Vertex Arrays and Vertex Buffer Objects

Mike Bailey

Oregon State University

The OpenGL Client-Server Model
The Difference Between Vertex Arrays and Vertex Buffer Objects

• Both vertex arrays and vertex buffers do the same thing, so functionally they are the same.

• **Vertex Arrays** live on the host (the “client”).

• **Vertex Buffers** live on the graphics card (the “server”).

Vertex Arrays: The Big Idea

• Store vertex coordinates and vertex attributes in arrays on the host (client).

• Every time you want to draw, transmit the arrays to the graphics card (server), along with indices that tell what vertex numbers need to be connected.

• This way, each vertex only needs to be transformed once.

• It also results in fewer overall function calls (to glVertex3f, for example).
Vertex Arrays: Cube Example

GLfloat CubeVertices[][3] =
{
{ -1., -1., -1. },
{  1., -1., -1. },
{ -1.,  1., -1. },
{  1.,  1., -1. },
{ -1., -1.,  1. },
{  1., -1.,  1. },
{ -1.,  1.,  1. },
{  1.,  1.,  1. }
};

GLfloat CubeColors[][3] =
{
{ 0., 0., 0. },
{ 1., 0., 0. },
{ 0., 1., 0. },
{ 1., 1., 0. },
{ 0., 0., 1. },
{ 1., 0., 1. },
{ 0., 1., 1. },
{ 1., 1., 1. }
};

GLuint CubeIndices[][4] =
{
{ 0, 2, 3, 1 },
{ 4, 5, 7, 6 },
{ 1, 3, 7, 5 },
{ 0, 4, 6, 2 },
{ 2, 6, 7, 3 },
{ 0, 1, 5, 4 }
};

Vertex Arrays: Step #1 – Fill the Arrays

GLfloat Vertices[][3] =
{
{ 1., 2., 3. },
{ 4., 5., 6. },
...
};
Vertex Arrays: Step #2 – Activate the Array Types That You Will Use

```c
void glEnableClientState(int type);
```

where `type` can be any of:

- GL_VERTEX_ARRAY
- GL_COLOR_ARRAY
- GL_NORMAL_ARRAY
- GL_SECONDARY_COLOR_ARRAY
- GL_TEXTURE_COORD_ARRAY

- Call this as many times as you need to enable all the arrays that you will need.
- There are other types, too.
- To deactivate a type, call:

```c
void glDisableClientState(int type);
```

Vertex Arrays: Step #3 – Specify the Data

```c
glVertexPointer(int size, int type, int stride, int array);
glColorPointer(int size, int type, int stride, int array);
glNormalPointer(int type, int stride, int array);
glSecondaryColorPointer(int size, int type, int stride, int array);
glTexCoordPointer(int size, int type, int stride, int array);
```

- `size` is the spatial dimension, and can be: 2, 3, or 4
- `type` can be:
  - GL_SHORT
  - GL_INT
  - GL_FLOAT
  - GL_DOUBLE
  - GL_DOUBLE

- `stride` is the byte offset between consecutive entries in the array (0 means tightly packed)
- `array` is the name of the corresponding data array
Vertex Arrays: Step #4 – Specify the Connections

List the vertices individually:

```c
glBegin( GL_TRIANGLES );
glArrayElement( 0 );
glArrayElement( 1 );
glArrayElement( 2 );
glArrayElement( 0 );
glArrayElement( 3 );
glArrayElement( 4 );
glEnd( );
```

Or, list the vertices explicitly:

```c
GLuint TriIndices[ ][3] =
{ 0, 1, 2 },
{ 0, 3, 4 };
glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_INT, TriIndices );
```

Vertex Arrays: Cube Example

```c
GLfloat CubeVertices[ ][3] =
{ -1., -1., -1. },
{ 1., -1., -1. },
{ -1., 1., -1. },
{ 1., 1., -1. },
{ -1., -1., 1. },
{ 1., -1., 1. },
{ -1., 1., 1. },
{ 1., 1., 1. }
};

GLfloat CubeColors[ ][3] =
{ 0., 0., 0. },
{ 1., 0., 0. },
{ 0., 1., 0. },
{ 1., 1., 0. },
{ 0., 0., 1. },
{ 1., 0., 1. },
{ 0., 1., 1. },
{ 1., 1., 1. };

GLuint CubeIndices[ ][4] =
{ 0, 2, 3, 1 },
{ 4, 5, 7, 6 },
{ 1, 3, 7, 5 },
{ 0, 4, 6, 2 },
{ 2, 6, 7, 3 },
{ 0, 1, 5, 4 };
```
Vertex Arrays: Cube Example

```c
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
glVertexPointer( 3, GL_FLOAT, 0, CubeVertices );
gColorPointer( 3, GL_FLOAT, 0, CubeColors );
gBegin( GL_QUADS );
glArrayElement( 0 );
glArrayElement( 2 );
glArrayElement( 3 );
glArrayElement( 1 );
glArrayElement( 4 );
glArrayElement( 5 );
glArrayElement( 7 );
glArrayElement( 6 );
glArrayElement( 1 );
glArrayElement( 3 );
glArrayElement( 7 );
glArrayElement( 5 );
glArrayElement( 0 );
glArrayElement( 4 );
glArrayElement( 6 );
glArrayElement( 2 );
glArrayElement( 2 );
glArrayElement( 6 );
glArrayElement( 7 );
glArrayElement( 3 );
glArrayElement( 0 );
glArrayElement( 1 );
glArrayElement( 5 );
glArrayElement( 4 );
glEnd();
```
Vertex Arrays:
Cube Example – `glDrawElements()` call

```c
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

glVertexPointer( 3, GL_FLOAT, 0, CubeVertices );
gColorPointer( 3, GL_FLOAT, 0, CubeColors );
glDrawElements( GL_QUADS, 24, GL_UNSIGNED_INT, CubeIndices );
```

Vertex Buffers: The Big Idea

- Store vertex coordinates and vertex attributes in arrays on the **graphics card** (server).
- Optionally store the connections on the graphics card too.
- Every time you want to draw, the vertex arrays are already on the graphics card, possibly along with indices that tell what vertex numbers need to be connected. If the indices are not already there, send them.
**A Little Background -- the OpenGL Rendering Context**

The OpenGL Rendering Context contains all the characteristic information necessary to produce an image from geometry. This includes transformations, colors, lighting, textures, where to send the display, etc.

Some of these characteristics have a default value (e.g., lines are white, the display goes to the screen) and some have nothing (e.g., no textures exist)

---

**More Background – What is an OpenGL “Object”?**

An OpenGL Object is pretty much the same as a C++, C#, or Java object: it encapsulates a group of data items and allows you to treat them as a single whole. For example, a Vertex Buffer Object could be defined in C++ by:

```cpp
class VertexBufferObject
{
    enum dataType;
    void *memStart;
    int memSize;
};
```

Then, you could create any number of Vertex Buffer Object instances, each with its own characteristics encapsulated within it. When you want to make that combination current, you just need to bring in (“bind”) that entire object. When you bind an object, all of its information comes with it.
More Background – How do you Create an OpenGL “Object”?

In C++, objects are pointed to by their address.

In OpenGL, objects are pointed to by an unsigned integer handle. You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:

```c
GLuint bufA;
glGenBuffers( 1, &bufA );
```

This doesn’t actually allocate memory for the buffer object yet, it just acquires a unique handle. To allocate memory, you need to bind this handle to the Context.

More Background – “Binding” to the Context

The OpenGL term “binding” refers to “attaching” or “docking” (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will “flow” through the Context into the object.

```c
glBindBuffer( bufA, GL_ARRAY_BUFFER );
glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
```
More Background -- “Binding” to the Context

When you want to use that Vertex Buffer Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again.

![Vertex Buffer Object Diagram]

```gl
glBindBuffer( bufA, GL_ARRAY_BUFFER );
```

Vertex Buffers: Putting Data in the Buffer Object

```gl
glBufferData( type, numBytes, data, usage );
```

- `type` is the type of buffer object this is:
  - `GL_ARRAY_BUFFER` to store floating point vertices, normals, colors, and texture coordinates
  - `GL_ELEMENT_ARRAY_BUFFER` to store integer vertex indices to connect for drawing

- `numBytes` is the number of bytes to store in all. Not the number of numbers, but the number of bytes!

- `data` is the memory address of (i.e., pointer to) the data to be transferred to the graphics card. This can be NULL, and the data can be transferred later.
**Vertex Buffers: Putting Data in the Buffer Object**

`glBufferData(type, numbytes, data, usage);`

*usage* is a hint as to how the data will be used: `GL_xxx_yyy`

where xxx can be:
- **STREAM**: this buffer will be written lots
- **STATIC**: this buffer will be written seldom and read seldom
- **DYNAMIC**: this buffer will be written often and used often

and yyy can be:
- **DRAW**: this buffer will be used for drawing
- **READ**: this buffer will be copied into
- **COPY**: not a real need for now, but someday…

---

**Vertex Buffers: A Choice of Terminology**

The architects of OpenGL had a choice at this point. They could let vertex buffer objects use the same terminology as vertex arrays, or they could invent new terminology.

They decided to re-use the same terminology to make conversion from vertex arrays to vertex buffers that much easier. Don’t take the use of the word *Client* seriously here!
Vertex Buffers: Step #1 – Fill the Arrays

```c
GLfloat Vertices[][3] =
{
    { 1., 2., 3. },
    { 4., 5., 6. },
    ...
};
```

Vertex Buffers: Step #2 – Create the Buffers and Fill Them

```c
glGenBuffers( 1, &bufA );

glBindBuffer( bufA, GL_ARRAY_BUFFER );

glBufferData( GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, Vertices, GL_DYNAMIC_DRAW );
```
**Vertex Buffers: Step #3 – Activate the Array Types That You Will Use**

```c
glEnableClientState( type )
```

where `type` can be any of:

- GL_VERTEX_ARRAY
- GL_COLOR_ARRAY
- GL_NORMAL_ARRAY
- GL_SECONDARY_COLOR_ARRAY
- GL_TEXTURE_COORD_ARRAY

- Call this as many times as you need to enable all the arrays that you will need.
- There are other types, too.
- To deactivate a type, call:

```c
glDisableClientState( type )
```

**Vertex Buffers: Step #4 – To Draw, Bind the Buffers**

```c
glBindBuffer( bufA, GL_ARRAY_BUFFER );
glBindBuffer( bufB, GL_ELEMENT_ARRAY_BUFFER );
```
Vertex Buffers: Step #5 – Specify the Data

- glVertexPointer( size, type, stride, offset);
- glColorPointer( size, type, stride, offset);
- glNormalPointer( type, stride, offset);
- glSecondaryColorPointer( size, type, stride, offset);
- glTexCoordPointer( size, type, stride, offset);

**size** is the spatial dimension, and can be: 2, 3, or 4

**type** can be:
- GL_SHORT
- GL_INT
- GL_FLOAT
- GL_DOUBLE

**stride** is the byte offset between consecutive entries in the array (0 means tightly packed)

**offset**, the 4th argument, is no longer an array memory location. It is the byte offset from the start of the data array buffer where the first element of this part of the data lives.

Vertex Buffers: Step #6 – Specify the Connections

**List the vertices individually:**

```c
glBegin( GL_TRIANGLES );
glArrayElement( 0 );
glArrayElement( 1 );
glArrayElement( 2 );
glArrayElement( 0 );
glArrayElement( 3 );
glArrayElement( 4 );
glEnd();
```

**List the vertices together:**

```c
GLuint TriIndices[ ][3] =
{   { 0, 1, 2 },
    { 0, 3, 4 }
};
glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_INT, TriIndices );
```
static GLfloat CubeVertices[][3] =
{
    { -1., -1., -1. },
    {  1., -1., -1. },
    { -1.,  1., -1. },
    {  1.,  1., -1. },
    { -1., -1.,  1. },
    {  1., -1.,  1. },
    { -1.,  1.,  1. },
    {  1.,  1.,  1. }
};

static GLfloat CubeColors[][3] =
{
    { 0., 0., 0. },
    { 1., 0., 0. },
    { 0., 1., 0. },
    { 1., 1., 0. },
    { 0., 0., 1. },
    { 1., 0., 1. },
    { 0., 1., 1. },
    { 1., 1., 1. }
};

static GLuint CubeIndices[][4] =
{
    { 0, 2, 3, 1 },
    { 4, 5, 7, 6 },
    { 1, 3, 7, 5 },
    { 0, 4, 6, 2 },
    { 2, 6, 7, 3 },
    { 0, 1, 5, 4 }
};
```

*Vertex Buffers:*

**Cube Example – glVertexElement() calls**

```
Vertex Buffers: Re-writing Data into a Vertex Buffer

```c
float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER, usage );
```

`usage` is a hint as to how the data will be used:

- `GL_READ_ONLY`  the vertex data will be read from, but not written to
- `GL_WRITE`     the vertex data will be written to
- `GL_READ_WRITE` the vertex data will be read from and written to

You can now use `vertexArray[]` like any other floating-point array.

When you are done, be sure to call:

```c
glUnMapBuffer( GL_ARRAY_BUFFER );
```