Introduction To OpenGL

How OpenGL Works

- OpenGL uses a series of matrices to control the position and way primitives are drawn
- OpenGL 1.x - 2.x allows these primitives to be drawn in two ways
  - immediate mode
  - retained mode

Immediate Mode

- In immediate mode the vertices to be processed are sent to the GPU (Graphics Card) one at a time.
- This means there is a bottleneck between CPU and GPU whilst data is sent
- This method is easy to use but not very efficient
- Immediate mode has become depricated as part of OpenGL 3.x and is not available in OpenGL ES

Retained Mode

- In this mode the data to be drawn is sent to the GPU and retained in graphics memory.
- A pointer to this object is returned to the user level program and this object reference is called when drawing etc.
- This requires a little more setup but does improve the Frame Rate considerably.
- For most of the examples we will use this mode.
- This is also the way DirectX works

GLSL

- OpenGL Shading Language is a high level shading language based on C
- It allows us to program directly on the GPU and can be used for Shading calculations for vertex and fragment based processing.
- We will look at this in more detail later but to use the full OpenGL 3.x functionality we need to also create a GLSL shader and process the vertices on the GPU
- Using GLSL we can also manipulate the actual vertices of the system.

Immediate Mode

- OpenGL provides tools for drawing many different primitives, including Points, Lines Quads etc.
- Most of them are described by one or more vertices.
- In OpenGL we describe a list of vertices by using the `glBegin()` and `glEnd()` functions.
- The argument to the `glBegin()` function determines how the vertices passed will be drawn.
- We will not be using this mode and most examples on the web will use this

OpenGL Profiles

- The current OpenGL version is 4.1, it has full compatibility with OpenGL ES 2.0 for embedded devices
- There are two profiles associated with this version
  - Core Profile
  - Compatibility profile

OpenGL 4.1 Core

- The Core profile is very new and many GPU's do not fully support this
- This profile doesn't contain any of the immediate mode GL elements and has no backward compatibility
- Will not be fully available in drivers / GPU cores for a while

OpenGL 4 Compatibility

- The compatibility profile contains lots of OpenGL immediate mode elements as well as earlier GLSL structures and components
- This means legacy code will still work with the drivers but may not be optimal
- At present my code base is using about 5% of this profile and the aim is to move all code away from this
- At present we are going to use a hybrid as the transition process continues
- I will attempt to point out the rationale for each of these decisions as we go

Compatibility Profile

- The main problem with the compatibility profile is there are no clear guidelines on the implementation
- Different vendors behave differently
- Usually the whole system will fall back to the “software implementation” (not on the GPU)

Programmers view of OpenGL

- To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.
- A typical program that uses OpenGL begins with calls to open a window into the framebuffer. (in our case using Qt)
- Once a GL context is allocated, the programmer is free to issue OpenGL commands.

Programmers view of OpenGL

- Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons)
- Others affect the rendering of these primitives including how they are lit or coloured and how they are mapped from the user’s two- or three-dimensional model space to the two-dimensional screen.
- There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

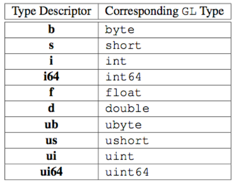
Client Server Model

- The model for interpretation of OpenGL commands is client-server.
- That is, a program (the client) issues commands, and these commands are interpreted and processed by the OpenGL (the server).
- The server may or may not operate on the same computer as the client.
- In this sense, the GL is “network-transparent.”
- A server may maintain a number of OpenGL contexts, each of which is an encapsulation of current GL state.

GL Command Syntax

- GL commands are functions or procedures.
- Various groups of commands perform the same operation but differ in how arguments are supplied to them.
- To specify the type of parameter GL uses a specific syntax
- GL commands are formed from a name which may be followed, depending on the particular command, by a sequence of characters describing a parameter to the command.
- If present, a digit indicates the required length (number of values) of the indicated type.
- Next, a string of characters making up one of the type descriptors

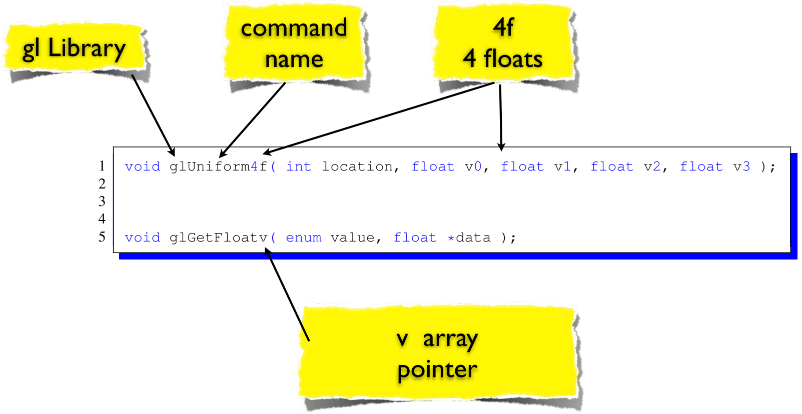
GL Command Syntax



Type Descriptor	Corresponding GL Type
b	byte
s	short
i	int
i64	int64
f	float
d	double
ub	ubyte
us	ushort
ui	uint
ui64	uint64

- A final v character, if present, indicates that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments.

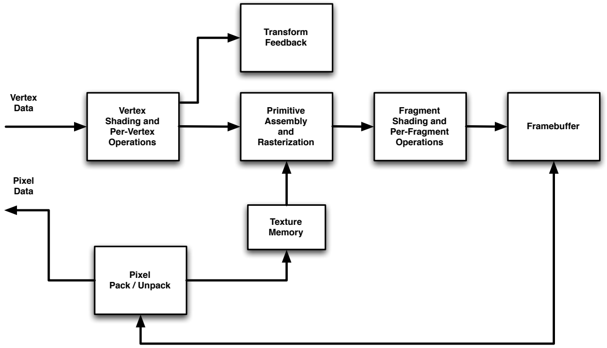
GL Command Syntax



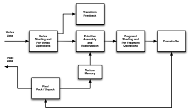
```
void glUniform4f( int location, float v0, float v1, float v2, float v3 );



void glGetFloatv( enum value, float *data );
```

Block Diagram of OpenGL

- To aid learning we will concentrate on each of the elements in turn
- Ignoring the others and assuming they just work out of the box without setup
- Finally we shall put the whole system together

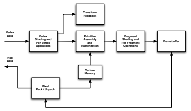

Transform Feedback
Vertex Data
Vertex Shading and Per-Vertex Operations
Primitive Assembly and Rasterization
Fragment Shading and Per-Fragment Operations
Framebuffer
Pixel Data
Texture Memory
Pixel Pack / Unpack

Block Diagram of OpenGL



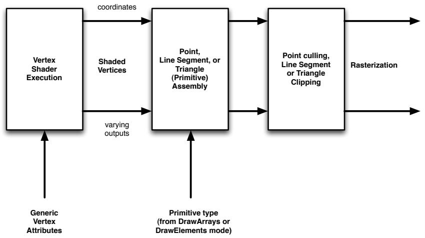
- Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Commands are effectively sent through a processing pipeline.
- The first stage operates on geometric primitives described by vertices: points, line segments, and polygons.
- In this stage vertices may be transformed and lit, followed by assembly into geometric primitives, which may optionally be used by the next stage, geometry shading, to generate new primitives.

Block Diagram of OpenGL



- The final resulting primitives are clipped to a viewing volume in preparation for the next stage, rasterization.
- The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon.
- Each fragment produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer.
- Finally, values may also be read back from the framebuffer or copied from one portion of the framebuffer to another.

Primitives and Vertices


coordinates
Vertex Shader Execution
Shaded Vertices
varying outputs
Point, Line Segment, or Triangle (Primitive) Assembly
Point culling, Line Segment or Triangle Clipping
Rasterization
Generic Vertex Attributes
Primitive type (from DrawArrays or DrawElements mode)

- In the OpenGL, most geometric objects are drawn by specifying a series of generic attribute sets using DrawArrays or one of the other drawing commands.
- Points, lines, polygons, and a variety of related geometric objects can be drawn in this way.

Primitives and Vertices

- Each vertex is specified with one or more generic vertex attributes.
- Each attribute is specified with one, two, three, or four scalar values.
- Generic vertex attributes can be accessed from within vertex shaders and used to compute values for consumption by later processing stages.
- For example we may set vertex colour, vertex normals, texture co-ordinates or generic vertex attributes used by the processing shader

Primitive types

- OpenGL has a number of primitive types that can be specified to DrawArrays and other primitive drawing commands
- These are
  - *Points, Line Strips, Line Loops, Separate Lines, Triangle Strips, Triangle Fans, Separate Triangles, Lines with adjacency, Line Strips with Adjacency, Triangles with Adjacency, Triangle Strips with Adjacency, Separate Patches*
- We will investigate these elements later for now we will concentrate on drawing some points

Vertex Arrays

- Vertex data is placed into arrays that are stored in the server's address space (GPU).
- Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single OpenGL command.
- The client may specify up to the value of MAX_VERTEX_ATTRIBS arrays to store one or more generic vertex attributes.

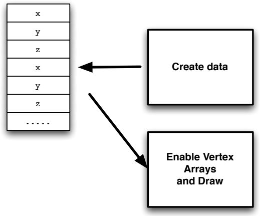
Vertex Arrays

- Vertex arrays are a simple way of storing data for models so that Vertices, normals and other information may be shared.
- This allows a more compact representation of data and reduces the number of calls OpenGL needs to make and the reduction of data stored.
- We can create the data in a series of arrays either procedurally or by loading it from some model format
- The data may then be stored either by downloading it to the GPU or storing it on the client side and telling the GPU to bind to it.

Example Drawing Points

- The following example will draw a series of points to the OpenGL context
- The data is stored on the client side (our program)
- We need to create the data (3xGLfloat values) in an array
- Then tell OpenGL where this data is and draw it

```
m_points = new GLfloat[3*s_numPoints];
```



Create the Data

```
GLfloat *m_points;


const static int s_numPoints=10000;


// first we generate random point x,y,z
     values
m_points = new GLfloat[3*s_numPoints];
ngl::Random *rand=ngl::Random::instance();
for( int i=0; i<3*s_numPoints; ++i)
{
  m_points[i]=rand->randomNumber(1);
}
```

in GLWindow.h

in ctor we allocate some random x,y,z values

Drawing the Data

tell GL we are using Vertex Arrays

```
void GLWindow::paintGL()
{
 // clear the screen and depth buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnableClientState(GL_VERTEX_ARRAY);

  glVertexPointer(3, GL_FLOAT, 0, m_points);
  glDrawArraysInstancedARB(GL_POINTS, 0, s_numPoints,s_numPoints/3.0);
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Now Draw

tell GL where the data is on the client

glVertexPointer

```
void glVertexPointer( GLint size,GLenum type,GLsizei stride,const GLvoid *pointer);
```

- size :- the number of coordinates per vertex. Must be 2, 3, or 4. The initial value is 4.
- type :- the data type of each coordinate in the array. Symbolic constants GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE are accepted. default : GL_FLOAT.
- stride :- the byte offset between consecutive vertices. If stride is 0, the vertices are understood to be tightly packed in the array. default : 0
- pointer :- a pointer to the first coordinate of the first vertex in the array. default : 0.

glVertexPointer

- glVertexPointer has been marked for deprecation
- At present it works but the new method of drawing is a lot more complex and requires different shaders to be developed.
- We usually do not use client side data anyway so this is just serving as an example before we add to the overall GL construction required for GL 3.x functionality
- We will expand on the new functions later

glDrawArraysInstancedARB

```
void glDrawArrays(  GLenum mode,GLint  first,GLsizei  count);
```

- mode :- what kind of primitives to render.
- Symbolic constants GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP, GL_QUADS, and GL_POLYGON are accepted.
- first :- the starting index in the enabled arrays.
- count :- the number of indices to be rendered.

ARB and GLEW

- ARB stands for Architecture review board and are extensions authorised by the OpenGL standards group
- Most of these extensions are part of the OpenGL driver for the GPU installed and we need to make link between the driver binary library and the OpenGL code we are writing
- This process is quite complex and to make it easier we can use GLEW (not required on Mac OSX)
- The ngl::Init class contains the following code to do this

Initialising GLEW

```
// we only need glew on linux mac ok (should add a windows ref as well)
#if defined(LINUX) || defined(WIN32)
{
  GLenum err = glewInit();
  if (GLEW_OK != err)
  {
    /* Problem: glewInit failed, something is seriously wrong. */
    std::cerr<< "Error: "<<glewGetErrorString(err)<<std::endl;
    exit(EXIT_FAILURE);
  }
  std::cerr<<"Status: Using GLEW "<<glewGetString(GLEW_VERSION)<<std::endl;
}
#endif
```

Adding Extensions

```
// make sure you've included glext.h
extern PFNGLISRENDERBUFFEREXTPROC glIsRenderbufferEXT;

and in one c/cpp file:
PFNGLISRENDERBUFFEREXTPROC glIsRenderbufferEXT;

// now bind the address from the driver to our function pointer

glIsRenderbufferEXT = glXGetProcAddressARB((const GLubyte*)"glIsRenderbufferEXT");
```

- For every function we need to access we have to write some code similar to the lines above
- This is quite tedious so GLEW does it for us

Adding Colour

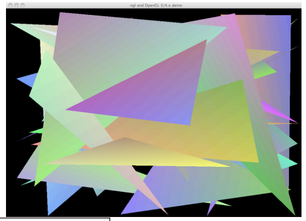
- The process of adding Colour is similar to that of setting the vertices
- We create a single array of per-vertex RGB values
- We then create a pointer to this and draw using the same draw command

```
/// @brief an array of colours
GLfloat *m_colours;

// first we generate random point x,y,z values
m_colours = new GLfloat[3*s_numPoints];
for( int i=0; i<3*s_numPoints; ++i)
{
  m_colours[i]=rand->randomPositiveNumber(0.6)+0.4;
}
```

Adding Colour

```
void GLWindow::paintGL()
{
 // clear the screen and depth buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);

  glVertexPointer(3, GL_FLOAT, 0, m_points);
  glColorPointer(3,GL_FLOAT,0,m_colours);
  glDrawArraysInstancedARB(GL_POINTS, 0, s_numPoints,s_numPoints/3.0);
  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_COLOR_ARRAY);

}
```

Other Primitives



```
void GLWindow::paintGL()
{
 // clear the screen and depth buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);

  glVertexPointer(3, GL_FLOAT, 0, m_points);
  glColorPointer(3,GL_FLOAT,0,m_colours);
  glDrawArraysInstancedARB(GL_TRIANGLES, 0, s_numPoints,s_numPoints/3.0);
  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_COLOR_ARRAY);

}
```

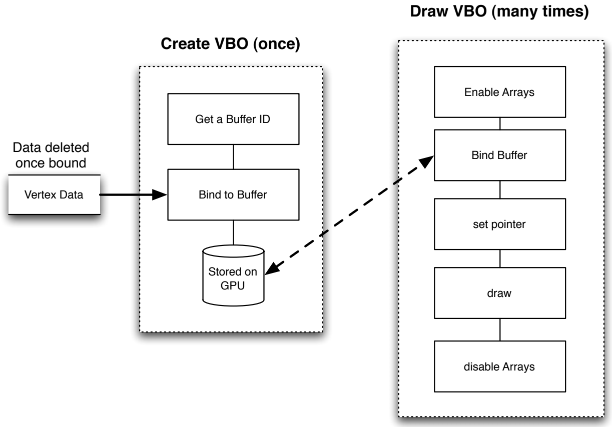
Problems

- Although this method works, it is still slow.
- Each frame the client (our program) has to send the data to the server (GPU)
- If we increase the s_numPoints constant to about 3000 the program slows to a halt
- We really need to create a faster method of doing things

Vertex Buffer Objects

- The idea behind VBOs is to provide regions of memory (buffers) accessible through identifiers.
- A buffer is made active through binding, following the same pattern as other OpenGL entities such as display lists or textures.
- Data is effectively stored on the GPU for execution and this greatly increases the speed of drawing.

VBO Allocation and process



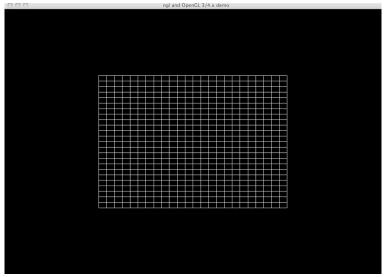
Data Packing

- We can pack the data in a number of ways for passing to the VBO
- The two simplest schemes are
  - All Vertex Data - All Normal Data - All Texture Data etc etc
  - Alternatively we can pack the data by interleaving the data in a number of pre-determined GL formats
- For the following examples we will use the 1st format.

Grid

in GLWindow.h

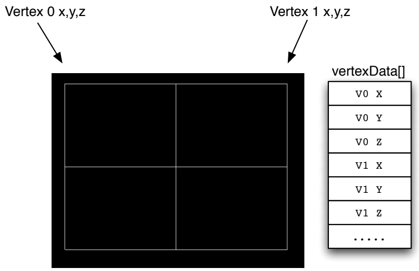
```
/// @brief a simple draw grid function
/// @param[in] _size the size of the grid (width and height)
/// @param[in] _step sxstep the spacing between grid points
/// @param[out] o_dataSize the size of the buffer allocated
/// @returns a pointer to the allocated VBO
GLuint  MakeGrid(
                  GLfloat _size,
                  int _steps,
                  int &o_dataSize
                 );
/// @brief a pointer to our VBO data
GLuint m_vboPointer;
/// @brief store the size of the vbo data
GLint m_vboSize;
```



```
const static float gridSize=1.5;
const static int steps=24;
```

Creating Data for vertices

```
// allocate enough space for our verts
// as we are doing lines it will be 2 verts per line
// and we need to add 1 to each of them for the <= loop
// and finally muliply by 12 as we have 12 values per line pair
o_dataSize= (_steps+2)*12;
float *vertexData= new float[o_dataSize];
// k is the index into our data set
int k=-1;
// claculate the step size for each grid value
float step=_size/(float)_steps;
// pre-calc the offset for speed
float s2=_size/2.0f;
// assign v as our value to change each vertex pair
float v=-s2;
// loop for our grid values
for(int i=0; i<=_steps; ++i)
{
  // vertex 1 x,y,z
  vertexData[++k]=-s2; // x
  vertexData[++k]=v; // y
  vertexData[++k]=0.0; // z

  // vertex 2 x,y,z
  vertexData[++k]=s2; // x
  vertexData[++k]=v; // y
  vertexData[++k]=0.0; // z

  // vertex 3 x,y,z
  vertexData[++k]=v;
  vertexData[++k]=s2;
  vertexData[++k]=0.0;

  // vertex 4 x,y,z
  vertexData[++k]=v;
  vertexData[++k]=-s2;
  vertexData[++k]=0.0;
  // now change our step value
  v+=step;
}
```



Binding the data

```
// now we will create our VBO first we need to ask GL for an Object ID
GLuint VBOBuffers;
// now create the VBO
glGenBuffers(1, &VBOBuffers);
// now we bind this ID to an Array buffer
glBindBuffer(GL_ARRAY_BUFFER, VBOBuffers);
// finally we stuff our data into the array object
// First we tell GL it's an array buffer
// then the number of bytes we are storing (need to tell it's a sizeof(FLOAT)
// then the pointer to the actual data
// Then how we are going to draw it (in this case Statically as the data will not change)
glBufferData(GL_ARRAY_BUFFER, o_dataSize*sizeof(GL_FLOAT) , vertexData, GL_STATIC_DRAW);

// now we can delete our client side data as we have stored it on the GPU
delete [] vertexData;
// now return the VBO Object pointer to our program so we can use it later for drawing
return VBOBuffers;
```

- Now we bind the data onto the GPU and once this is done we can delete the client side data as it's not needed

Building the Grid

- We must have a valid GL context before we can call this function, and if required we must initialise GLEW

```
void GLWindow::initializeGL()
{
  ngl::NGLInit *Init = ngl::NGLInit::instance();
  Init->initGlew();
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  glColor3f(1,1,1);
  m_vboPointer=MakeGrid(gridSize,steps,m_vboSize);
}
```

note glColor is deprecated

Drawing the buffer

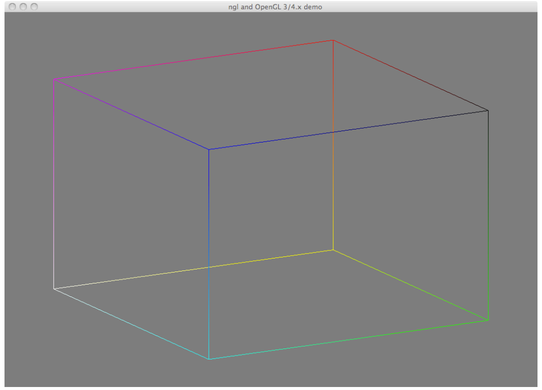
```
void GLWindow::paintGL()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // enable  vertex array drawing
  glEnableClientState(GL_VERTEX_ARRAY);
  // bind our VBO data to be the currently active one
  glBindBuffer(GL_ARRAY_BUFFER, m_vboPointer);
  // tell GL how this data is formated in this case 3
  // floats tightly packed starting at the begining
  // of the data (0 = stride, 0 = offset)
  glVertexPointer(3,GL_FLOAT,0,0);
  // draw the VBO as a series of GL_LINES starting at 0
  // in the buffer and _vboSize/3 as we have x,y,z
  glDrawArraysInstancedARB(GL_LINES, 0, m_vboSize/3,1);
  // now turn off the VBO client state as we have finished with it
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Vertex arrays

- To enable the use of vertex arrays we need very few steps as shown below
  1. Invoke the function `glEnableClientState(GL_VERTEX_ARRAY);` to activate the vertex-array feature of OpenGL
  2. Use the function `glVertexPointer` to specify the location and data format for the vertex co-ordinates
  3. Display the scene using a routine such as `glDrawArraysInstancedARB`

A Cube

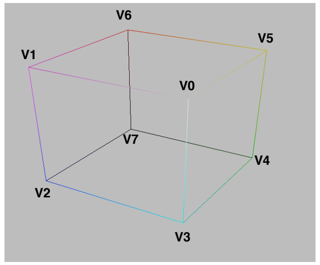
- This cube has vertex colours and vertex normals



Cube Vertices

```
 // vertex coords array
GLfloat vertices[] = {
                        1, 1, 1, -1, 1, 1, -1,-1, 1,  1,-1, 1,  // v0-v1-v2-v3
                        1, 1, 1,  1,-1, 1,  1,-1,-1,  1, 1,-1,  // v0-v3-v4-v5
                        1, 1, 1,  1, 1,-1, -1, 1,-1, -1, 1, 1,  // v0-v5-v6-v1
                       -1, 1, 1, -1, 1,-1, -1,-1,-1, -1,-1, 1,  // v1-v6-v7-v2
                       -1,-1,-1,  1,-1,-1,  1,-1, 1, -1,-1, 1,  // v7-v4-v3-v2
                        1,-1,-1, -1,-1,-1, -1, 1,-1,  1, 1,-1   // v4-v7-v6-v5
                       };
```

- The array above stores the vertices for a unit cube in face order of quads



Vertex Normals

```
// normal array
GLfloat normals[] = {
                      0, 0, 1,   0, 0, 1,  0, 0, 1,  0, 0, 1,     // v0-v1-v2-v3
                      1, 0, 0,   1, 0, 0,  1, 0, 0,  1, 0, 0,     // v0-v3-v4-v5
                      0, 1, 0,   0, 1, 0,  0, 1, 0,  0, 1, 0,     // v0-v5-v6-v1
                     -1, 0, 0,  -1, 0, 0, -1, 0, 0, -1, 0, 0,     // v1-v6-v7-v2
                      0,-1, 0,   0,-1, 0,  0,-1, 0,  0,-1, 0,     // v7-v4-v3-v2
                      0, 0,-1,   0, 0, -1, 0, 0,-1,  0, 0,-1      // v4-v7-v6-v5
                    };
```

- This array stores the normals for each vertex

Vertex Colour Array

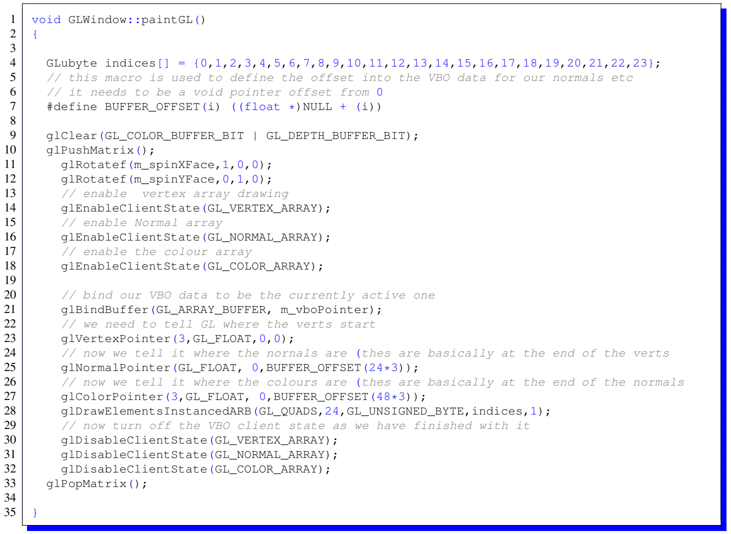
```
// color array
GLfloat colours[] =
                    {
                      1,1,1,  1,1,0,  1,0,0,  1,0,1,  // v0-v1-v2-v3
                      1,1,1,  1,0,1,  0,0,1,  0,1,1,  // v0-v3-v4-v5
                      1,1,1,  0,1,1,  0,1,0,  1,1,0,  // v0-v5-v6-v1
                      1,1,0,  0,1,0,  0,0,0,  1,0,0,  // v1-v6-v7-v2
                      0,0,0,  0,0,1,  1,0,1,  1,0,0,  // v7-v4-v3-v2
                      0,0,1,  0,0,0,  0,1,0,  0,1,1   // v4-v7-v6-v5
                    };
```

- Array of colour values for each vertex these will be interpolated across the faces

Assigning the data

- In this case we have 3 different arrays which we are going to combine into one VBO buffer.
- The data will be packed in the format
  - Vertices -> Normal -> Colour
- First we have to allocate enough space for all 3 arrays

```
void GLWindow::createCube(
                            GLfloat _scale,
                            GLuint &o_vboPointer
                         )
{


 // first we scale our vertices to _scale
  for(int i=0; i<24*3; ++i)
  {
    vertices[i]*=_scale;
  }
  // now create the VBO
  glGenBuffers(1, &o_vboPointer);
  // now we bind this ID to an Array buffer
  glBindBuffer(GL_ARRAY_BUFFER, o_vboPointer);

  // this time our buffer is going to contain verts followed by normals
  // so allocate enough space for all of them
  glBufferData(GL_ARRAY_BUFFER, 72*3*sizeof(GL_FLOAT) , 0, GL_STATIC_DRAW);
  // now we copy the data for the verts into our buffer first
  glBufferSubData(GL_ARRAY_BUFFER,0,24*3*sizeof(GL_FLOAT),vertices);
  // now we need to tag the normals onto the end of the verts
  glBufferSubData(GL_ARRAY_BUFFER,24*3*sizeof(GL_FLOAT),24*3*sizeof(GL_FLOAT),normals);

  // now we need to tag the colours onto the end of the normals
  glBufferSubData(GL_ARRAY_BUFFER,48*3*sizeof(GL_FLOAT),24*3*sizeof(GL_FLOAT),colours);

}
```



```
void GLWindow::paintGL()
{

  GLubyte indices[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23};
  // this macro is used to define the offset into the VBO data for our normals etc
  // it needs to be a void pointer offset from 0
  #define BUFFER_OFFSET(i) ((float *)NULL + (i))

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
    glRotatef(m_spinXFace,1,0,0);
    glRotatef(m_spinYFace,0,1,0);
    // enable  vertex array drawing
    glEnableClientState(GL_VERTEX_ARRAY);
    // enable Normal array
    glEnableClientState(GL_NORMAL_ARRAY);
    // enable the colour array
    glEnableClientState(GL_COLOR_ARRAY);

    // bind our VBO data to be the currently active one
    glBindBuffer(GL_ARRAY_BUFFER, m_vboPointer);
    // we need to tell GL where the verts start
    glVertexPointer(3,GL_FLOAT,0,0);
    // now we tell it where the nornals are (thes are basically at the end of the verts
    glNormalPointer(GL_FLOAT, 0,BUFFER_OFFSET(24*3));
    // now we tell it where the colours are (thes are basically at the end of the normals
    glColorPointer(3,GL_FLOAT, 0,BUFFER_OFFSET(48*3));
    glDrawElementsInstancedARB(GL_QUADS,24,GL_UNSIGNED_BYTE,indices,1);
    // now turn off the VBO client state as we have finished with it
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);
  glPopMatrix();

}
```

What Next

- We have used several deprecated features in this lecture but they serve to easily demonstrate the OpenGL process
- Next time we will investigate the OpenGL shading language and begin to learn some other elements of the pipeline
- We can then combine them to produce a full Core profile OpenGL application

References

- Segal M, Akeley K The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010)
- F S. Hill Computer Graphics Using Open GL (3rd Edition)
- Shreiner Et Al OpenGL Programming Guide: The Official Guide to Learning OpenGL
- Foley & van Dam Computer Graphics: Principles and Practice in C (2nd Edition)
- (Redbook online) http://fly.cc.fer.hr/~unreal/theredbook/