CS 432 – Interactive Computer Graphics

Lecture 2 –Part 1

Primitives and Buffers

Rendering in OpenGL

- Ok, so now we want to actually draw stuff!
- OpenGL (like most rendering APIs) is a *state machine* that can render a limited set of *primitives*
- The main idea:
  - The client program copies data to the GPU
  - The client program then calls a sequence of
    - Change some states
    - Request some drawing using some data on the GPU

Rendering in OpenGL

- So we need to know:
  - Available states to set
  - *How* to copy data to GPU
  - *What* we need to store on GPU in order to draw.
  - Calls to draw

Rendering in OpenGL

- All of our objects will be specified via *vertices*
- Vertices can have attributes like:
  - Vertex location
  - Vertex color
  - Vertex normal (for 3D graphics)
- When it comes time to draw an object we then need to tell the GPU where to get the data from
  - I.e. where each attribute data is located.
  - As we'll see in a moment, the data will be stored in things called *vertex buffers*
- To make things easier, the state information containing which buffers are used and where each attribute is stored within that buffer can be saved in something called a *vertex array object* (VAO).
  - So let's start there!

Vertex Array Objects

- First thing's first, let's create a VAO to store this state information!
- To do this we request an ID from the GPU:

```
    GLuint abuffer;
    glGenVertexArrays(1, &abuffer);
```

- Once we've created a VAO we can make it the active one with the command:

```
        glBindVertexArray(abuffer);
```

Vertex Array Objects

- Once a VAO is active:
  - Any drawing requests will use its stored state information to determine what buffer(s) is being used, and where the attributes are within it.
  - Any new assignment of vertex attribute related states (changes of buffers, changes to locations within buffers) will be stored/updated in this active VAO
- So let’s start creating data and associating it with the current VAO!

Vertices

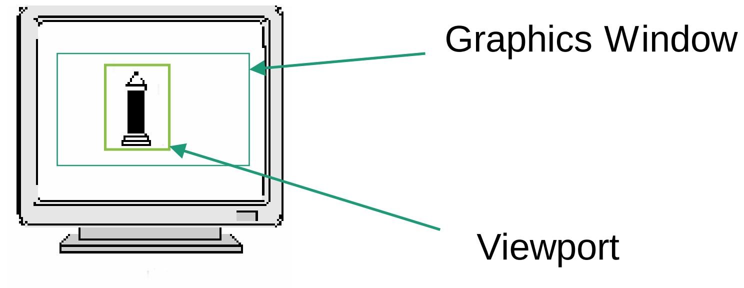
- Therefore, the first thing we need to do when creating an object is to specify the vertices.
- Lets use the data types provided in Angel's `vec.h` file:
  - `vec2, vec3, vec4`
- If we're doing a 2D application we can just specify a 2D vertex using a `vec2` object (although we'll see later that we may want a 3rd coordinate even for 2D graphics)
  - `vec2 point = vec2(0,0);  //x,y`
- We can also use this data type to specify colors
  - `vec4 blue = vec4(0,0,1,1);  //r,g,b,a`

Vertex Locations

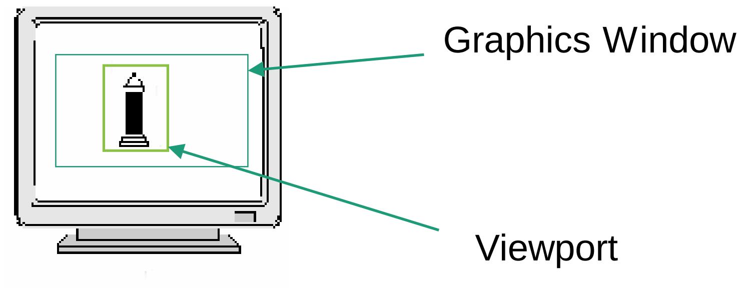
- In this course we'll talk about various coordinate systems.
- The first two we'll talk about are the *camera coordinate system* and the *window coordinate system.*
- The *window coordinate system* refers to screen pixels.
  - These go from (0,0) to (width, height) as specified in our `glutInitWindowSize(width,height)` function
- The *camera coordinates* go from (-1,-1) to (1,1)
- OpenGL automatically maps the camera coordinates to the window coordinates such that (-1,-1)→(0,0) and (1,1)→(width,height)

Vertex Locations

- So the camera coordinate system is a square.
- What happens if the window is not a square?
  - Stretching
- We may want to force our OpenGL program to render to an area (square?) inside of the window so there is no stretching.
- This area is called the *viewport*.



Vertex Locations



- We can set the viewport using the `glViewport` function

```
glViewport(GLint x, GLint y, GLsizei w, GLsizei h);
```

- And whenever the window is reshaped, we can call this function.
  - In the main assign a callback function:

```
glutReshapeFunc(reshape);
```

  - The callback function will get passed the current window’s width and height which we can then use to set the viewport:

```
void reshape(GLsizei w, GLsizei h){
    GLsizei m = w;
    if (h<w)
        m = h;
    glViewport(0,0,m,m);
}
```

Vertex Arrays

- Typically we'll store *sets* of vertices pertaining to objects in arrays.
- Depending on if your sets need to grow or not, you could use static or dynamic arrays:
  - `vec2 points[4];`
  - `vec2 *points = new vec2[4];`
  - `vector<vec2> points;`
    - You can easily get the data form this via `&points[0]`
- Remember to delete your pointers when you're done with them!
  - Often this data can be deleted once it has been moved to the GPU

Moving Data to the GPU

- Ok so now we have our vertices as data on the client application (CPU).
- Next step is to move that data to the GPU
- Memory on the graphics card that are used to store vertex information are called *vertex buffers*.
- To process to move vertices onto a vertex buffer are as follows:
    1. Allocate a *vertex buffer object (VBO)* for the data
    2. Create the data (in our CPU-bound client application)
    3. Make sure the desired VAO is active.
    4. Bind the VBO to the current VAO
    5. Move the data to the active VBO

VBOs

- Start by getting a valid buffer object name(s)

```
GLuint buffer;
glGenBuffers(1, &buffer);
```

- We'll probably want to make sure we are using the VAO we want, so:

```
glBindVertexArray(abuffer);
```

- Next we'll bind the VBO to the VAO:

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

- And finally we'll copy the data from the client application to the buffer that's currently bound to the VBO

```
glBufferData(GL_ARRAY_BUFFER, sizeof(points),
        points, GL_DRAWHINT);
```

  - Where the most common types of GL_DRAWHINT are
    - `GL_STATIC_DRAW` - Optimized if data won't change often
    - `GL_DYNAMIC_DRAW` - Optimized if data will change often

Buffer Size

- Note:
  - If our array is static, we can just use
    - `size(data)`
  - If our array is dynamic, we must keep track of how many elements are in it and the data type. Given we have *n* elements each of which are a `vec2`, we need to move:
    - `n*size(vec2)`

Moving Data to the GPU

```
GLuint VBO, VAO;


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


    glGenVertexArray(&VAO);
    glBindVertexArray(VAO);


    glGenBuffers(1,&VBO);
    glBindBuffer(GL_ARRAY_BUFFER,VBO);

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


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

}
```

Create a VAO to bundle together the object's attributes

Get a buffer

Bind this buffer to the VAO's array buffer so it's the "active" buffer

Copy data to the active buffer

Create data

Sub-Buffers

- Sometimes we only want to replace some of the data in a buffer
  - Maybe only some over the vertices changed.
  - Maybe we're storing both position and color in the same buffer and only want to change colors.
- To do this we can use the `glBufferSubData` function:

```
glBufferSubData(GLenum target, GLint offset,
                GLsizei size, void* data);
```

Deleting Buffers

- Since our graphics should be high-performing, memory management is a big deal.
- Therefore we should have a way to delete memory from the GPU when we're done with it.

```
glDeleteBuffers(GLsizei n, const Gluint* ids);
```

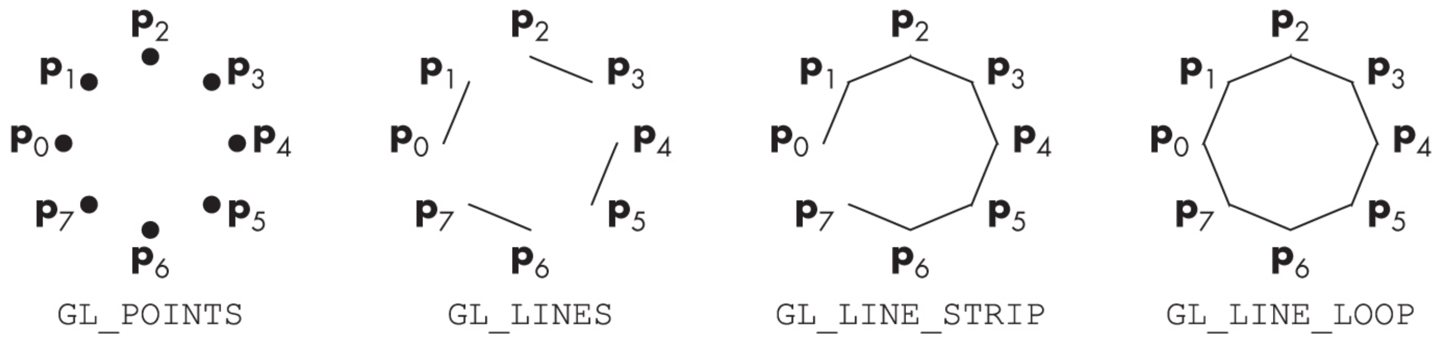
Primitives

- Ok so now we have vertex locations stored on the graphics card.
- How does it know how to *assemble* these into objects?
- Modern OpenGL allows for assembling vertices into variations of *primitives*:
  - Points
  - Lines
  - Triangles
- Any other polygons should be converted into a set of triangles.
  - Though technically most OpenGL still allow for quadrilaterals and polygons

Point and Line Primitives

- Line Segments are the basic graphical entity
  - Used to approximate curves
- We can choose how to render the vertices as lines:
  - `GL_POINTS` – Each vertex is displayed at a size of at least one pixel
  - `GL_LINES` – Successive pairs of vertices are interpreted as endpoints of individual segments
  - `GL_LINE_STRIP` – Successive vertices act as points in a continuing line strip
  - `GL_LINE_LOOP` – Same as `GL_LINE_STRIP`, but the last vertex connects to the first vertex. This creates an polygon

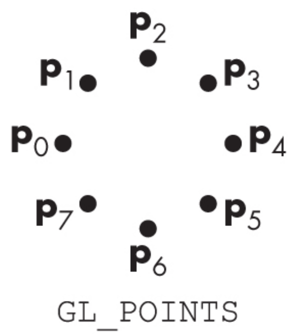
Line Primitives

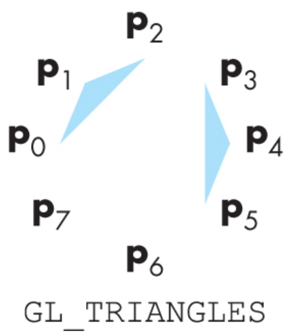


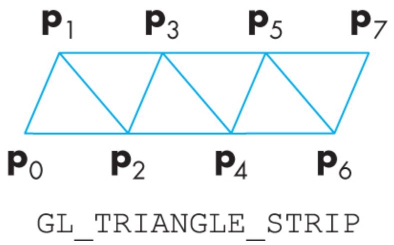
Triangle Primitives

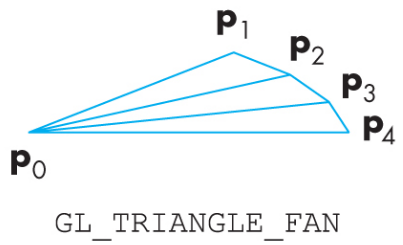
- We can also choose different ways to render triangles using vertices:
  - `GL_TRIANGLES` – Same idea as lines, but each successive group of three vertices specifies a new triangle
  - `GL_TRIANGLE_STRIP` – Similar to line-strip, but each vertex is combined with the previous 2 to create a new triangle.
  - `GL_TRIANGLE_FAN` – The first point is the first vertex for all triangles. The first triangle also uses the next to vertices. Subsequent triangles just use the last used vertex and one more one.
    - This can be used to render a polygon as a set of triangles (albeit naively)

Triangles


p_2
p_1
p_3
p_0
p_4
p_7
p_5
p_6
GL_POINTS


p_2
p_1
p_3
p_0
p_4
p_7
p_5
p_6
GL_TRIANGLES


p_1
p_3
p_5
p_7
p_0
p_2
p_4
p_6
GL_TRIANGLE_STRIP


p_1
p_2
p_3
p_4
p_0
GL_TRIANGLE_FAN

Rendering States

- Some useful states that we can change include:
  - `glPointSize(float)`
  - `glPolygonMode(face,mode)`
    - Where face can be
      - `GL_FRONT_AND_BACK`
      - `GL_FRONT`
      - `GL_BACK`
    - And mode can be
      - `GL_FILL`
      - `GL_LINE`
      - `GL_POINT`

Rendering Objects

- Now that we know what the different states and assembling methods are, and since we already have vertex data on the GPU we're ready to draw!
- To *render* the object (in the display function) you:
  1. Make its vertex array object active
     - With `glBindVertexArray(VAO);`
     - This will switch all attribute information (like the VBO, where in the VBO attributes come from, etc..) to the states stored in the VAO.
  2. Make active the desired GPU program
  3. Set any OpenGL state variables you want to be applied to this object
     - This may include `uniform` variables in the shader program.
  4. Tell the GPU to draw the data by specifying
     - How (the primitive types)
     - What (starting location in buffer and how many elements)

Modern OpenGL

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

                glBinderVertexArray (VAO);
                glUseProgram(program);

                glDrawArrays(GL_LINES, 0, 32);

                glFlush();
}
```

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

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

Shaders

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

        glBinderVertexArray (VAO);
        glUseProgram(program);

        glDrawArrays(GL_LINES, 0, 32);

        glFlush();
}
```

- Of course this may raise the questions..
  - How do the GPU know where the color information is? Or any other information about the vertices for that matter….
- There is one more piece to the puzzle!
  - The GPU code!!!
- We call these small programs, *shaders*….