OpenGL Shading Language (GLSL)

Sang Il Park

Dept. of Software

A First Program: Many points with GPU

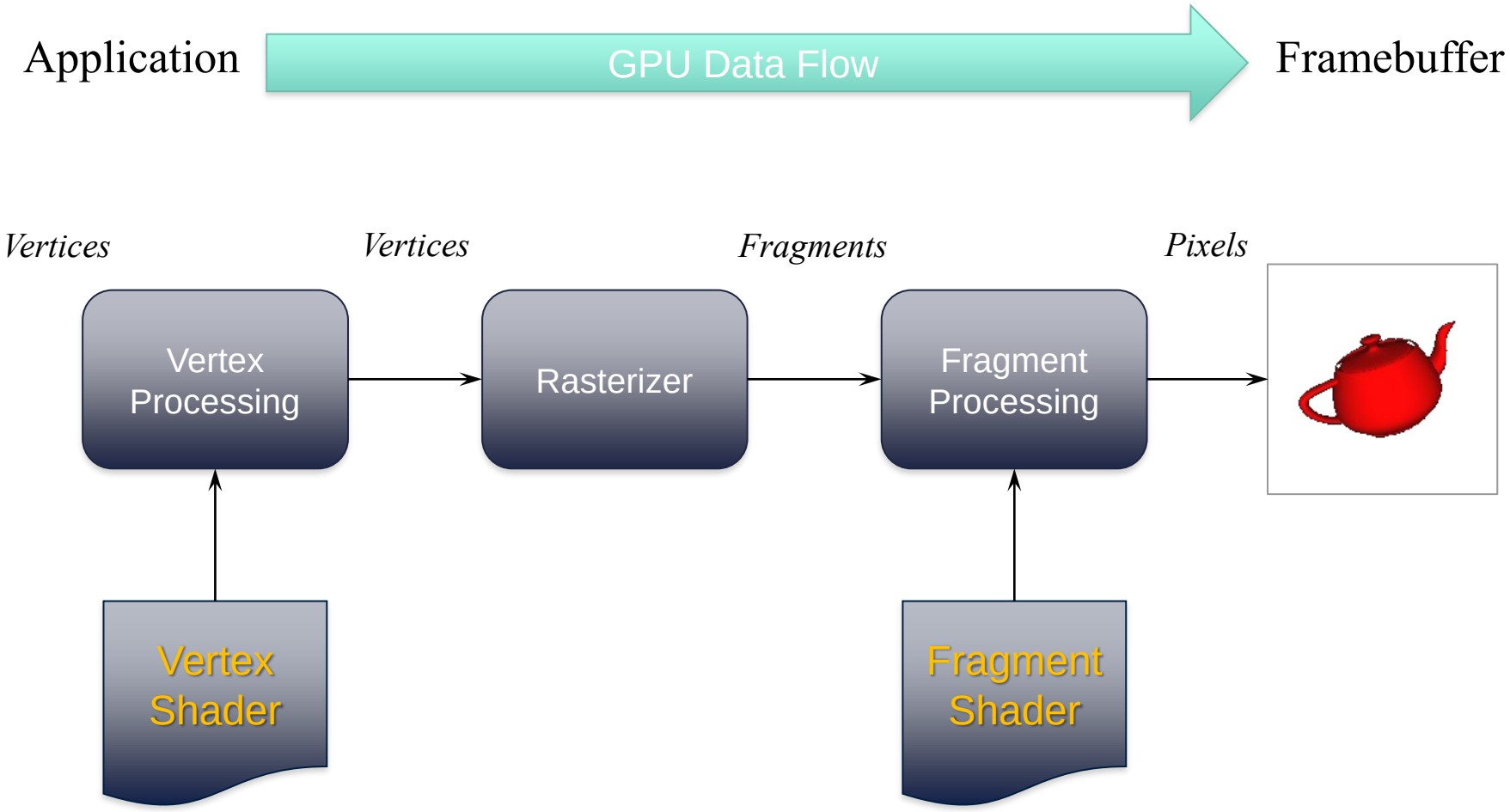
Setting for the most current opengl version for your computer

```
int main(int argc, char ** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA);
        glutInitWindowSize(512,512);
        glutCreateWindow("Many Points GPU");

        glewExperimental = true;
        glewInit();

        printf("OpenGL %s, GLSL %s\n",
                glGetString(GL_VERSION),
                glGetString(GL_SHADING_LANGUAGE_VERSION));

        glutDisplayFunc(display);
        glutMainLoop();

        return 0;
}
```

For using the modern OpenGL

To check the Current version

OpenGL Pipeline (Simplified)

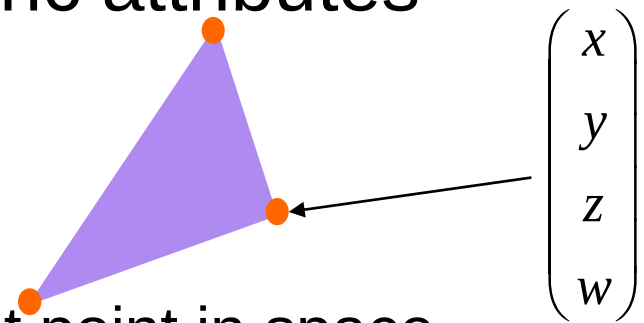


OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  1. Create buffer objects and load data into them
  2. Create shader programs
  3. “Connect” data locations with shader variables
  4. Render

Representing Geometric Objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space



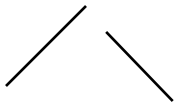
- Position stored in 4 dimensional homogeneous coordinates

- Vertex data must be stored in *vertex buffer objects* (***VBO***s)
- VBOs must be stored in *vertex array objects* (***VAO***s)

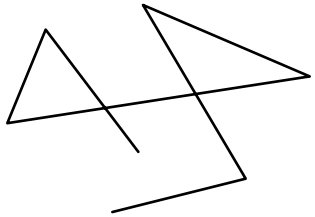
OpenGL's Geometric Primitives

- All primitives are specified by vertices

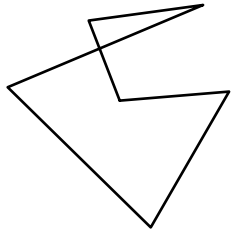
GL_POINTS



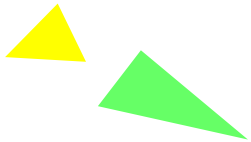
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Creating Data

- Define an array for storing all the points

```
struct vec2
{
        float x;
        float y;
};

const int NumPoints = 5000;

void init()
{
        vec2 points[NumPoints];

        for ( int i = 0; i < NumPoints; i++ )
        {
                points[i].x = (rand()%200)/100.0f-1.0f;
                points[i].y = (rand()%200)/100.0f-1.0f;
     }
}
```

Draw the array at once

- Define an array for storing all the points

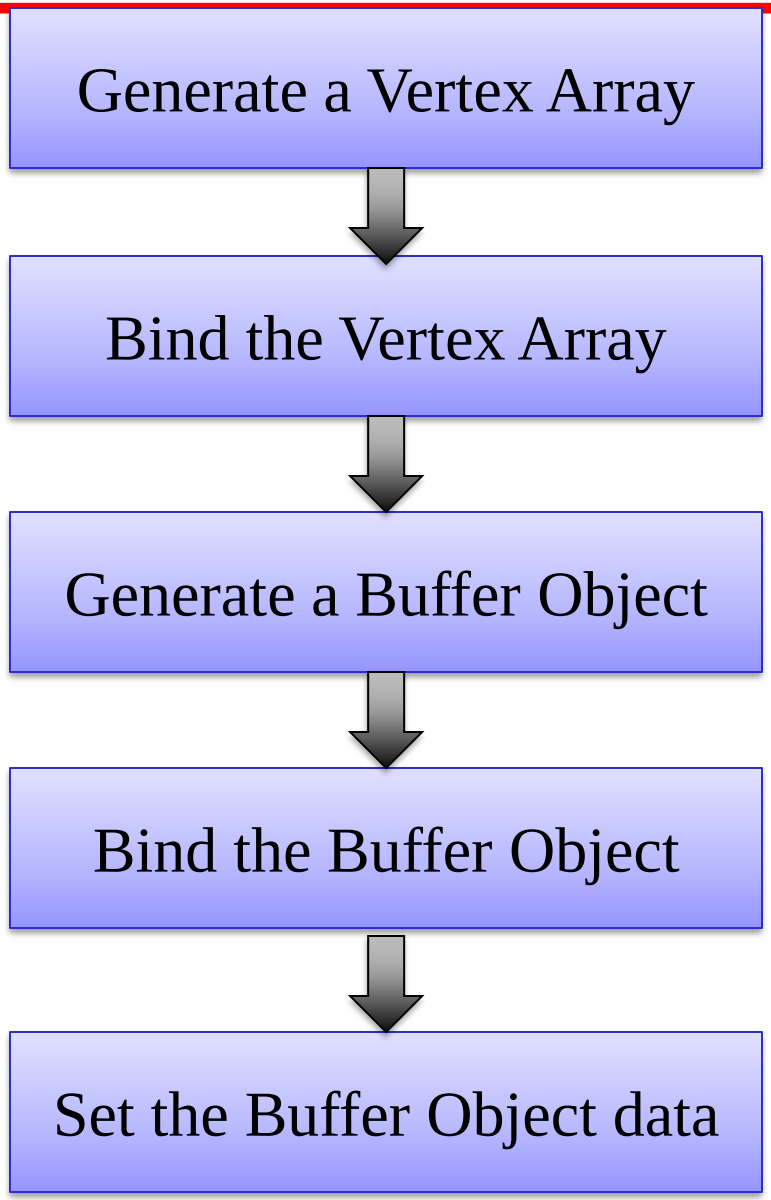
```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glDrawArrays(GL_POINTS, 0, NumPoints);
}
```

Above code draws the data in GPU.
But we didn't send the data to GPU at all!!

How to send your data

- Vertex data must be stored in *vertex buffer objects* (***VBO***s)
- VBOs must be stored in *vertex array objects* (***VAO***s)

How to send your data




```
glGenVertexArray(..)
glBindVertexArray(..)
glGenBuffers(..)
glBindBuffer(..)
glBufferData(..)
```

Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Storing Vertex Attributes

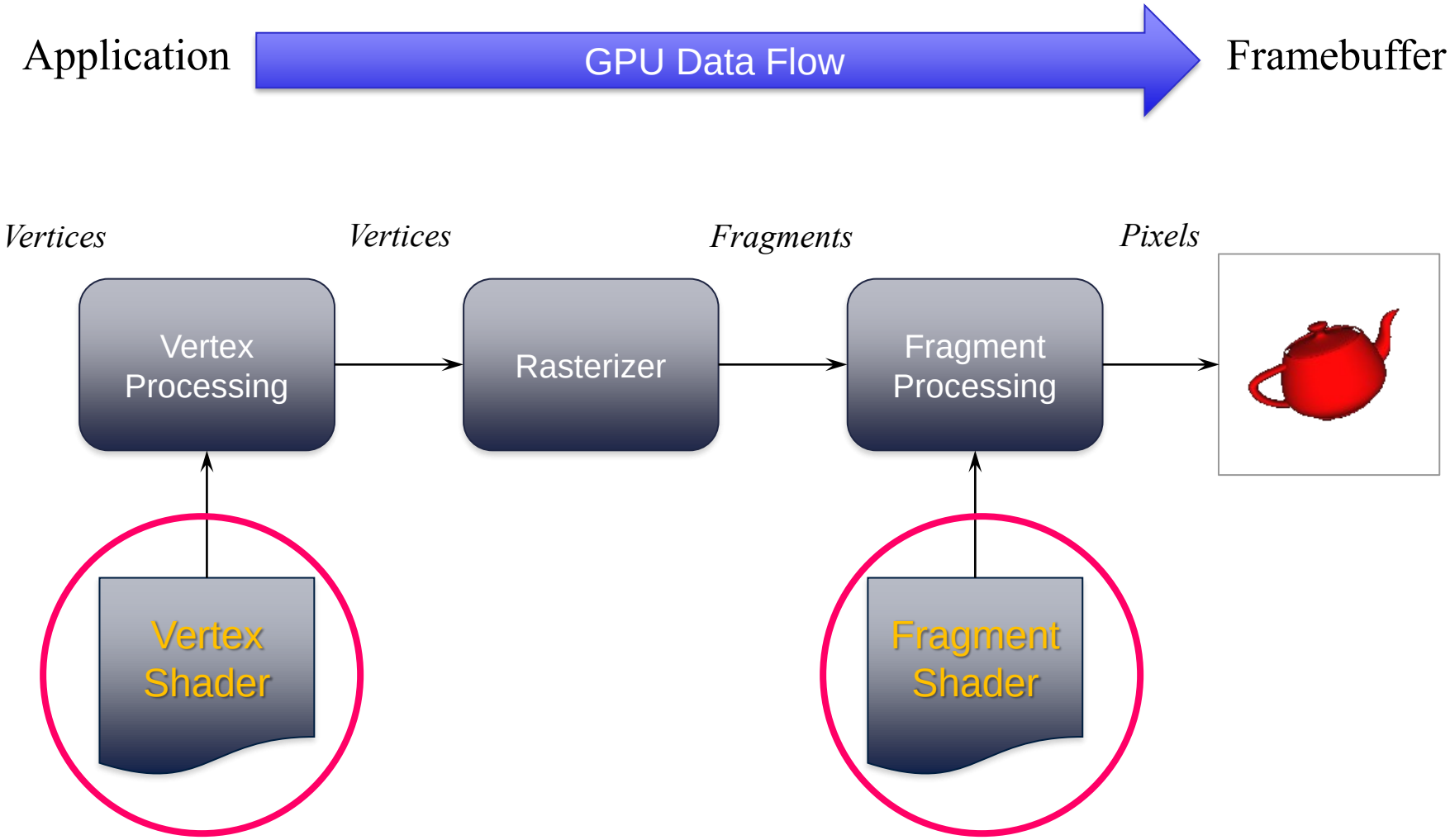
- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`

  - bind VAO for use in rendering later `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points),
                  points, GL_STATIC_DRAW);
```

We need shaders before drawing



Vertex Shader Code (vshader.glsl)

```
#version 330

in vec4 vPosition;

void main()
{
     gl_Position = vPosition;
}
```

Fragment Shader Code (fshader.glsl)

```
#version 330

out vec4 fColor;

void main()
{
     fColor = vec4(1.0,0.0,0.0,1.0);
}
```

Loading Shaders

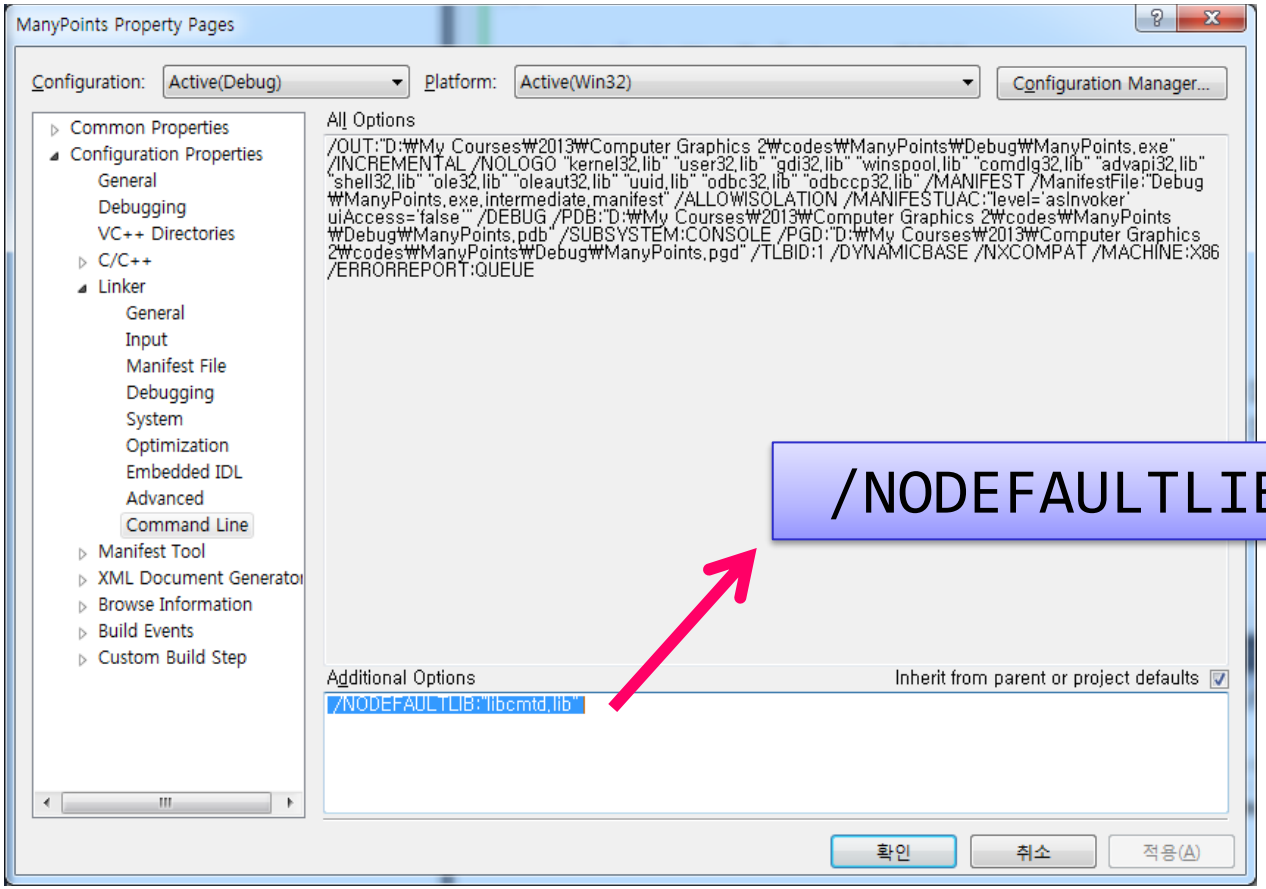
```
#include <InitShader.h>
```

```
// Load and use shaders
GLuint program
      = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );
```

glsl : opengl shader language
Those are provided in our homepage.

If you see an error:
You should change Project Setting

- Conflict with an existing lib “libcmtd.lib”:



Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
int vPosition = 0;
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 2, GL_FLOAT,
                       GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Drawing Geometric Primitives

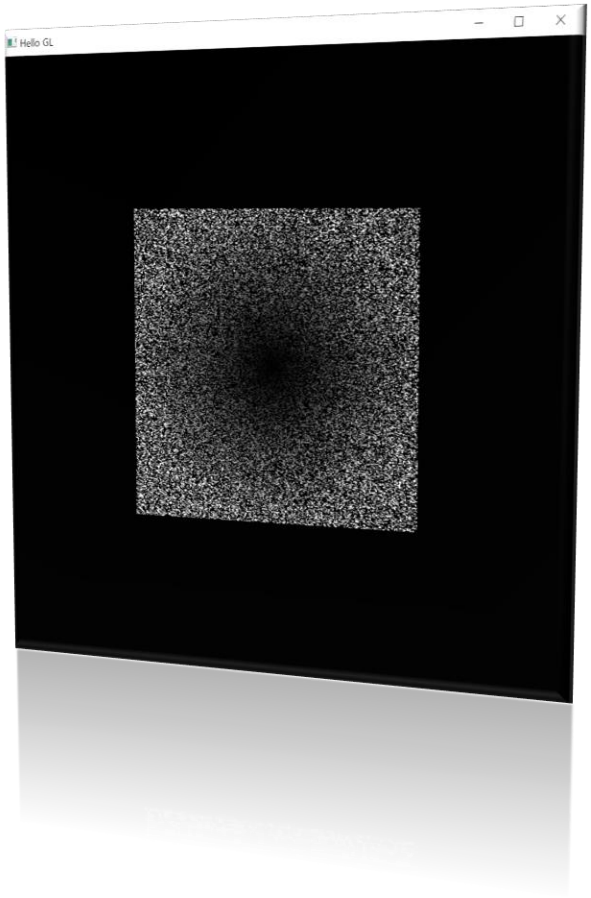
- For contiguous groups of vertices

```
glDrawArrays(GL_POINTS, 0, NumPoints);
```

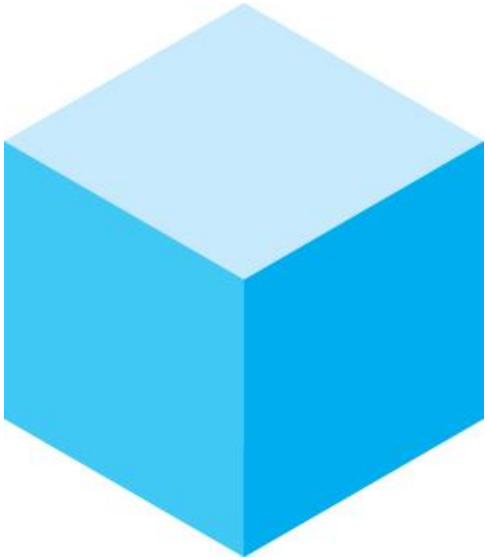
- Usually invoked in display callback
- Initiates vertex shader

Summary

- Setting for libraries
  - Set include/lib folder
  - `#include <vgl.h>`
  - `#include <initshader.h>`
- Creating data(in an array form)
- Sending the data
  - VAO – vertex array object
  - VBO – vertex buffer object
- Loading the shaders
- Draw it with `glDrawArrays(...)`



A Color Cube example



Rendering a Cube

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a vec4 class (implemented in C++) similar to GLSL's vec4 type

```
#include <vec.h>
```

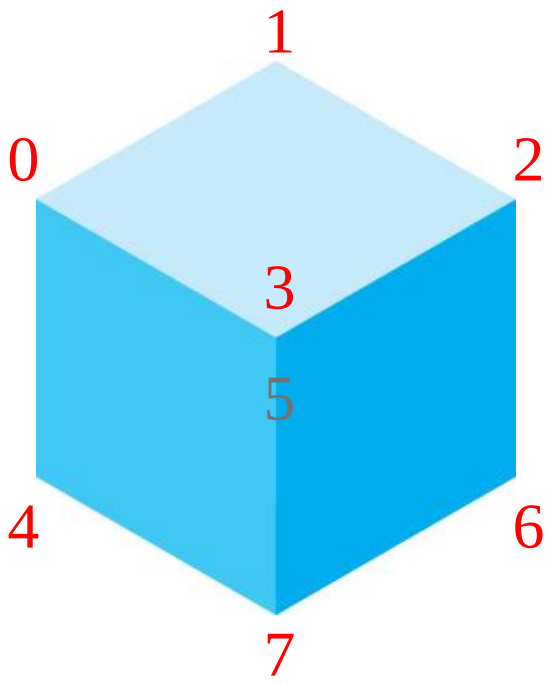
Initializing the Cube's Data (cont'd)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
vec4  points[NumVertices];
vec4  colors[NumVertices];
```

Cube Data

```
// Vertices of a unit cube centered at origin, sides aligned
  with axes
  vec4 vertex_pos [8] = {
        vec4 ( -0.5, -0.5,  0.5, 1.0 ),
        vec4 ( -0.5,  0.5,  0.5, 1.0 ),
        vec4 (  0.5,  0.5,  0.5, 1.0 ),
        vec4 (  0.5, -0.5,  0.5, 1.0 ),
        vec4 ( -0.5, -0.5, -0.5, 1.0 ),
        vec4 ( -0.5,  0.5, -0.5, 1.0 ),
        vec4 (  0.5,  0.5, -0.5, 1.0 ),
        vec4 (  0.5, -0.5, -0.5, 1.0 )
};
```

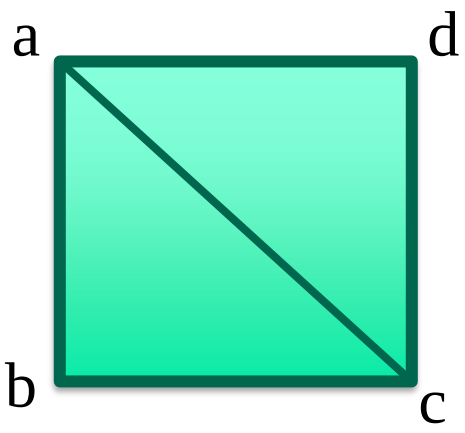


Cube Data

```
// RGBA colors
  vec4 vertex_colors[8] = {
       vec4 ( 0.0, 0.0, 0.0, 1.0 ),  // black
       vec4 ( 1.0, 0.0, 0.0, 1.0 ),  // red
       vec4 ( 1.0, 1.0, 0.0, 1.0 ),  // yellow
       vec4 ( 0.0, 1.0, 0.0, 1.0 ),  // green
       vec4 ( 0.0, 0.0, 1.0, 1.0 ),  // blue
       vec4 ( 1.0, 0.0, 1.0, 1.0 ),  // magenta
       vec4 ( 1.0, 1.0, 1.0, 1.0 ),  // white
       vec4 ( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

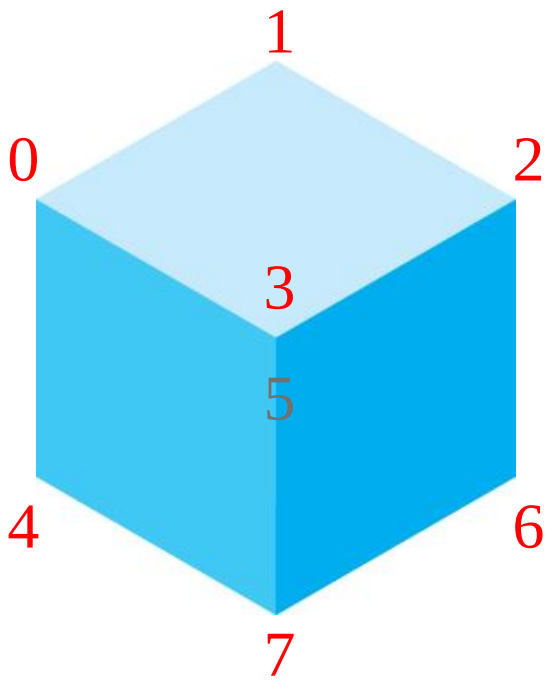
Generating a Cube Face from Vertices

```
// quad() generates two triangles for each face and assigns colors to the
  vertices
int Index = 0;  // global variable indexing into VBO arrays

void quad(int a, int b, int c, int d) {
    colors[Index] = vertex_colors[a]; points[Index] = vertex_pos[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertex_pos[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_pos[c]; Index++;

    colors[Index] = vertex_colors[a]; points[Index] = vertex_pos[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_pos[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertex_pos[d]; Index++;
}
```



Generating the Cube from Faces

```
// generate 12 triangles: 36 vertices and 36
  colors
void
colorcube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```



Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`
  - bind VAO for use in rendering `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
             sizeof(colors), NULL, GL_STATIC_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
             sizeof(colors), colors);
```

Loading Shaders

```
#include <InitShader.h>
```

```
// Load and use shaders
GLuint program
      = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );
```

glsl : opengl shader language
Those are provided in our homepage.

Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation(program, "vPosition”);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(0));


GLuint vColor = glGetAttribLocation(program, "vColor”);
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0,
  BUFFER_OFFSET(sizeof(points)));
```

Drawing Geometric Primitives

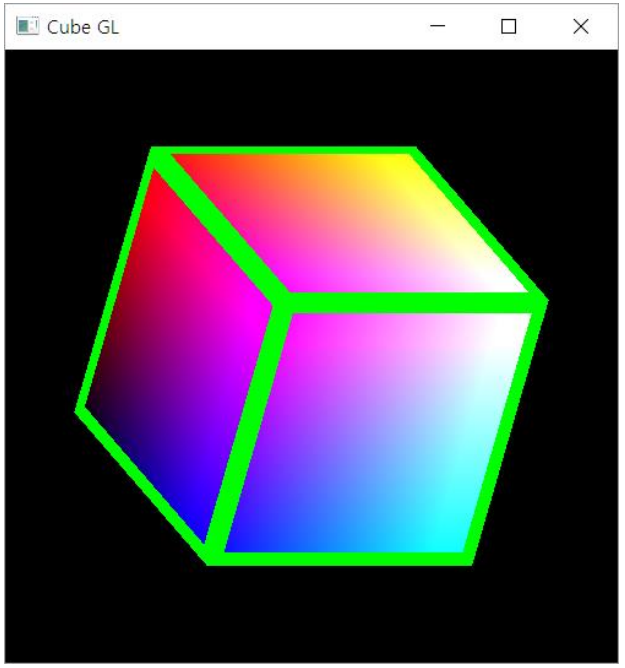
- For contiguous groups of vertices

```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```

- Usually invoked in display callback
- Initiates vertex shader

Summary

- We learnt the very basic of defining geometry and use it on GPU
- Today, we will learn how to code the shaders



- ***Vertex Shader*** :
  - Determining the positions of vertex
  - Useful for scaling, rotating, deforming and so on.
  - Also important for preparing some information sending to Fragment Shader
- ***Fragment Shader*** :
  - Determining the color of each fragment