CMSC 335
COMPUTER GRAPHICS

LECTURE 5

• MANAGING DATA ON THE GPU
• VERTEX ARRAY AND BUFFER OBJECTS
• UNIFORM VARIABLES
We first need to learn how to get data into our programs
• Vertex data – split between invocations of each shader and data is interpolated through rasterization
• Uniform data – shared among all invocations of each shader
EXAMPLES

- Mesh data is an example of vertex data
- Light information is an example of uniform data
OVERVIEW OF VERTEX DATA

• Create Vertex Array Object
  • Create Vertex Buffer Object (and maybe Element Buffer Object)
  • Send data to the GPU
  • Specify how data is laid out to the GPU through attribute pointers

• Draw Vertex Array Object (as Array or Element Array)

• Will be explained out of order for a reason!
VERTEX BUFFER OBJECTS

• A **buffer** in computing typically refers to a place in memory that is used to temporarily store data while being moved from one place to another

• A **vertex buffer (VBO)** is the collection of vertex data that is sent to the GPU for processing

• **Vertex attributes** inform the GPU how the data is laid out in memory

• Vertex buffers can have different types, i.e., vertex data or element indices (to name two)
  • Indices would be provided for out of order primitive specification

• **Process:**
  • Request name of buffer from Open GL
  • Copy data to buffer
  • Specify vertex attributes
Request name of buffer from Open GL:

```c
GLuint vbo;
nGenBuffers(1, &vbo);
```
NOTE ON OPEN GL NAMING

• Open GL manages named objects that live on the GPU
  • Only so many can be created of each type
  • Names are always unique integers \(> 0\) (0 is reserved to be "null" object)

• Typically will request name from Open GL
COPY DATA TO THE BUFFER

• Open GL keeps track of which buffer is "active" or bound to the GPU. Only one object may be bound at a time

• So bind, copy data, then unbind:
  
  ```
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW)
  glBindBuffer(GL_ARRAY_BUFFER, 0);
  ```

Buffer "type" – e.g., GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER
Name of VBO
Number of bytes to copy and location (pointer) to initial byte

Dynamicity of data, i.e., does it change?
VERTEX ATTRIBUTE POINTERS

• Attribute pointers specify layout of the data, and you can specify as many as you want

• Data Layout Options
  • Batched
    
    \[ p_1, \ldots, p_n \quad n_1, \ldots, n_n \quad t_1, \ldots, t_n \]
  • Interleaved
    
    \[ p_1 \quad n_1 \quad t_1 \quad \ldots \quad \ldots \quad \ldots \quad p_n \quad n_n \quad t_n \]
  • Which is better?
• Specify layout per data attribute

```c
glBindBuffer(GL_ARRAY_BUFFER, vbo); // Or skip this if already bound
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
```

// Cleanup
```c
glDisableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

- **Index of attribute** – order/layout as specified in vertex shader.
- **Number of components to the attribute** (e.g., 3 for a vec3)
- **Type of each component of the attribute**
- **Stride**, i.e., distance in memory between successive attributes
- **Offset from beginning of buffer**
- **Should GL normalize or not?**
VERTEX ATTRIBUTE POINTERS

• Example:

\[
p_1 \quad n_1 \quad t_1 \quad \ldots \quad \ldots \quad \ldots \quad p_n \quad n_n \quad t_n
\]

\[\text{glVertexAttribPointer}(0, 3, \text{GL\_FLOAT}, \text{GL\_FALSE}, \text{sizeof(Vertex)}, 0)\];

• Write code for the normals and textures

• This is how we laid out the Obj model data!

Explained:
• 0 – first attribute
• 3 – points have 3 coordinates
• GL\_FLOAT – points are composed of floating-point numbers
• GL\_FALSE – points should not be normalized
• Sizeof(Vertex) – is the distance in memory between successive elements
• 0 – positions start off the memory, the start of the memory is the start of the first position
VERTEX ARRAY OBJECTS

• Vertex array objects collectively store all of this state – VBOs, EBOs, attributes, etc.
• Open GL requires us to work through this (and it makes drawing easier, trust me!)
• Create a name:
  GLuint vao;
  glGenVertexArrays(1, &vao); // Similar to before
• Bind:
  glBindVertexArray(vao);
• Don't forget to unbind after specifying all information
DRAWING VERTEX ARRAY OBJECTS

• Bind, draw, and unbind:

```c
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, vertices.size());
glBindVertexArray(0);
```
PRIMITIVE TYPES IN OPEN GL

• GL_POINTS – render vertices as points

• GL_LINES – render each successive pair as lines
PRIMITIVE TYPES IN OPEN GL

• **GL_LINE_STRIP** – render a line from each vertex to its previous vertex

• **GL_LINE_LOOP** – same as GL_LINE_STRIP but connects first and last vertex too

\[ \begin{align*}
  &v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6 \quad v_7 \quad v_8 \\
&v_1 \quad v_3 \quad v_5 \quad v_7 \quad v_8
\end{align*} \]
PRIMITIVE TYPES IN OPEN GL

• **GL_TRIANGLES** – render each successive triplet as a triangle

• **GL_TRIANGLE_STRIP** – render a triangle between each vertex and its previous two vertices
PRIMITIVE TYPES IN OPEN GL

• **GL_TRIANGLES_FAN** – render a triangle between each vertex, its previous vertex, and the first vertex

• **GL_QUADS, GL_QUAD_STRIP** are similar

• Other options exist for specialized cases and use with specific shader stages
Keep in mind

- All output data from the vertex shader is interpolated and sent to the fragment shader
**UNIFORMS**

- Uniform data is *shared* among all invocations of shader stages.
- Simply need to learn how to send data to the program.
  - Request name of uniform inside of the shader program.
  - Send data to the GPU.
- Then in shaders, specify a global variable with the keyword `uniform`.
  ```glsl
  uniform mat4 proj_matrix;
  ```
UNIFORMS

• To request name:
  
  GLuint loc = glGetUniformLocation(programName, "variable_name");

• To send data:

  glUniformX(loc, ...);

  where X is the specific method, size, and type specification and ... represents the required parameters. Example:

  glUniformMatrix4fv(loc, 1, GL_FALSE, &proj_matrix);