**EECS 487: Interactive Computer Graphics**

Lecture 17:
- Vertex Buffer Object (VBO)
- Vertex-Array Object (VAO)

---

**Vertex Buffer Object (VBO)**

Vertex array must be copied on every draw

*VBO allows data to be stored in GPU memory but still client read/write-able*

Steps to set up and use a VBO:
1. Generate and bind VBO descriptor(s)
   ```
   glGenBuffers(), glBindBuffer()
   ```
2. Allocate space for VBO and populate it with vertex attribute(s) data
   ```
   glBufferData(), glBufferSubData()
   ```
3. Tell the GPU which attribute is where on the VBO
   ```
   glEnableClientState(), glVertexPointer()
   ```

---

**Vertex Array**

- **GL_VERTEX_ARRAY**
- **GL_COLOR_ARRAY**

App/Client Memory

```
color[]
```

Graphics Card/Server Memory

```
vertices[]
```

- Vertex attribute data read from client memory to server memory whenever `glDraw*()` is called
- See `va.cpp`

---

**Buffer Object Descriptor**

First, generate buffer object descriptor(s)*:

```
int vbods[N];
glGenBuffers(GLsizei N, GLuint *vbods);
```

for example, if only 1 buffer object is needed:

```
int vbo;
glGenBuffers(1, &vbo);
```

*descriptor == handle == name == id
**Current Buffer Object**

Next bind each descriptor to a type of buffer

```c
glBindBuffer(GLenum target, GLuint vbod);
// target: GL_ARRAY_BUFFER,
// GL_ELEMENT_ARRAY_BUFFER (for indices)
```

for example:

```c
glBindBuffer(GL_ARRAY_BUFFER, vbod);
```

This also makes the vbod buffer object the “current/in-use” buffer object: all subsequent buffer object operations apply to this buffer object

If vbod is NULL, disable Buffer Object

---

**Buffer Object Data**

Allocated (and initialize) space for buffer object:

```c
glBufferData(GLenum target, GLsizei size, GLvoid* data, GLenum usage);
// target: as before, size: size of data,
// data: pointer to source data array, NULL to allocate
// memory without content initialization
// usage: will data be modified? Helps OpenGL determine
// memory location to optimize performance
```

for example:

```c
float vertices[3][2] =
{ {0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0} };
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);
```
Vertex Buffer Object

App/Client Memory

- vbod
- vbo
- object holders

system bus/network

Graphics Card/Server Memory

- vertices[]
- color[]

OpenGL state
- glGenBuffers()
- glBindBuffer()
- glBufferData()

Vertex attribute data copied from client memory to server memory using
- glBufferData() or
- glBufferSubData()

When glDraw*() is called, data is already resident

Multiple Vertex Attributes

If there are multiple vertex attributes, we can put them back to back in the buffer object using:

- glBufferSubData(GLenum target, GLint offset, GLsizei size, GLvoid* data);
  // target: as before, offset: byte offset from the
  // start of buffer as copy target, size: size of data,
  // data: pointer to source data array

To avoid having two copies of data, client can share server memory using glMapBuffer() / glUnmapBuffer() after the call to glBufferData() (see Pixel Buffer Object (PBO) later)

Multiple Vertex Attributes Example

```c
float vertices[3][2] = 
{ {0.0, 0.0}, {0.0, 1.0}, {1.0, 0.0} };
float color[3][3]= { {1.0, 0.0, 0.0},
{0.0, 1.0, 0.0}, {0.0, 0.0, 1.0} };
glBufferData(GL_ARRAY_BUFFER,
  sizeof(vertices)+sizeof(color), NULL,
  GL_STATIC_DRAW);  // allocate space only
glBufferSubData(GL_ARRAY_BUFFER, 0,
  sizeof(vertices), vertices); // copy data
glBufferSubData(GL_ARRAY_BUFFER,
  sizeof(vertices), sizeof(color), color); // copy data
```

Vertex Buffer Object

App/Client Memory

- vbod
- vbo
- object holders

system bus/network

Graphics Card/Server Memory

- vertices[]
- color[]

OpenGL state
- glGenBuffers()
- glBindBuffer()
- glBufferData()
- glBufferSubData()

Vertex attribute data copied from client memory to server memory using
- glBufferData() or
- glBufferSubData()

When glDraw*() is called, data is already resident
Common Error

`glGenBuffers()` not called before `glBindBuffer()`

Offset is given in count, not in bytes, e.g., if the first array (at offset 0) is of N elements of class Vertex, the correct offset to the second array is: \( N \times \text{sizeof(Vertex)} \), not just N

If offset is wrong, especially when using `glMapBuffer`, GPU can lock up and system must be rebooted

---

Vertex Buffer Object Use

All we have done so far is copied the `vertices[]` and `color[]` arrays to a buffer object

The graphics pipeline has no idea of the existence of the buffer object nor what is contained in it

We need to:
1. tell the graphics pipeline what is in the buffer object
2. pass the buffer object to the graphics pipeline

---

Vertex Buffer Object Use

To tell the graphics pipeline that we're passing it an array of vertices, as with vertex array, enable `GL_VERTEX_ARRAY` in client memory:

- `glEnableClientState(GL_VERTEX_ARRAY)`

Next, pass the array, specifying the data format used (different from vertex array, the last argument is a byte offset from the start of buffer)

- `glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *offset);`
- // size: # coordinates per vertex, type: data type,
- // stride=0 means coordinates are back to back,
- // offset: byte offset from the start of buffer object

Finally, draw as with vertex array

---

Vertex Buffer Object Use Example

```c
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(2, GL_FLOAT, 0, 0);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, sizeof(vertices));
glDrawArrays(GL_TRIANGLES, 0, 3);
```

With vertex buffer object, client-side data array (`vertices[]` and `color[]`) do not need to be in scope when `glDraw*()` is called
Vertex Buffer Object

OpenGL Object In General

OpenGL is a state machine

An OpenGL object is a container that encapsulates a particular set of states

The set of states an OpenGL object contain depends on the context it is bound to: the states that an object contains are mapped into and become the context’s states, e.g.,
glBindBuffer(GL_ARRAY_BUFFER, vbod);

To read more:
http://www.opengl.org/wiki/OpenGL_Objects
http://www.opengl.org/wiki/Buffer_Objects
http://www.opengl.org/wiki/Vertex_Buffer_Object

Various OpenGL Objects

Buffer Objects: unformatted linear memory
• Vertex Buffer Objects
• Pixel Buffer Objects
• Uniform Buffer Objects

Vertex Array Objects: encapsulate sets of vertex array states including vbo bindings

Texture Objects: an OpenGL object that contains one or more images with the same image format

Framebuffer Objects
• Renderbuffer Objects

OpenGL Objects Management

All OpenGL objects are managed the same way:
• glGen*()
• glBind*()
• use
• glDelete*()
• as soon as object usage is done, always call glBind*() with object handle 0 to unbind object and call glDelete*() to delete object [I’ll assume you know this already and won’t be mentioning it again]
Drawing Multiple Shapes

To draw different shapes, e.g., cube, sphere, cone, etc. multiple times, want to store the vertex attributes of each shape once ONLY, with re-use.

Even then, with VBO alone, must make several API calls to change OpenGL state for each shape, each draw:

```c
glBindBuffer(GL_ARRAY_BUFFER, vbods[i]);
glVertexPointer(2, GL_FLOAT, 0, vertex_offset);
glColorPointer(3, GL_FLOAT, 0, color_offset);
```

Vertex-Array Object

**Vertex-Array Object (VAO)** encapsulates vertex attribute states
- format, type, stride, storage info, and
- attached vertex buffer object if vbo is used

Enables rapid switching between sets of vertex array states and vbo bindings with a single API call, changing one OpenGL state:

```c
glBindVertexArray(vaods[i]);
```

Requires OpenGL 3.0+

Vertex-Array Object

As with other OpenGL objects, first generate vertex-array object descriptor(s)/handle(s):

```c
GLuint vaods[N];
glGenVertexArrays(GLsizei N, GLuint *vaods);
```

For example, if only 2 vertex-array objects are needed:

```c
GLuint vaods[2];
glGenVertexArrays(2, vaods);
```

Next bind and make current each descriptor:

```c
glBindVertexArray(GLuint vaods[int i]);
```

If vaods is NULL, disable Vertex-Array Object

Vertex Array Object without VBO

VAO w/o VBO is supported by Mac OS X, but not by all other graphics card drivers
Setup Client-Side Vertex Array

For current/bound vertex-array object, populate VAO by enabling client-side access:

```cpp
glEnableClientState(GL_VERTEX_ARRAY);
```

then specify vertex array location:

```cpp
glVertexPointer(2, GL_FLOAT, 0, vertices);
```

Both client state and vertex array location above are set within the context of a VAO

Then draw:

```cpp
glDrawArrays(GL_TRIANGLES, 0, 3);
```

VAO w/o VBO is supported by Mac OS X, but not by all other graphics card drivers

VAOs Switching Example

```cpp
float cone[N][3], ccolors[N][3];
float cube[M][3], bcolors[M][3];

// set up: assuming vaods[] have been generated
glBindVertexArray(vaods[CONE]);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, cone);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, ccolors);

// use: cone[], ccolors[], cube[], and bcolors[] must be in scope
glBindVertexArray(vaods[CUBE]);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, cube);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, bcolors);

// draw:
glBindVertexArray(vaods[CONE]);
glDrawArrays(GL_TRIANGLE_STRIP, 0, N);

See vao-va.cpp
```
VAO with VBO
First, Setup VBO

float cone[N][3], ccolors[N][3];
float cube[M][3], bcolors[M][3];
enum {CONE=0, CUBE, NSHAPES};

GLuint vbods[NSHAPES];
glGenBuffers(NSHAPES, vbods);

// set up the two vertex buffer objects
glBindBuffer(GL_ARRAY_BUFFER, vbods[CONE]);
glBufferData(GL_ARRAY_BUFFER, sizeof(cone)+sizeof(ccolors),
0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(cone), cone);
glBufferSubData(GL_ARRAY_BUFFER,
sizeof(cone), sizeof(ccolors), ccolors);

glBindBuffer(GL_ARRAY_BUFFER, vbods[CUBE]);
glBufferData(GL_ARRAY_BUFFER, sizeof(cube)+sizeof(bcolors),
0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(cube), cube);
glBufferSubData(GL_ARRAY_BUFFER,
sizeof(cube), sizeof(bcolors), bcolors);

Next, Setup VAOs

GLuint vaods[NSHAPES];
glGenVertexArrays(NSHAPES, vaods);

// set up the two vertex-array objects
glBindVertexArray(vaods[CONE]);
glBindBuffer(GL_ARRAY_BUFFER, vbods[CONE]);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, 0 /* buffer byte offset */);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, sizeof(cone) /* byte offset */);

glBindVertexArray(vaods[CUBE]);
glBindBuffer(GL_ARRAY_BUFFER, vbods[CONE]);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, 0 /* buffer byte offset */);
};
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, 0, sizeof(cube) /* byte offset */);
Finally, Use VAOs to Switch Shapes

```cpp
// use, same as client-side, but cone[], ccolors[],
// cube[], and bcolors[] don't need to be in scope
glBindVertexArray(vaos[CONE]);
glDrawArrays(GL_TRIANGLE_STRIP, 0, N);

glBindVertexArray(vaos[CUBE]);
glDrawArrays(GL_TRIANGLE_STRIP, 0, M);

glBindVertexArray(vaos[CONE]);
glDrawArrays(GL_TRIANGLE_STRIP, 0, N);
```

See `vao-vbo.cpp`

Next: VBO and VAO with shaders