Vertex Buffer Objects

CGT 520

OpenGL objects

- OpenGL objects aren't like C++ objects
- They don't have member functions, they are simply represented as unsigned integer identifiers
    - Created using glGen*() functions
- You use an object by binding the object to a target
    - glBind*(target, id)
- Once bound you can change the state of the object or use it for rendering

Vertex Buffer Object motivation

- Loops like this can be a major bottleneck:

```
for(int n=0; n < nVertices; n++)
{
    glVertex3f(X[n], Y[n], Z[n]);
}
```

- Function call overhead
  - One call per vertex
- Too much data traffic when X,Y,Z do not change.
  - It would be nice to keep static data in video memory
- This is why glBegin/glVertex/glEnd is obsolete
    - You can write backward compatible OpenGL...
    - But don't

The idea of VBOs

- Create storage for vertex data on graphics server (GPU)
  - Only send data once for static data
  - Can hold vertex attributes
    - Vertex coordinates, texture coordinates, color, normals
- Render that data with a single function call
- Similar to other OpenGL objects which reside on server (such as textures, shaders)
  - Refer to VBOs by ID
  - Functions modify currently bound VBO

The idea of VBOs

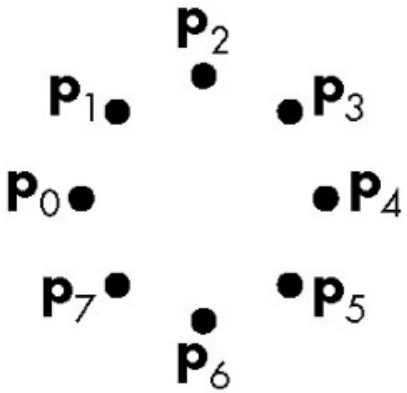
- VBOs are a very flexible feature and there are many different ways to put data into them and get data out of them
- In these notes we will simplify things by assuming only a single attribute (vertex position) is in the VBO
- More complex scenarios are covered in CGT 521 (multiple attributes, interleaved/noninterleaved attribs, indexed/nonindexed vertices)

VBO steps for static data

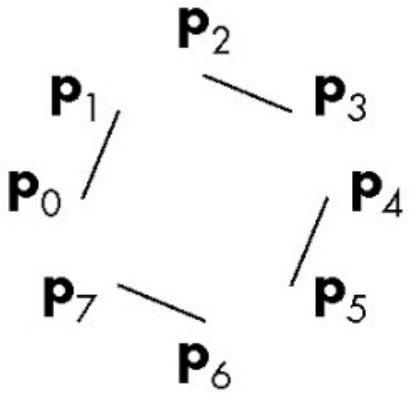
- Initialize VBO
  1. Get a new Buffer ID
  2. Bind ID to target (GL_ARRAY_BUFFER)
  3. Fill buffer with data
     - Client → server transfer
- Use VBO
  1. Bind ID to target
  2. Enable VBO attribute (glEnableVertexAttribArray)
  3. Set pointers (where each attribute is located in the buffer)
  4. Draw buffer contents

OpenGL primitives

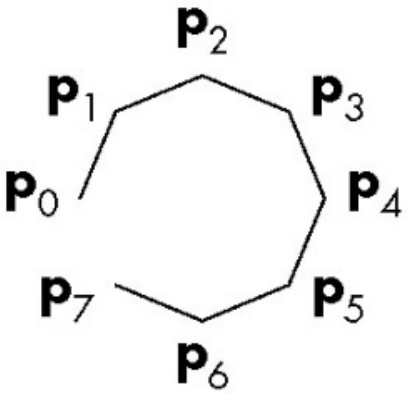
- Order of vertices in VBO depends on primitive type



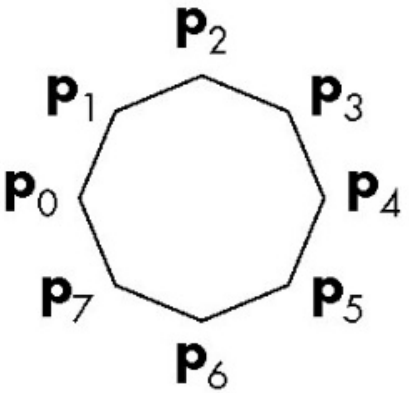
GL_POINTS



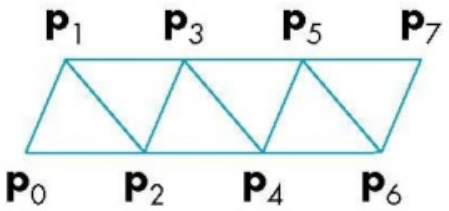
GL_LINES



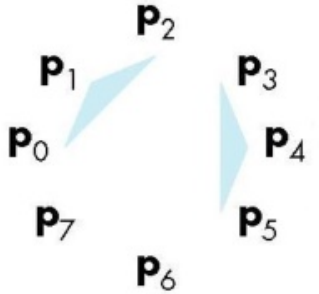
GL_LINE_STRIP



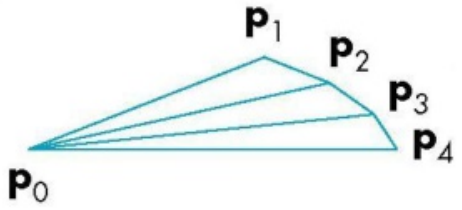
GL_LINE_LOOP



GL_TRIANGLE_STRIP



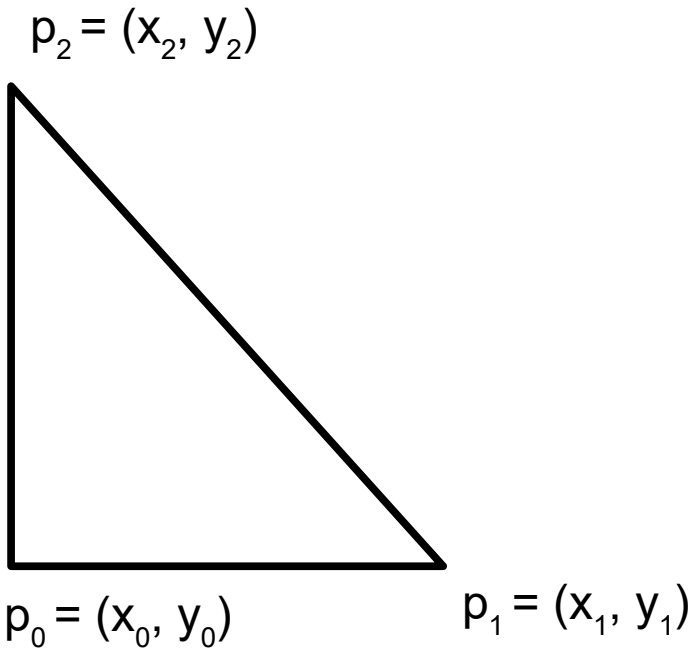
GL_TRIANGLES



GL_TRIANGLE_FAN

Representing primitives

- One GL_TRIANGLE



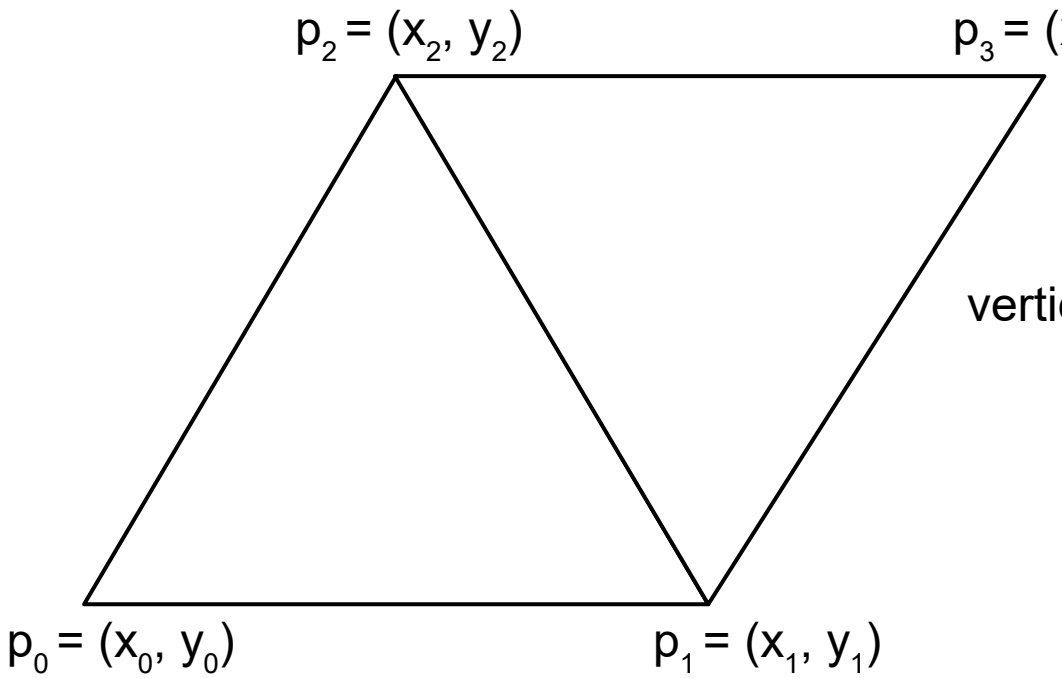
VBO will include this data:

vertices = $[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2]$

An array like this can be copied into a VBO to represent a triangle:

```
const float triangle_verts[] =
     {-1.0f,-1.0f, 0.0f,
      1.0f, -1.0f, 0.0f,
      1.0f, 1.0f, 0.0f };
```

Representing primitives



Two GL_TRIANGLES

vertices = [x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2,
x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3]

One GL_TRIANGLE_STRIP

vertices = [x_0, y_0, z_0, x_1, y_1, z_1,
x_2, y_2, z_2, x_3, y_3, z_3]

Keep in mind the vertex ordering for the primitive (e.g. fans, strips)

VBO Creation

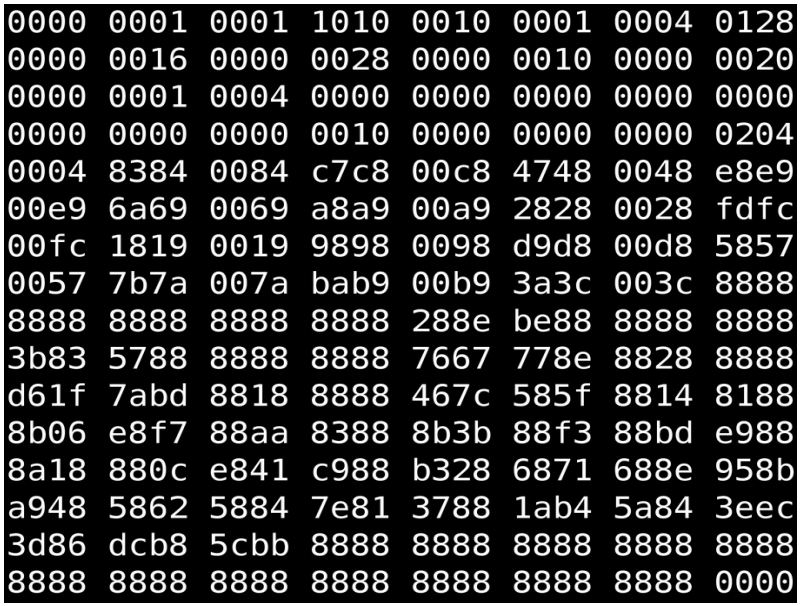
Example: Loading a single triangle into a VBO

```
GLuint init_vbo()
{
   float triangle_verts[] =
      {  0.0f, 0.0f, 0.0f,
         1.0f, 0.0f, 0.0f,
         0.0f, 1.0f, 0.0f};

   GLuint vbo;
   glGenBuffers(1, &vbo); // Generate vbo to hold vertex attributes for triangle

   //binding vbo means that subsequent glBufferdata calls will load data into this object
   glBindBuffer(GL_ARRAY_BUFFER, vbo);

   //upload from main memory to gpu memory
   glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_verts), &triangle_verts[0], GL_STATIC_DRAW);

   return vbo;
}
```

VBO setup

- To OpenGL the VBO is just a chunk of memory until we tell it how to interpret the memory.

What are the data types?
Where are vertex coordinates?
Are they 2D or 3D or 4D?
Are there other vertex attributes?
How are they laid out?
What are the names in the shader?



These are the questions we need to answer in order to use VBOs.

VBO Drawing

Example: Drawing the triangle from VBO

```
glUseProgram(shader_program); //enable the shader we want to use
glBindBuffer(GL_ARRAY_BUFFER, vbo); //specify the buffer where vertex attribute data is stored

//get a reference to an attribute variable name in a shader
GLint pos_loc = glGetAttribLocation(shader_program, "pos_attrib");

glEnableVertexAttribArray(pos_loc); //enable this attribute

//tell opengl how to get the attribute values out of the vbo
glVertexAttribPointer(pos_loc, 3, GL_FLOAT, false, 0, 0);

//Draw 3 vertices from the VBO as GL_TRIANGLES, starting from vertex 0
glDrawArrays(GL_TRIANGLES, 0, 3);
```

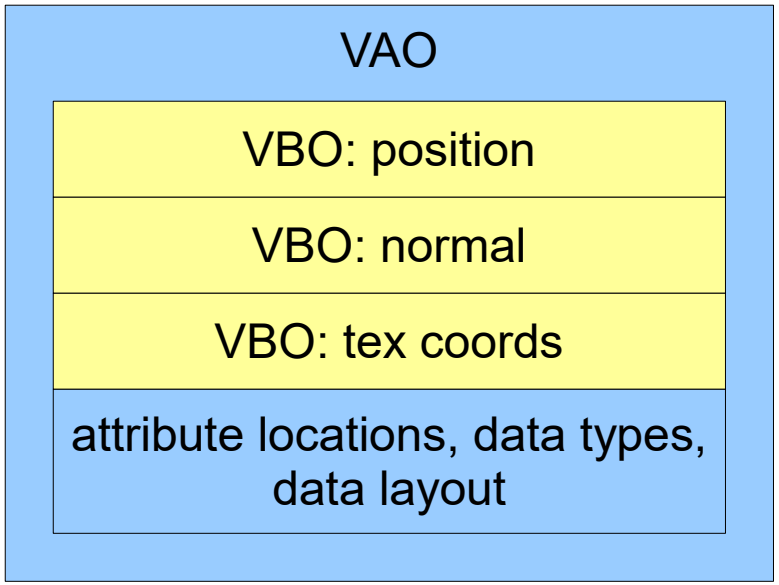
Example: vertex shader

```
#version 400
in vec3 pos_attrib;
 void main(void)
{
    gl_Position = vec4(pos_attrib, 1.0);
}
```

Problem: Drawing with VBO takes too many function calls.
Solution: Vertex Array Objects (VAO)

Vertex Array Objects

- Vertex Array Objects (VAOs) are like wrappers for VBOs
- They simplify the process of drawing from VBOs by eliminating the need to respecify many parameters when drawing



VAO + VBO steps for static data

- Initialize
  1. Get a new VAO ID, Bind VAO
  2. Get a new VBO ID
  3. Bind VBO ID to target (GL_ARRAY_BUFFER)
  4. Fill buffer with data

     Client →server transfer
  5. Enable VBO attribute (glEnableVertexAttribArray)
  6. Set pointers (where each attribute is in the buffer)
- Use
  1. Bind **VAO** ID to target
  2. Draw buffer contents

VAO Creation

Example: Loading a single attribute into a VBO using VAO
VAO is like a wrapper object that stores some state that the VBO doesn't.

```
void init_vao()
{
    GLuint vao, vbo;
    float triangle_verts[] = { 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f};

    //generate vao id to hold the mapping from attrib variables in shader to memory locations in vbo
    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vbo); // Generate vbo to hold vertex attributes for triangle


    //binding vao means that bindbuffer, enablevertexattribarray and vertexattribpointer
    // state will be remembered by vao
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo); //specify the buffer where vertex attribute data is stored
    //upload from main memory to gpu memory
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_verts), &triangle_verts[0], GL_STATIC_DRAW);

    //get a reference to an attrib variable name in a shader
    GLint pos_loc = glGetAttribLocation(shader_program, "pos_attrib");

    glEnableVertexAttribArray(pos_loc); //enable this attribute

    //tell opengl how to get the attribute values out of the vbo (stride and offset)
    glVertexAttribPointer(pos_loc, 3, GL_FLOAT, false, 0, 0);
    glBindVertexArray(0); //unbind the vao

    return vao;
}
```

Drawing the VAO

Example: Drawing the triangle from VAO

```
glUseProgram(shader_program);

glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Example: vertex shader

```
#version 400
in vec3 pos_attrib;
 void main(void)
{
    gl_Position = vec4(pos_attrib, 1.0);
}
```

VAO Creation

- Get a new VAO ID
  - Can get an array of n VAO IDs
  - `glGenVertexArrays(GLsizei n, GLuint* arrayIDs)`
- Bind ID (there is no target for VAO)
  - `glBindVertexArray(GLuint vao);`

VBO Creation

- Get a new Buffer ID
  - Can get an array of n buffer IDs
  - `glGenBuffers(GLsizei n, GLuint* bufferIDs)`
- Bind ID to target
  - `glBindBuffer(ARRAY_BUFFER, id);` For attribute values
- Fill buffer with data (2 options)
    - Send data from an array in client memory
      - `glBufferData(target, size, data, usage);`

VBO creation

- Fill buffer with data (2 options)
  - Send data from an array in client memory
    - `glBufferData(target, size, data, usage);`
  - Get a pointer into server memory and write there
    - `void* glMapBuffer(target, access);`
    - We won't use this one in CGT 520

For all functions taking `target` as an argument, use the same target as when the buffer was first bound :
ARRAY_BUFFER for vertex attributes

There are other targets we will discuss in CGT 521

Buffering Data

- Send data from an array in client memory
  - `glBufferData(target, size, data, usage);`
    - size : number of **bytes**
    - data: pointer to data
      - Passing 0 or NULL allocates uninitialized storage
    - usage: more about this next...
  - Can also write a sub-block of the buffer
    - `glBufferSubData(...)`

VBO Usage

- These are hints that determine which memory (system, PCIe, video) is used
- Based on frequency of access (update-to-draw)
  - Static:1-to-many
  - Dynamic: many-to-many
  - Stream: 1-to-1
- Nature of access
  - Draw: written by application, read by GPU
  - Read: written by GPU, read by application
  - Copy: written by GPU, read by GPU
- Possible values are: GL_STREAM_DRAW, GL_DYNAMIC_COPY, etc...

VBO Usage

- Most common cases
  - Loading and drawing a mesh that never changes: **GL_STATIC_DRAW**
  - Animating particles on the CPU and rebuffering them every frame: **GL_STREAM_DRAW**

Connection to shaders

- The vertex shader will declare variables that correspond to attributes
    - `in vec4 pos; // position attribute`
- Attributes have locations, just like uniforms
    - `glGetAttribLocation(...)`
- The VAO can have multiple attributes, but for now we consider only one

Overview: Connection with shader

```
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(...);
```

```
GLint pos_loc = glGetAttribLocation(shader_program, "pos_attrib");
glEnableVertexAttribArray(pos_loc);
glVertexAttribPointer(pos_loc, 4, GL_FLOAT, false, 0, 0);
```

Associating the VAO with shader attribute variables

Setting Attribute Info

- `glVertexAttribPointer(index, size, type, normalized, stride, *pointer);`

  - **`index:`** which attribute location? (Mesh will often have multiple attributes)
  - **`size`**: number of components (=3 for vec3, =4 for vec4, ...)
  - **`type`**: data type (= GL_FLOAT for vec4, vec3 …)
  - **`normalized:`** `GL_TRUE, GL_FALSE (usually GL_FALSE)`
    - Fixed-point types can be converted to [0,1] range for unsigned types or [-1,1] range for signed types.
    - This is **not** vector normalization

  `stride` and `pointer` specify locations of attributes...
  Use 0, 0 for now (since we only have one attribute in the buffer)

Drawing from the VAO

1. Bind ID to target
    - `glBindVertexArray(...)`
2. Draw buffer contents
    - `glDrawArrays(...)`

Drawing Buffer Contents

- `glDrawArrays(mode, first, count)`
  - Draw `count` vertices, starting with `first`.
  - `mode`: is primitive type
    - `GL_POINTS, GL_TRIANGLES, etc.`

- `glMultiDrawArrays(mode, *first, *count, primcount)`
  - Specify several ranges within the VBO

Wrapping it up

- Use `glDeleteBuffers` to free memory at runtime
- Check for errors along the way (glGetError())
  - INVALID_ENUM
    - Bad enumerated parameter for usage, access, etc.
  - INVALID_VALUE
    - Negative size, count, etc.
  - INVALID_OPERATION
    - Cannot Unmap buffer that is not currently mapped, etc
  - OUT_OF_MEMORY
    - Buffer too large...
    - Cannot map buffer...