CS 432 – Interactive Computer Graphics

Lecture 2 –Part 2

Introduction to Shaders

Shaders

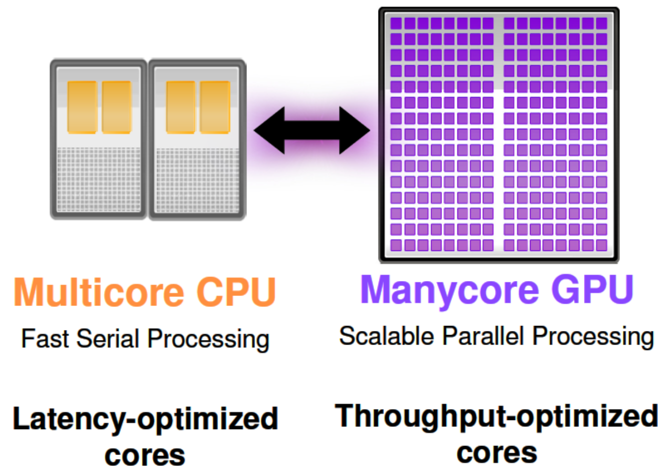
- To understand shaders, let's look at the graphics pipeline again
- The job of the client/CPU/OpenGL program is to have the logic to get the necessary vertex data to the GPU/server and provide instructions on what to do with it.

Shaders

- Now the GPU can take those instructions and do things like:
  - Move vertices
  - Color/shade vertices
  - Clip primitives
  - Assemble primitives
  - Project to 2D (rasterize)
  - Alter each fragment (pixel with depth info)

Shaders

- Why not just do this all in the OpenGL program?
- GPUs have hundreds if not thousands of specialized processing units.
  - So we can do a lot of stuff in parallel!
  - In particular, each vertex is processed independently as is each fragment.



Shaders

- Ok so how do we write these GPU shader programs?
- We'll use the OpenGL Shading Language (GLSL)
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Code sent to GPU/shaders as source code
- There are OpenGL functions to compile, link, and get information from shaders

The Programmable Pipeline

- Within the GPU pipeline, there are several places where we can provide custom functions to dictate what should happen.
- We'll focus on the two required stages: vertex shading and fragment (pixel) shading.

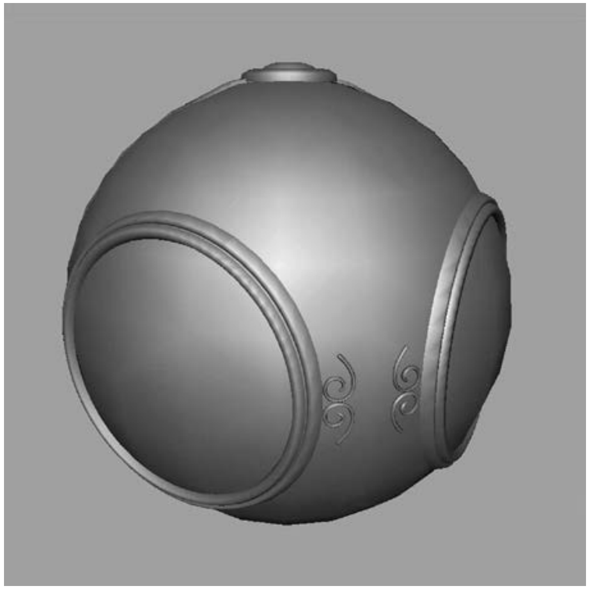
1. The *vertex shading stage*
    - This receives the vertex data that is in the vertex-buffer objects (VBO) and process each vertex separately.
2. The *Fragment shading stage*
    - Processes the fragments that come of out the rasterizer
    - Depth and color is also computed here

Vertex Shader Applications

- Moving Vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

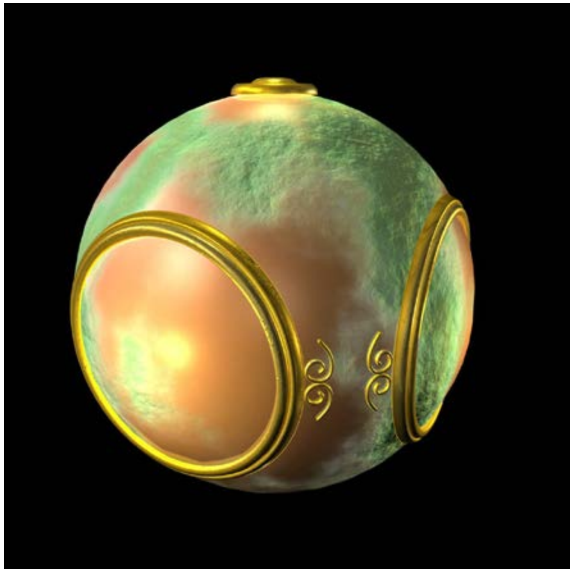
Fragment Shader Applications

- Texture Mapping



smooth shading

environment mapping



bump mapping

GLSL Data Types

- C types: `int, float, bool;`
- Vectors
  - `vec2, vec3, vec4`
- Matrices
  - `mat2, mat3, mat4`
  - Stored by columns (column major)
  - Standard referencing m[row][column]

GLSL Qualifiers

- In order to connect data between stages of the pipeline we need special qualifiers
  - `in` (from previous stage, or VBO if vertex shader)
  - `out` (out from this stage)
  - `uniform` (set directly by the client/OpenGL program)
  - `flat` - forces non-interpolation. Both the *out* variable of the previous stage and the matching *in* variable of the current stage must have this qualifier.
- In older version of GLSL (pre-1.3, these were called)
  - `attribute`
  - `varying`
  - `uniform`
- NOTE: In order for the GPU to connect parts, the `in/out` must have the same name.

Built-in GLSL Variables

- There are special **built-in** OpenGL state variables (don't need to declare them)
  - `gl_Color`
  - `gl_Position` (out from the vertex shader)
  - `gl_FragColor` (out from the fragment shader)
- However `gl_FragColor` is now considered depreciated and instead you should have an `out` qualified variable from the fragment stage to output the color of the fragment.

GLSL Code Format

- Start by stating the GLSL version
  - For Windows/Mac we'll use version 1.5
    - `#version 150`
  - On tux we'll use version 1.3
    - `#version 130`
- Specify the in/out/uniform variables
  - `in vec4 vPosition`
  - `uniform vec4 color`
  - `out vec3 normal`
- Do whatever you want in the main function
  - ```
    void main(){

    }
    ```
    ```

Basic GLSL Program

- Ok let’s look at a simplest GLSL program
- Vertex shader
  - Gets the vertex position from the VBO and set the default output variable `gl_Position` to it.
- Fragment shader
  - Allows the client program to set the fragment color to some value

Shader programs

Vertex Shader

```
#version 150

in vec4 vPosition;
void main(){

    gl_Position = vPosition;
}
```

Fragment Shader

```
#version 150

in vec4 color;
uniform vec4 color;
out vec4 FragColor;

void main(){
        FragColor = color;
}
```

Linking client application w/shader programs

- So the last step to have the client application:
- Initialize the shaders
  - Read in the source code of the shaders
  - Tell the GPU to compile/link them
- Whenever data changes (and at least once at the beginning) we need to tell the GPU where the data's coming from. We do that by:
  - Making the desired VAO and shader program active
    - So that subsequent calls affect the currently active shader program.
    - Assign values to any uniform variables
    - Specify where the data is in the current VBO for each attribute variable

Linking client application w/shader programs

- When drawing an object
  - Make sure the desired VAO is active
  - Make sure you are using the shader program you want (that it’s the active one)
  - Set any necessary `uniform` variables.
  - Tell the GPU to draw (via `glDrawArrays` etc..)

Linking client application w/shader programs

- How do we do all this!
- Initialize
  - We'll use Angel's `InitShader` function

```
GLuint program = InitShader("vshader.glsl", "fshader.glsl");
```

- Pass in uniform values and indicate where in the VBO attributes come from:

```
glBindVertexArray(VAO)

glUseProgram(program);  //make this program active
GLuint colorLoc =
       glGetUniformLocation(program, "color");

glUniform4fv(colorLoc, 1, vec4(0,0,1,1));
Gluint vPosLoc =
       glGetAttribLocation(program, "vposition");

glEnableVertexAttribArray(vPosLoc);
glVertexAttribPointer(vPosLoc, 2, GL_FLOAT, GL_FALSE,
       0, BUFFER_OFFSET(0));
```

- Yikes! Lets look at each of these closer….

Linking client application w/shader programs

- Just like how we need to make our VAOs and VBOs active when we want to change/use them, we need to do the same with shader programs:

  ```
  glUseProgram(program);
  ```

- We can use the `glGet*Location(Gluint,char*)` function to get the location of variables within the shader program.
  - For uniform variables:

    ```
    GLuint glGetUniformLocation(Gluint,char*)
    ```

  - For attribute (in) variables

    ```
    GLuint glGetAttribLocation(Gluint,char*)
    ```

- To pass a value into a uniform variable…

  ```
  glUniform4fv(GLuint, Gluint, void*);
  ```

Linking client application w/shader programs

- To tell the GPU where in the VBO to get the data for the attributes:
  - First enable the attribute:

    ```
    glEnableVertexAttribArray(vPosLoc);
    ```
  - Then tell it where the data is in the current VBO

    ```
    glVertexAttribPointer(GLuint, GLuint num,
            GL_type, GL_bool norm, GLuint stride
            Gluint offset));
    ```
    - Where `num` is how many of GL_type to take from the buffer for each vertex.
    - Where `GL_type` is the data type of the data in the buffer
    - Where `norm` is a boolean value (GL_TRUE, GL_FALSE) indicating if the data should be normalized (so that it's unit length) prior to use.
    - Where `stride` is how much data to skip between each piece of data
    - Where `offset` is where the data starts.
      - For simplicity we usually specify this relative to the address of the current buffer, `BUFFER_OFFSET(GLuint)`
- This information is now stored in the VAO (so it'll be loaded whenever we make the VAO active)
  - And we don't need to re-do it unless where the data is changes.

Example 1

```
//Mesh 0
GLuint VAO;
GLuint buffer;//VBO
GLuint color_loc;
GLuint program; //shader ID
const int NumVertices = 4;

// Vertices of a unit cube centered at origin, sides aligned with axes
vec2 points[4] = {
    vec2( 0.25, 0.25),
    vec2( 0.75, 0.25),
    vec2( 0.75, 0.75),
    vec2( 0.25, 0.75)
};

// RGBA colors
vec4 blue_opaque = vec4( 0.0, 0.0, 1.0, 1.0 );
```

Example 1: Init

```
// OpenGL initialization
void init()
{

    //Set up VAO
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    // Create and initialize a buffer object
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );
```

Get a VAO and make it the current one

Get a buffer name

Make buffer active (updates the VBO state in the current VAO)

Move data to the VBO

Example 1: Init (continued)

```
    // Load shaders and use the resulting shader program
    program = InitShader( "vshader00_v150.glsl", "fshader00_v150.glsl" );
    glUseProgram( program );

    // set up vertex arrays
    GLuint vPosition = glGetAttribLocation( program, "vPosition" );

    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );

    color_loc = glGetUniformLocation(program, "color");

    glClearColor( 1.0, 1.0, 1.0, 1.0 );
}
```

Initialize and make active a shader program

Get the `vPosition` variable from the shader program

Allow this attribute in the shader to pull from the VBO (state stored in VAO)

Tell the program to pull 2 floating point numbers for each vertex starting at the beginning of the VBO for use for the `vPosition` variable

Set the clear color to be white

Get the locations of the uniform color variable in the shader program

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );

    glBindVertexArray(VAO);
    glUseProgram(program);
    glUniform4fv(color_loc, 1, blue_opaque);
    glDrawArrays( GL_TRIANGLE_FAN, 0, NumVertices );

    glFlush();
}
```

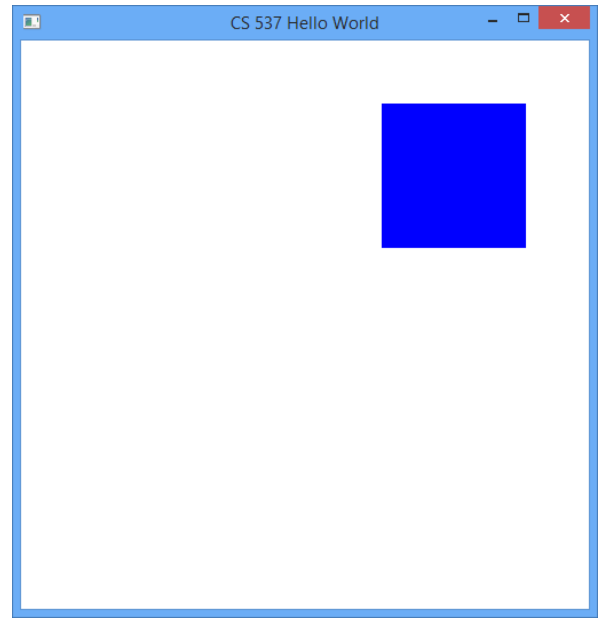
Every time we display:

- Clear the color buffer
- Draw the current buffer as a triangle fan using `NumVertices` vertices starting at 0

```
#version 150

in vec4 vPosition;

void main()
{
  gl_Position = vPosition;
}
```




```
#version 150

uniform   vec4  color;

out vec4 FragColor;

void main()

{

    FragColor = color;

}
```

Use the vertices to draw a triangle fan (first vertex acts as a hub)

Common performance issues

```
void
display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );

    glDrawArrays( GL_TRIANGLE_FAN, 0, NumVertices );

    glFlush();
}
```

```
void display(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBufferData(GL_ARRAY_BUFFER, sizeof(points),points,GL_STATIC_DRAW);
    glDrawArrays(GL_TRIANGLE_FAN, 0, NumVertices);

    glFlush();
}
```

```
void display(){
    glClear(GL_COLOR_BUFFER_BIT);
    for(int i = 0; i < NumVertices - 3; i++)
        glDrawArrays(GL_TRIANGLE_FAN, i, 3);

    glFlush();
}
```

Per-Vertex Color Example

- Instead of having the same color for every vertex and passing that to the GPU as a uniform variable, let's have for each vertex
  - A location
  - A color
- So now we need to have our VBO store both of these
  - And link our shader to them..

Per-Vertex Color Example

Notice the use of **glBufferSubData**

- Move the data onto GPU whenever necessary
  - glBindVertexArray(VAO);
  - glBindBuffer(GL_ARRAY_BUFFER,VBO);
  - glBufferData(GL_ARRAY_BUFFER, sizeof(points)+sizeof(colors), NULL, GL_STATIC_DRAW);
  - glBufferSubData(GL_ARRAY_BUFFER,0,sizeof(points),points);
  - glBufferSubData(GL_ARRAY_BUFFER,sizeof(points),sizeof(colors),colors);
- Get both the attributes from the shader and enable them
  - glBindVertexArray(VAO);
  - vPosition = glGetAttributeLocation(program, “vPosition”);
  - glEnableVertexAttribArray(vPosition);
  - cPosition = glGetAttributeLocation(program, “cPosition”);
  - glEnableVertexAttribArray(cPosition);
- Link the current buffer data to the attribute locations
  - glBindVertexArray(VAO);
  - glBindBuffer(GL_ARRAY_BUFFER,VBO);
  - glVertexAttribPointer(vPosition,2,GL_FLOAT,GL_FALSE,0,BUFFER_OFFSET(0));
  - glVertexAttribPointer(cPosition,4,GL_FLOAT,GL_FALSE,0,BUFFER_OFFSET(sizeof(points));
- Of course now we also need a color attribute in our shader!

Per-Vertex Color Shader

Vertex Shader

```
#version 150

in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main(){
        gl_Position = vPosition;
        color = vColor;
}
```

Fragment Shader

```
#version 150

in vec4 color;
out vec4 fColor;

void main(){
        fColor = color;
}
```

OpenGL Program

```
glBindVertexArray(VAO);
glUseProgram(program);
GLuint vPosition = glGetAttribLocation(program, “vPosition”);
glEnableVertexAttribArray(vPosition);
GLuint cPosition = glGetAttribLocation(program, “vColor”);
glEnableVertexAttribArray(cPosition);

/////
glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glVertexAttribPointer(cPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(size(points));
```

Modern OpenGL

Make the VAO active

```
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glUseProgram(programID);

        glBinderVertexArray (GL_ARRAY_BUFFER,VAO);

        glDrawArrays(GL_LINES, 0, 32);

        glFlush();

}
```

Tell the GPU to draw this data as lines, using locations 0 through 32 in the buffer

- Note we are not doing any calculations on the CPU nor moving any data!
- Only do these on the CPU if necessary

Putting it All Together

```
GLuint VBO, vPosition, color_loc;
vec4 color = vec4(0.0,0.0,1.0,1.0);
GLuint VAO;
GLuint program;

void init(){
   glClearColor(1,0,1.0,1.0,1.0);

   vec2 vertices[2];
   vertices[0] = vec2(0.0,0.0);
   vertices[1] = vec2(1.0,1.0);

   glGenVertexArrays(1,&VAO);

   GLuint VBO;
   glGenBuffers(1,&VBO);

   glBindBuffer(GL_ARRAY_BUFFER,VBO);
   glBufferData(GL_ARRAY_BUFFER,sizeof(vertices),vertices,GL_STATIC_DRAW)

   program = InitShader("vshader00_v150.glsl","fshader00_v150.glsl");
   glUseProgram(program);

   glBindVertexArray(VAO);
   glBindBuffer(GL_ARRAY_BUFFER_VBO);


   vPosition = glGetAttribLocation(program, "vPosition");

   glVertexAttribPointer(vPosition,2, GL_FLOAT,GL_FALSE,0,BUFFER_OFFSET(0));
   glEnableVertexAttribArray(vPosition);


   color_loc = glGetUniformLocation(program,"color");

}
```

```
void display(){

   glClear(GL_COLOR_BUFFER_BIT);

   glUseProgram(program);

   glBindVertexArray(VAO);
   glUniform4fv(color_loc,1,color);

   glDrawArrays(GL_LINES,0,2);


   glFlush();

}
```

Cleaning Up…

- While most OS/OpenGL/GPUs will automatically clean up stuff when the program exits, it may be good to do this yourself too
- Can use the `glutWMCloseFunc` callback
  - Freeglut has a newer version, `glutCloseFunc`
- Here we can release the data from our buffers

```
void onCloseWindow(){
   glDeleteBuffers(4,buffers);
}
```

Multiple Shaders

- Your programs will inevitably have to have multiple shaders
  - At least one for each different "way" you want to shade objects.
- For objects that use the same shader, every time you draw the object you'll most likely have to
  - Ensure necessary attributes are enabled.
  - Set the uniform variables as desired.
  - Make sure the attributes are pointing to the correct memory
    - Most likely you'll have to `glVertexAttribPointer`…
- And unfortunately the VAOs don't store the object's shader so when drawing an object you'll also want to make sure it's shader is the currently active one.