Vertex Arrays

Professor Raymond Zavodnik

Fachhochschule Regensburg, Germany, May 2, 2013

Vertex Arrays

- Geometric Primitives have been deprecated by OpenGL
- Calls to `glVertex*()`, `glColor*()` are too expensive for applications
- Data is repetitive
- Consider a cube consisting of 6 quads
  - Each quad requires 4 vertices
  - Thus 24 points required
  - But there are only 8 different vertices!!

There Must Be A Better Way

- Put the eight points into a one-dimensional array with 24 floating point entries
- Do the same for colors, normals, textures to be used in rendering
- Access them by dereferencing pointers to this data
- Number of function calls goes way down
- If each vertex also had a normal vector associated with it, these can likewise be put into (parallel)arrays
- Or even interleaved for even more speed of access
- You also gain by having no redundancy

Steps in Creating Vertex Arrays

1. Define your arrays, mostly as global arrays
2. Then, instead of listing the raw data using `glVertex*()`, etc. rewrite your `redraw()` function to access the arrays.
3. Define pointers to your arrays
4. Dereference the elements

An Example

- Consider our 2D triangle with vertices $(-0.9, -0.0), (0.9, -0.9), (0.0, 0.9)$
- Use the three floating point colors red = $(1.0, 0.0, 0.0)$, green = $(0.0, 1.0, 0.0)$ and blue = $(00.0, 0.0, 1.0)$
- For access use a three element array

An Example: Write the Data

- `const GLfloat vertices[] = {-0.9,-0.9,0.9,-0.9,0.0,0.9}`
- `const GLfloat colors[] = {1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,1.0}`
- `indices[] = {0,1,2}`

Enable The Arrays

- `glEnableClientState(GL_VERTEX_ARRAY);`
- `glEnableClientState(GL_COLOR_ARRAY);`
- This activates both arrays
- You might need more, e.g. Normal Vectors and Textures
- IMPORTANT: Vertex Arrays are stored client side, so they are not stored in display lists

Accessing the Vertex Array Elements

- Use pointers
- Here you need to specify size and type of each coordinate
- There is an optional `stride` entry if it is desired to interleave the arrays into one
- Finally you specify a pointer to your data

Accessing the Vertex Array elements

- `glVertexPointer(2, GL_FLOAT, 0, vertices);`
- `glColorPointer(3, GL_FLOAT,0,colors);`

Draw by Dereferencing

- If you want to jump around in the array, use `glArrayElement()`
- To access sequentially use `glDrawElements()`
- Specify the type of primitive, the number, the data type, and the index array
- Thus `glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, indices);`

Don’t Forget to Disable

- You may not need the data, e.g. in certain sections of your program
- `glDisableClientState(GL_VERTEX_ARRAY);`
- `glDisableClientState(GL_COLOR_ARRAY);`
- That’s it!!

Buffer Objects

- Vertex Arrays eliminate immediate mode
- **BUT** they cause too many round-trips between client and server
- Use `Buffer Objects` to store vertices and associated data on the graphics server
- Can also use Buffer Objects to store pixel data (less important)
- Steps to utilize
  1. Create the Buffer Object
  2. Activate it
  3. Transfer the data from user to server

Create the Buffer Object

- Associate an integer as a handle to the Buffer Object
- OpenGL can do this automatically
- Use `glGenBuffers(GLSizei n, GLuint *buffers)`
- Check if a specific handle is in active use call `glIsBuffer()` with the integer in question as parameter

Activate a Specific Buffer

- In general you can have many buffer handles
- Call to initialize the buffer and its data in order to define it
- Thereafter it can be deactivated at will to use selectively when its data is required
- For example, when rendering is done with just that data
- Use `glBindBuffer(GLenum target, GLuint buffer)`
- `target` is usually `GL_ARRAY_BUFFER` to be used with Vertex Arrays
- Stop usimg Buffer by binding to a buffer value of zero

Getting Data into Buffer Objects

- Start by reserving space on the server
- Copy the data from the client's memory into the buffer object
- Can load data later by passing `NULL`
- Use `glBufferData()` with the following parameters
    1. Specify a target, usually `GL_ARRAY_BUFFER` for vertex data
    2. Specify a buffer size in bytes
    3. Specify a pointer to the user data (or `NULL`)
    4. Specify how to read and write after the transfer, e.g. `GL_STATIC_DRAW`

Using Buffer Objects with Vertex Arrays

- You still need to tell OpenGL where to find your vertex data in the buffer
- Render using vertex-array rendering functions, such as `glDrawArrays()` or `glDrawElements()`
- Steps to do:
  1. Obtain a buffer handle
  2. Bind the buffer object
  3. Obtain server-side storage
  4. Specify offsets in the server data with `glVertexPointer()`. This is different for shaders
  5. Render

Define Buffer Objects

- Initialize buffers, in this case vertex data

```
unsigned int handle[3];
glGenBuffers(3, handle);
```

- Bind to Vertices given by v (use GL_ARRAY_ELEMENT for index arrays)

```
    glBindBuffer(GL_ARRAY_BUFFER, handle[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(v), v,
                 GL_STATIC_DRAW);
```

- Define a pointer to this data on the server:

```
     glVertexPointer(3, GL_FLOAT, 0,
                 (GLvoid *)((char *)NULL) );
```

- Do this for each buffer needed: colors, normals, texture, indices

Using Buffer Objects

```
glEnableClientState(GL_VERTEX_ARRAY);
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE,
                        (GLvoid *)((char *)NULL + (0));
glDisableClientState(GL_VERTEX_ARRAY);
```

- In `draw()` routine, for example, when drawing quads:
- Enable for each attribute
- Notice fake pointer to server-side storage

Vertex Array Objects

- It is desirable to switch between groups of vertex data
- Therefore, encapsulate vertex array state, such as data and buffers, into an object, call it, a *Vertex Array Object*
- Proceed analogously to Buffer definitions
  1. Generate the handle to the Vertex Array Object with `glGenVertex Arrays`
  2. Bind to an object (if previously created) or make a new one with `glBindVertexArray()`
  3. See Shader lecture–only important for vertex data in shaders

Define Vertex Array Objects

- Define Buffer Objects as above
- *Before* first definition define a handle for each attribute in the VAO:

```
GLuint vaoHandle[ATTRIBS];
glGenVertexArrays(ATTRIB, &vaoHandle);
glBindVertexArray(vaoHandle);
```

- Do this for each Vertex, Normal, Color buffer required
- Issue a `glBindVertexArray(0);` to deactivate the VAO until use

Use Vertex Array Objects

- In your `redraw()` routine bind, draw and unbind (if needed)

```
glBindVertexArray(vaoHandle);
glDrawElements(GL_QUADS, 24,
               GL_UNSIGNED_BYTE,
                (GLvoid *)((char *)NULL + (0)
glBindVertexArray(0);
```

- This technique allows turning large amounts of data off and on for rendering
- Issue a `glBindVertexArray(0);` to deactivate the VAO until the next use