, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP
lines_adjacency	GL_LINES_ADJACENCY
triangles_adjacency	GL_TRIANGLES_ADJACENCY

```
layout (max_vertices = n) out;
```

Input Primitive Type	Size o
points	1
lines	2
triangles	3
lines_adjacency	4
triangles_adjacency	6

EmitVertex() & EndPrimitive()

- Tells the geometry shader that you've finished filling in all of the information for this vertex.
- Setting up the vertex works same vertex shader
  - You need to write into the built-in variable gl_Position
    - This sets the clip-space coordinates of the vertex that is produced by the geometry shader.
  - Any other attributes that you want to pass from the geometry shader to the fragment shader can be declared in an interface block or as global variables in the geometry shader.
  - You can call EmitVertex() as many times as you like
    - until you reach the limit you specified in your **max_vertices** layout qualifier.
  - Each time, you put new values into your output variables to generate a new vertex.
- Note:
  - EmitVertex() makes the values of any of your output variables (such as gl_Position) undefined
    - For example:
      - if you emit a triangle with a single color, you need to write that color with every one of your vertices
      - otherwise, you will end up with undefined results

- Indicates that you have finished appending vertices to the end of the primitive
  - Support the strip primitive types (**line_strip** and**triangle_strip**).
    - If your output primitive type is **triangle_strip** and you call EmitVertex() more than three times, the geometry shader will produce multiple triangles in a strip.
    - Likewise, if your output primitive type is **line_strip** and you call EmitVertex() more than twice, you'll get multiple lines.
- In the geometry shader, EndPrimitive() refers to the strip
  - If you want to draw individual lines or triangles, you have to call EndPrimitive() after every two or three vertices.
  - You can also draw multiple strips by calling EmitVertex() many times between multiple calls toEndPrimitive().
- Note:
  - EmitVertex() and EndPrimitive() in the geometry shader is that if you haven't produced enough vertices to produce a single primitive (e.g., you're generating**triangle_strip** outputs and you call EndPrimitive() after two vertices), nothing is produced for that primitive, and the vertices you've already produced are simply thrown away.

“Hello World” Geometry Shader

```
#version 330
#layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

smooth in vec4 theColorV[];
out vec4 theColor;

void main(void)
{
    int i;

   for (i = 0; i < gl_in.length(); i++)
   {
      gl_Position = gl_in[i].gl_Position
      theColor = theColorV[i];
      EmitVertex();
   }
   EndPrimitive();
}
```

2nd Task

- Implement simple geometry shader
  - Shift the vertices in geometry shader
  - Change color in geometry vertex

3rd Task

Explosions

- Create explosion of your models
  - Hints:
    - Calculate normal to each face in geometry shader

```
vec3 ab =
vec3 ac =
vec3 normal = normalize(cross(ab, ac));
```

    - Use an explosion factor passed from your program to move each face along the normal
    - Make sure faces are always the same size
    - Bonus: make something the explosion interesting (add colour decay)

Extras

Changing the Primitive Type in the Geometry Shader

- http://www.informit.com/articles/article.aspx?p=2120983&seqNum=2

- Bonus: Show normals with your figure
  - Hint: need to rander twice

- Submit your code as follows:

  submit 4431 lab5 filename(s)