CG Practice 2

COLLEGE OF COMPUTING

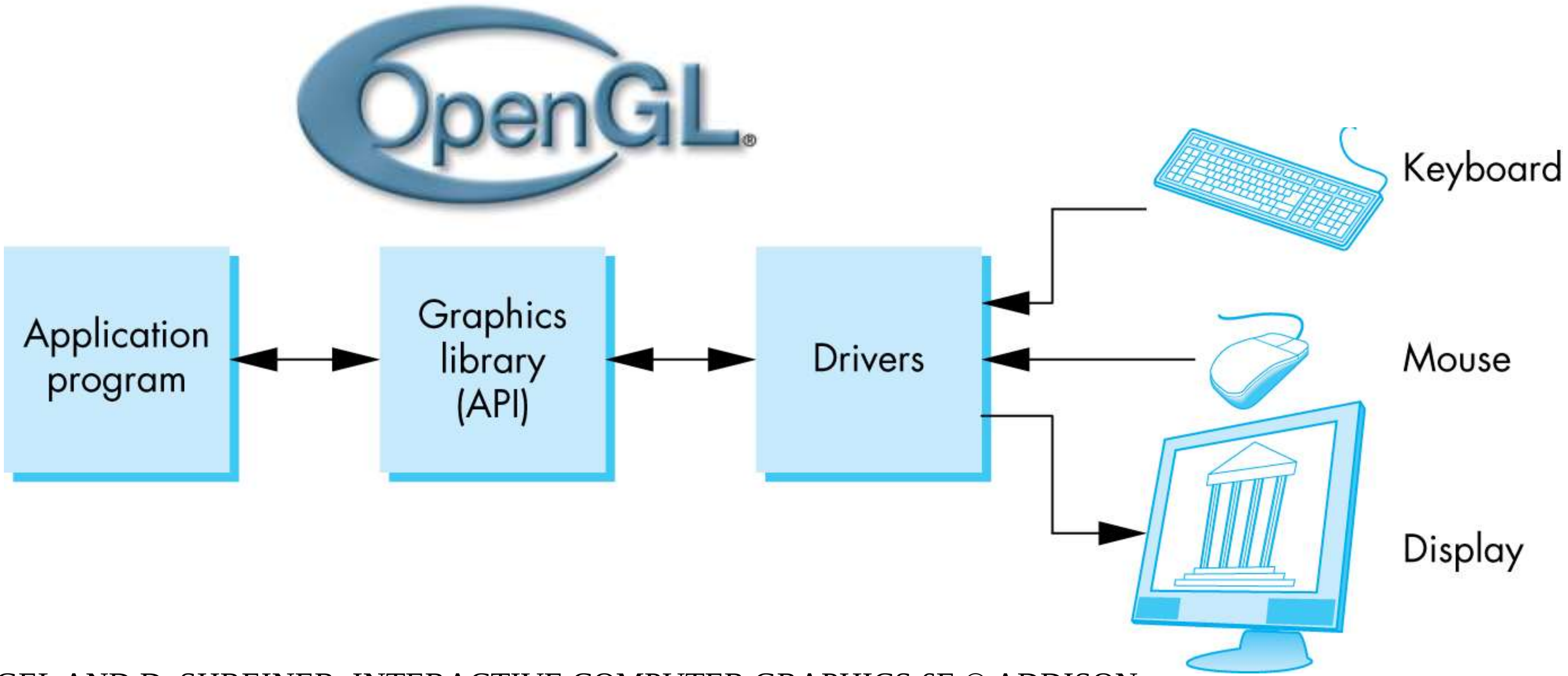
HANYANG ERICA CAMPUS

Q YOUN HONG (홍규연)

OpenGL Basics (Review)

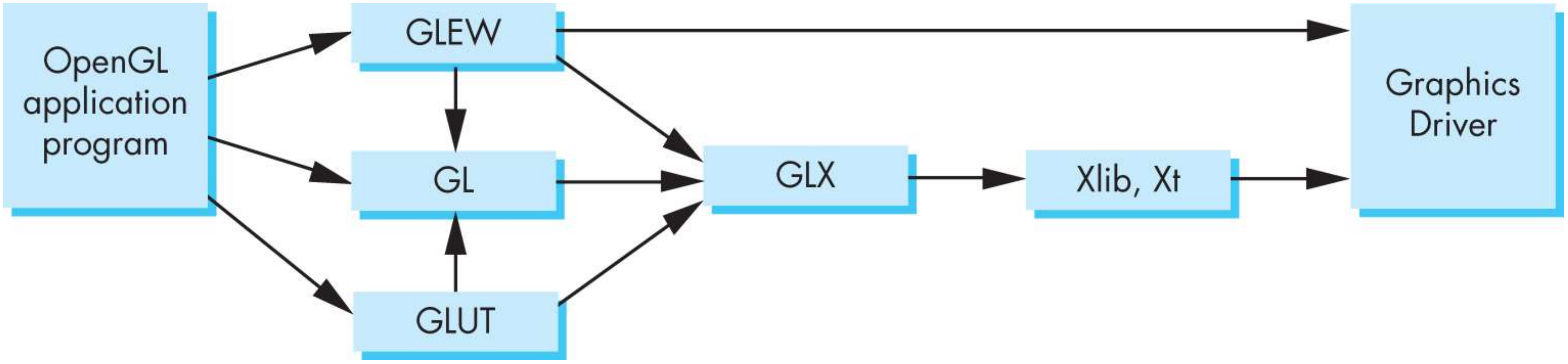
Review

- OpenGL is a software API to graphics hardware



E. ANGEL AND D. SHREINER: INTERACTIVE COMPUTER GRAPHICS 6E © ADDISON-WESLEY 2012

Review – OpenGL Libraries



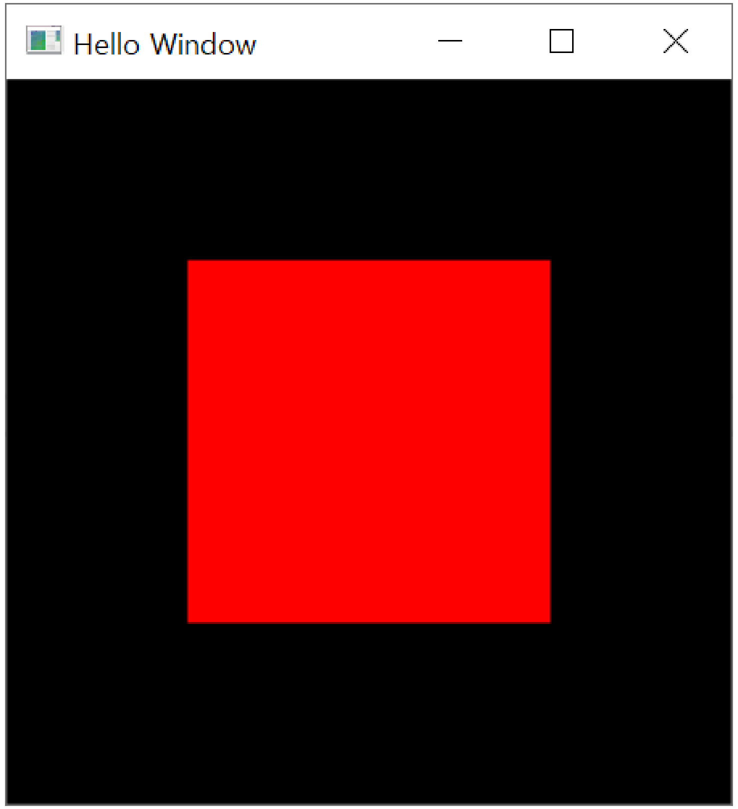
E. ANGEL AND D. SHREINER: INTERACTIVE COMPUTER GRAPHICS 6E © ADDISON-WESLEY 2012

Review – OpenGL Modes

Immediate Mode	Core-Profile Mode
• Fixed-function based • Draw immediately (no need for memory of geometric data) • Used in legacy OpenGL (Ver. 1.0)	• Store the geometric data in buffer (GPU) • Draw the data in the buffer • Used in modern OpenGL (≥ Ver. 1.1)

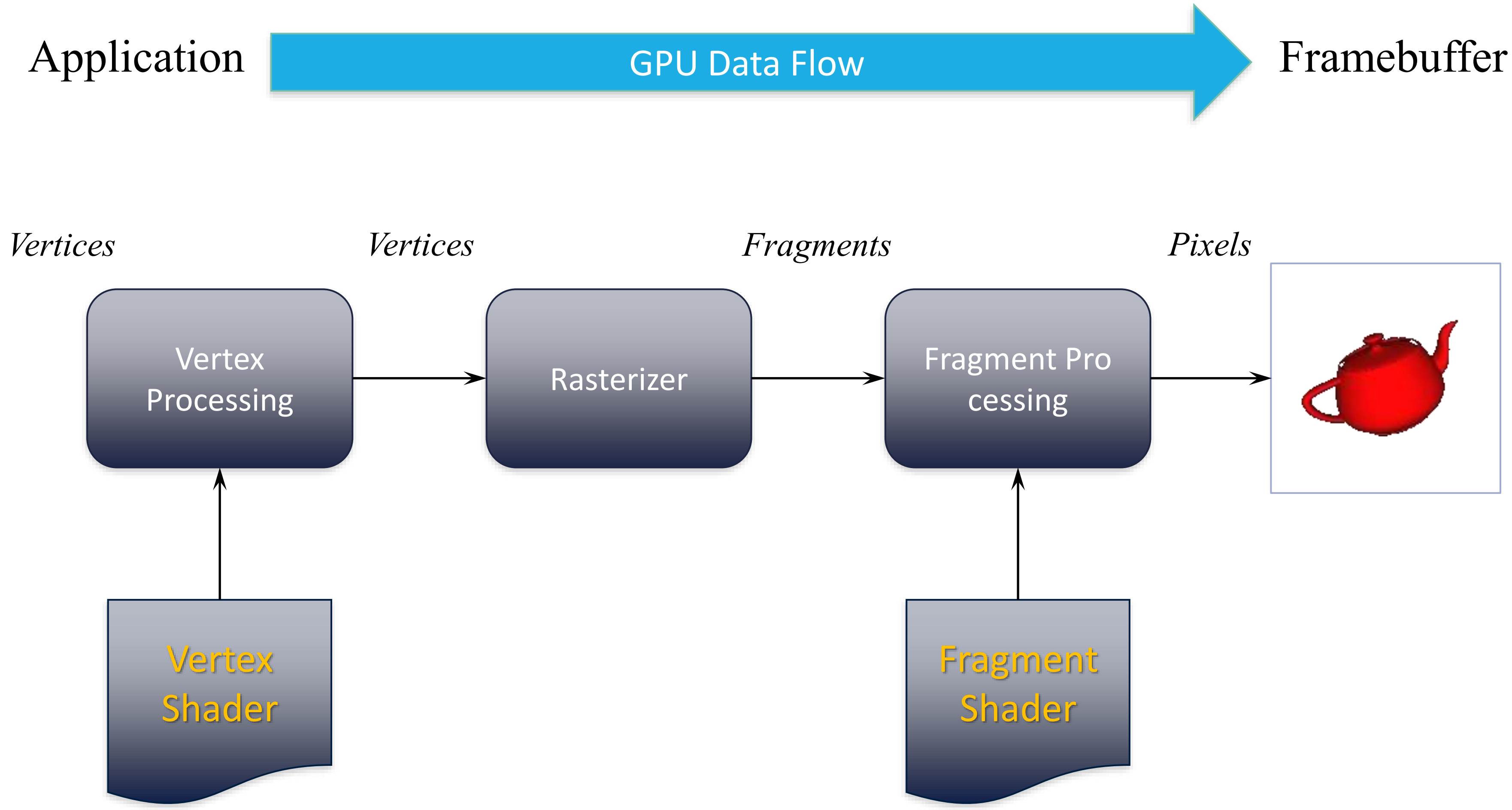
Review - immediate mode example

```
#include <vgl.h>

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
    glVertex2f(-0.5, -0.5);
    glVertex2f(0.5, -0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(-0.5, 0.5);
    glEnd();
    glFlush();
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutCreateWindow("Hello Window");

    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```



OpenGL Shaders

OpenGL Pipeline



OpenGL Shader Programming

1. Create buffer objects and load data
2. Create shader programs
3. Connect data locations with shader variables
4. Render

Example1.

Drawing many random points on 2D plane

Creating Data

- Define a 2D point with vec2 (in vec.h)
- vec2, vec3, vec4, mat2, mat3, mat4 are provided (vec.h, mat.h)
- Define an array to store all points at once (application-side)

```
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;
        }
}
```

Drawing the array at once

- Define an array for storing all the points

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

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

main()

- Almost same as the source code in the immediate mode
- Initialization added!

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

    glewInit();
    init();

    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

How to send 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 data

Generate a Vertex Array	`glGenVertexArray(..)`
Bind the Vertex Array	`glBindVertexArray(..)`
Generate a Buffer Object	`glGenBuffers(..)`
Bind the Buffer Object	`glBindBuffer(..)`
Set the Buffer Object data	`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);
```

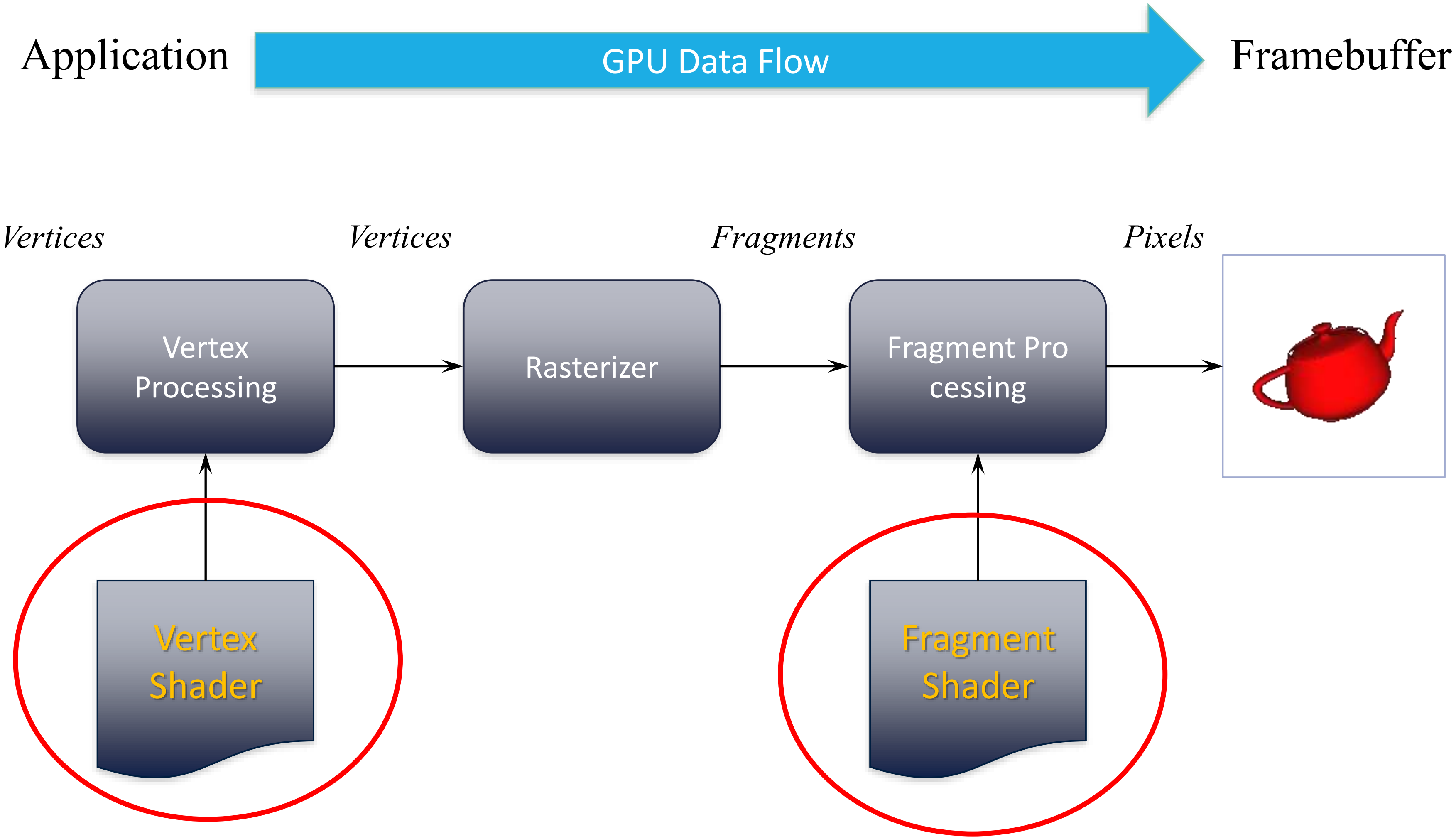
Storing Vertex Attributes

- 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!



Loading Shaders

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

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

GLSL : opengl shader language

Vertex Shader (vshader.glsl)

```
#version 330

in vec4 vPosition;

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

Fragment Shader (fshader.glsl)


```
#version 330

out vec4 fColor;

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

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 vPos = glGetAttribLocation(program,
"vPosition");

glEnableVertexAttribArray( vPos );

glVertexAttribPointer( vPos, 2, GL_FLOAT,
     GL_FALSE, 0, BUFFER_OFFSET(0) );
```

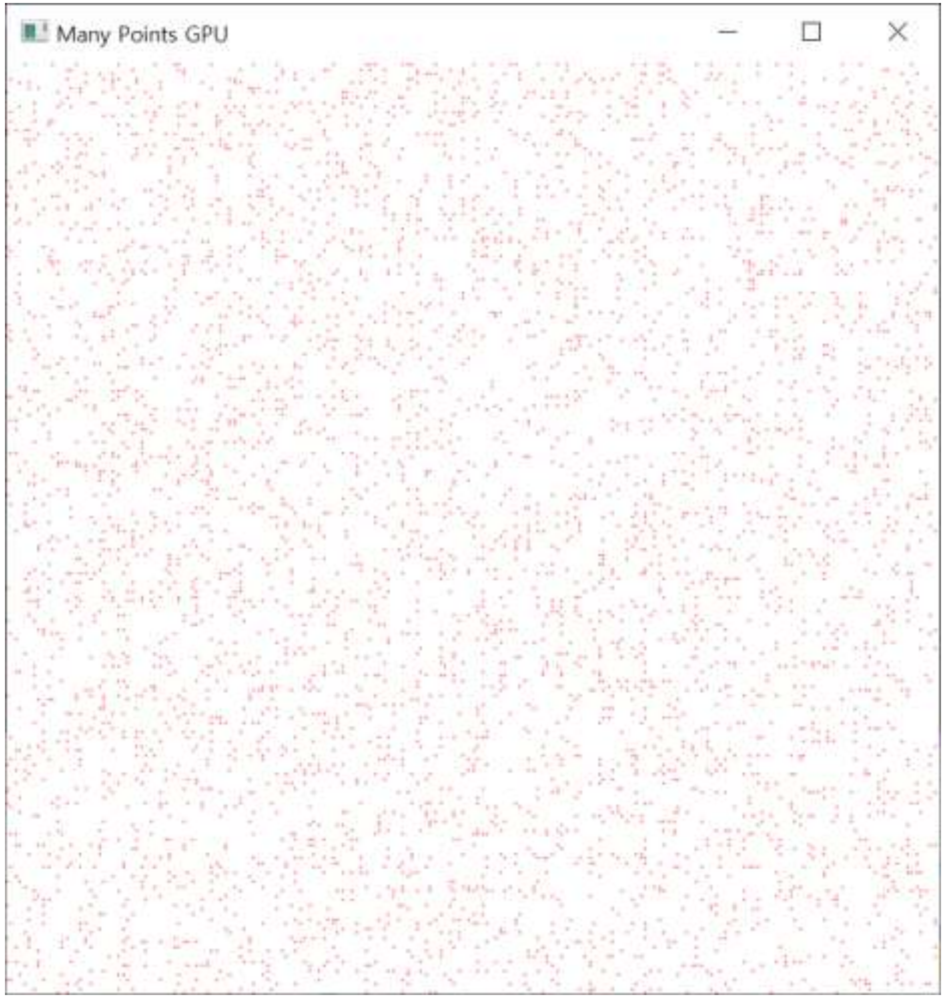
Drawing Geometric Primitives

- For contiguous groups of vertices

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

- Usually invoked in display callback
- Initiates vertex shader

Let's Run Program!

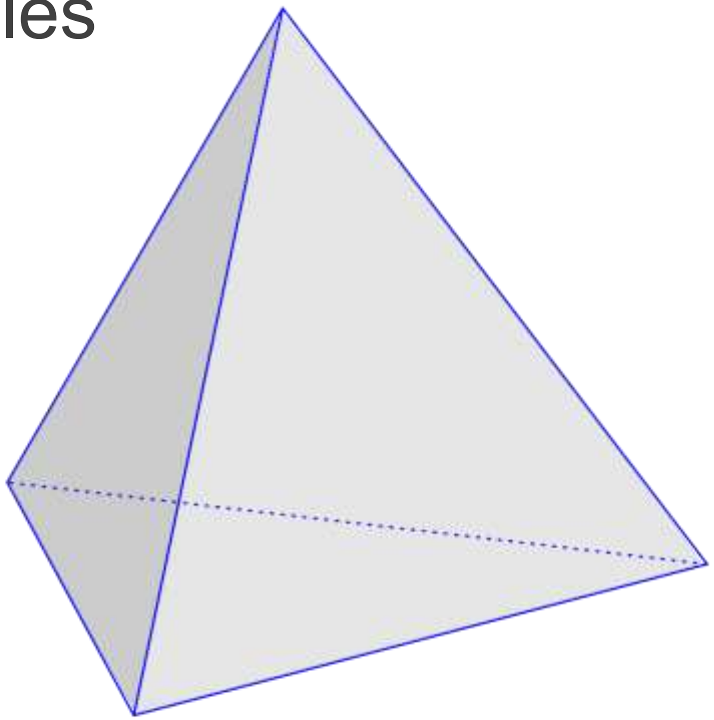


Example2.

Drawing many tetrahedra on 3D space

Creating Tetrahedra Data

- A tetrahedron is made by 4 triangles, and a triangle is made by 3 vertices
- We'll create a tetrahedron by creating 4 triangles
  - Need (4 triangles)(3 vertices/triangle)
- We'll create many tetrahedra by recursively dividing each edge of tetrahedron
  - Create 4 half-length tetrahedra from 1 tetrahedron
- Assign different color to each vertex



Creating Tetrahedra Data

- Need to determine how much storage is required
- Position and color of each point are defined as vec3

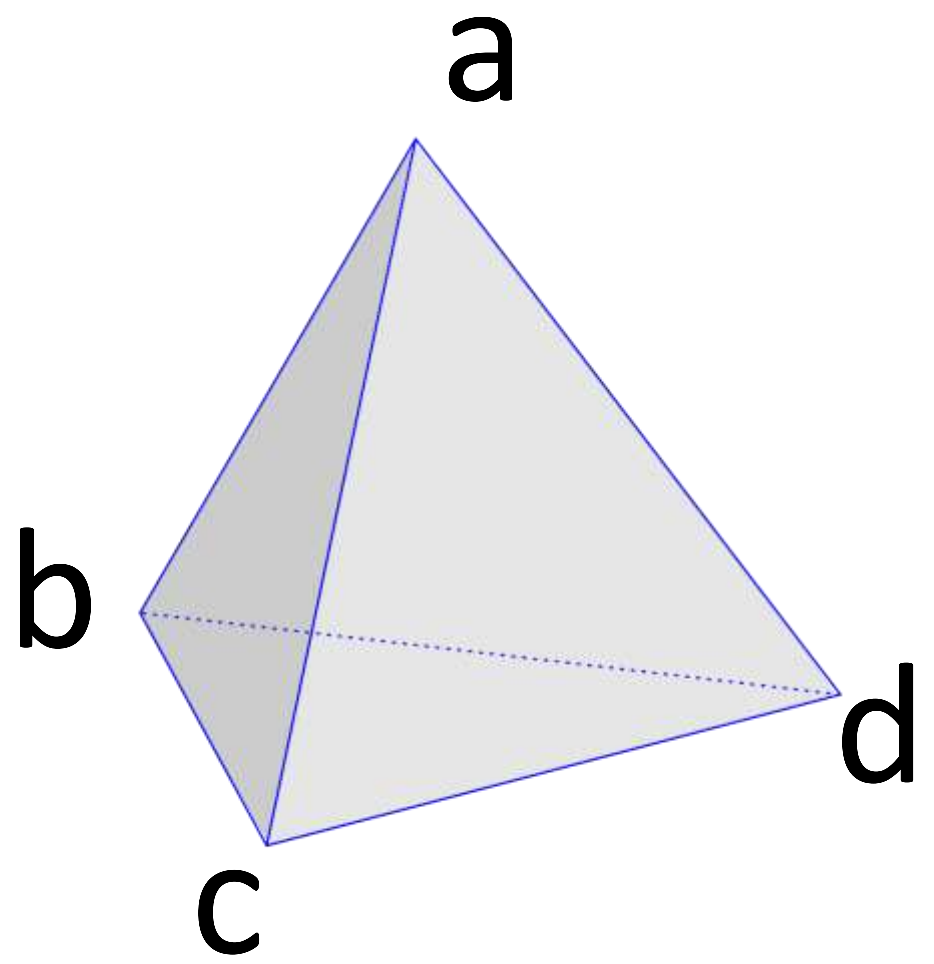
```
const int NumTimesToSubdivide = 4;
const int NumTetrahedra = 256; //4^4
const int NumTriangles = 4 * NumTetrahedra;
const int NumVertices = 3 * NumTriangles;

vec3 points[NumVertices];
vec3 colors[NumVertices];

int Index = 0;
```

Creating Tetrahedra Data


```
// Create a triangle from three vertices
void triangle(const vec3 &a, const vec3 &b, const vec3 &c, const int
color)
{
    static vec3  base_colors[] = {
        vec3(1.0, 0.0, 0.0),
        vec3(0.0, 1.0, 0.0),
        vec3(0.0, 0.0, 1.0),
        vec3(0.0, 0.0, 0.0)
    };

    points[Index] = a;  colors[Index] = base_colors[color]; Index++;
    points[Index] = b;  colors[Index] = base_colors[color]; Index++;
    points[Index] = c;  colors[Index] = base_colors[color]; Index++;
}
```

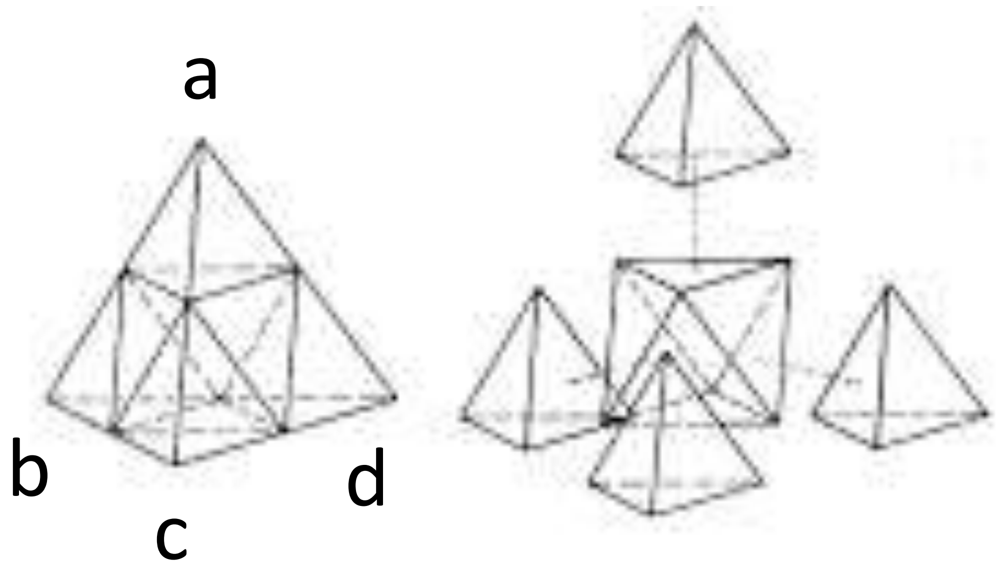
Creating Tetrahedra Data


```
// Create a tetrahedron from 4 triangles
void tetra( const vec3& a,
           const vec3& b,
            const vec3& c,
            const vec3& d )
{
    triangle( a, b, c, 0 );
    triangle( a, c, d, 1 );
    triangle( a, d, b, 2 );
    triangle( b, d, c, 3 );
}
```



Creating Tetrahedra Data

```
// Divide a tetrahedron recursively
void divide_tetra( const vec3& a, const vec3& b,
                const vec3& c, const vec3& d, int count )
{
    if ( count > 0 ) {
        vec3 v0 = ( a + b ) / 2.0;
        vec3 v1 = ( a + c ) / 2.0;
        vec3 v2 = ( a + d ) / 2.0;
        vec3 v3 = ( b + c ) / 2.0;
        vec3 v4 = ( c + d ) / 2.0;
        vec3 v5 = ( b + d ) / 2.0;
        divide_tetra( a, v0, v1, v2, count - 1 );
        divide_tetra( v0, b, v3, v5, count - 1 );
        divide_tetra( v1, v3, c, v4, count - 1 );
        divide_tetra( v2, v4, v5, d, count - 1 );
    }
    else {
        tetra( a, b, c, d );
    }
}
```



Vertex Shader (vshader.glsl)

한양대학교 ERICA
소프트웨어융합대학
COLLEGE OF COMPUTING

```
#version 330

in vec4 vPosition;
in vec3 vColor;
out vec4 color;

void main()
{
	gl_Position = vPosition;
	color = vec4(vColor, 1.0);
}
```

Fragment Shader (fshader.glsl)


```
#version 330

in vec4 color;
out vec4 fColor;

void main()
{
	fColor = color;
}
```

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with VAO
- We load both vertex and color data to the buffer object
  - Place vertex data at the start of the buffer and append color data

V0	V1	V2	…	Vn	C0	C1	C2	…	Cn

  - Separate allocating a buffer and loading data into the buffer

`glBufferData ()` – allocate a buffer (for vertex and color)

`glBufferSubData()` – load data

VBOs in Code

```
//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);
```

Connecting Vertex Shader with Geometry

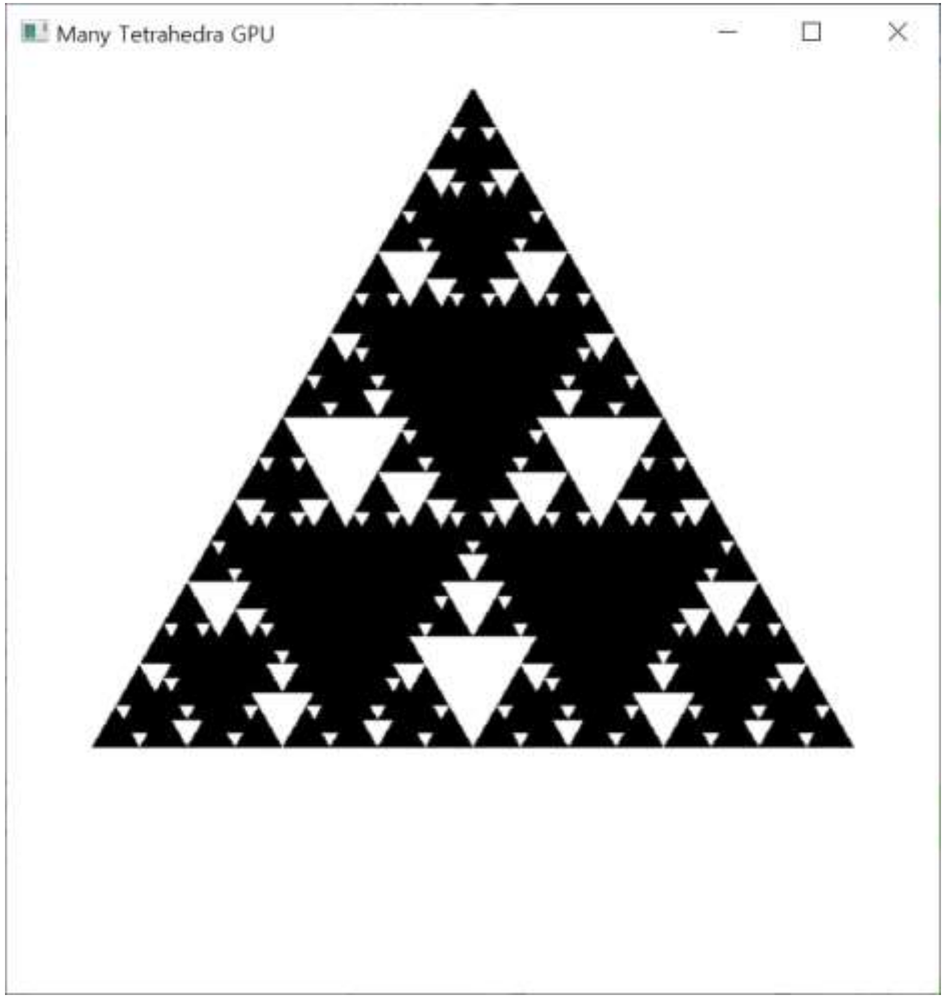
- The new vertex shader takes two input per vertex (vPosition, vColor)
- Need to connect vertex data to shader variables
  - Requires knowing the attribute location
- Attribute location can be queried by calling `glGetVertexAttribLocation()`

```
#version 330

in vec4 vPosition;
in vec3 vColor;
out vec4 color;

void main()
{
        gl_Position = vPosition;
        color = vec4(vColor, 1.0);
}
```

Vertex Array Code

```
//initialize vertex position attribute from vertex shader
    GLuint vPos = glGetAttribLocation(program, "vPosition");
    glEnableVertexAttribArray(vPos);
    glVertexAttribPointer(vPos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));


    GLuint vCol = glGetAttribLocation(program, "vColor");
    glEnableVertexAttribArray(vCol);
    glVertexAttribPointer( vCol, 3, GL_FLOAT, GL_FALSE, 0,
                           BUFFER_OFFSET(sizeof(points)) );
```

Let’s Run Program!



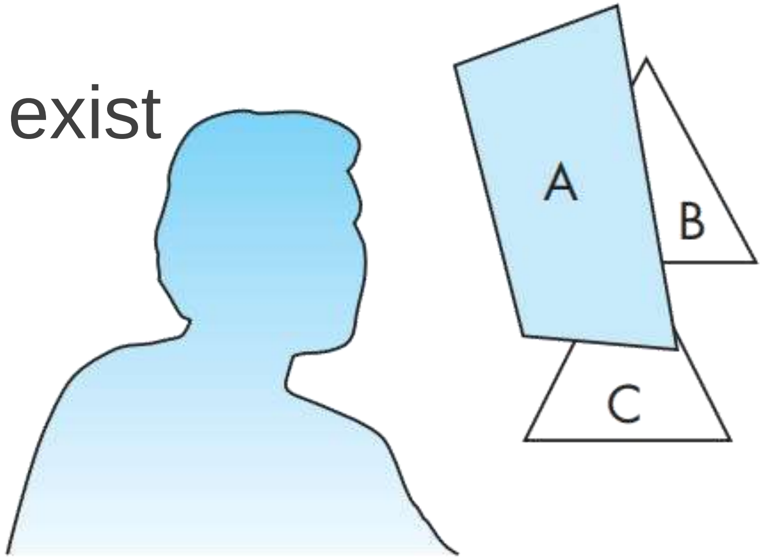
Why???

Hidden Surface Removal

- Draw objects that are only visible to our eyes
- Hidden surface removal
  - Remove the surface that are hidden by other surfaces
- Various hidden surface removal

  (or visible surface detection) algorithms exist
- OpenGL supports the z-buffer algorithm

  for removing hidden surfaces



The hidden-surface problem

Hidden Surface Removal in the code

- Initialize GLUT window mode

```
glutInitDisplayMode (GLUT_RGBA | GLUT_DEPTH);
```

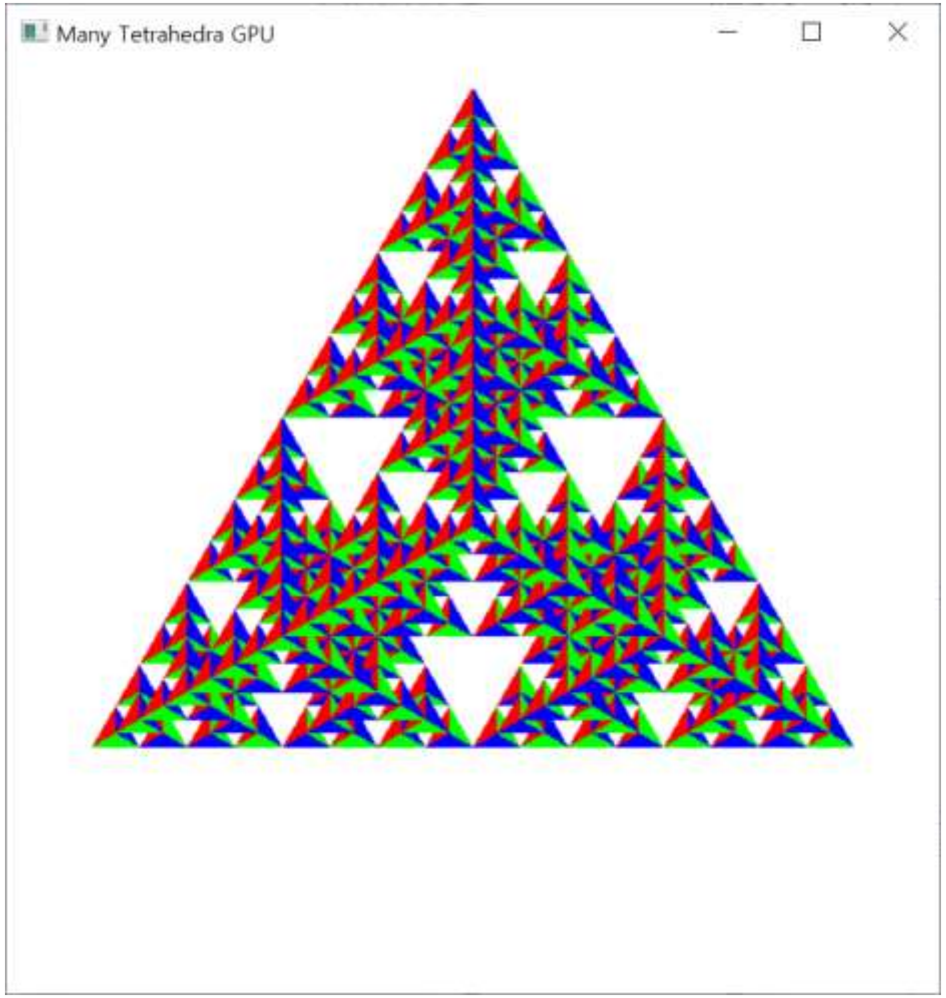
- Enable usage of the z-buffer

```
glEnable(GL_DEPTH_TEST);
```

- Clear depth information from previous rendering

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Let’s Run Program Again!



Example2-1.

Drawing many tetrahedra on 3D space (with graded colors)

Vertex Color Revisited

- Assign color to each vertex as follows:

$$r = \frac{1+x}{2}$$
$$g = \frac{1+y}{2}$$
$$b = \frac{1+z}{2}$$

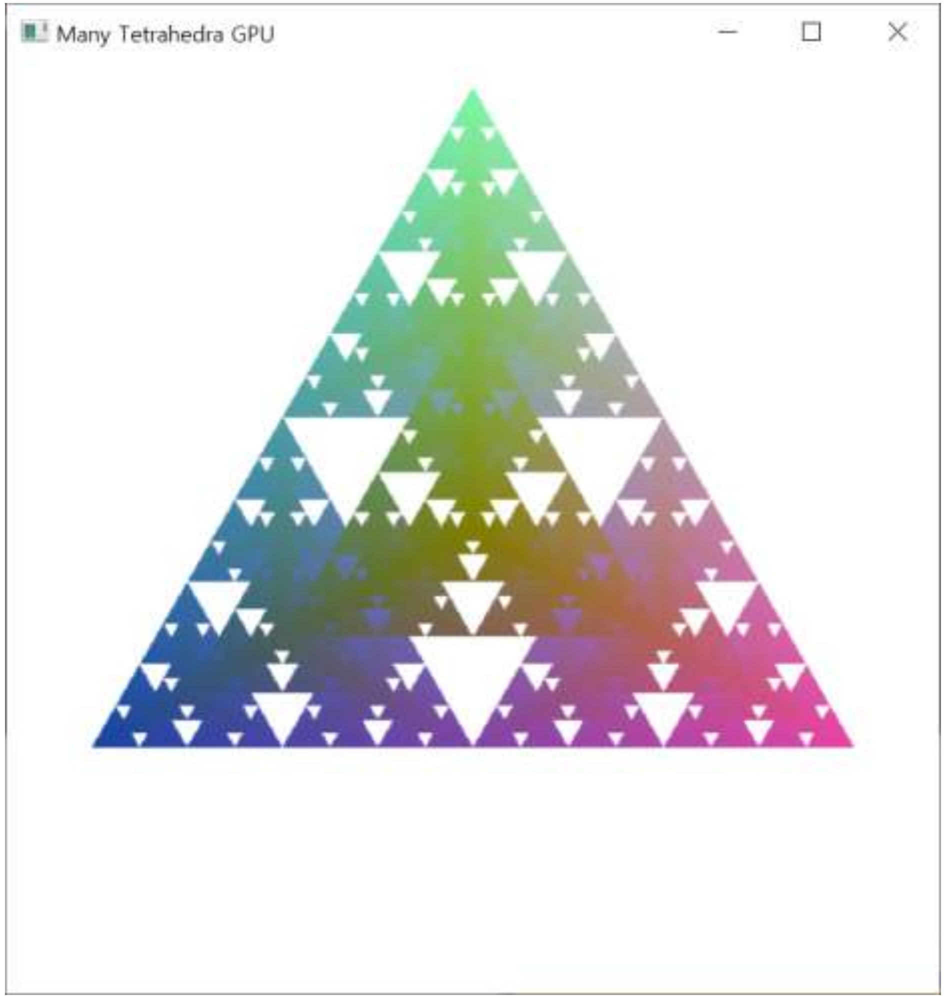
  - Color of a vertex depends on the position of the vertex
  - Do we need to store color of all vertices in memory?

Vertex Shader Revisited

```
#version 330

in vec4 vPosition;
out vec4 color;

void main()
{
	gl_Position = vPosition;
	color = vec4((1.0 + vPosition.x)/2.0,
				(1.0 + vPosition.x)/2.0,
				(1.0 + vPosition.x)/2.0, 1.0);
}
```

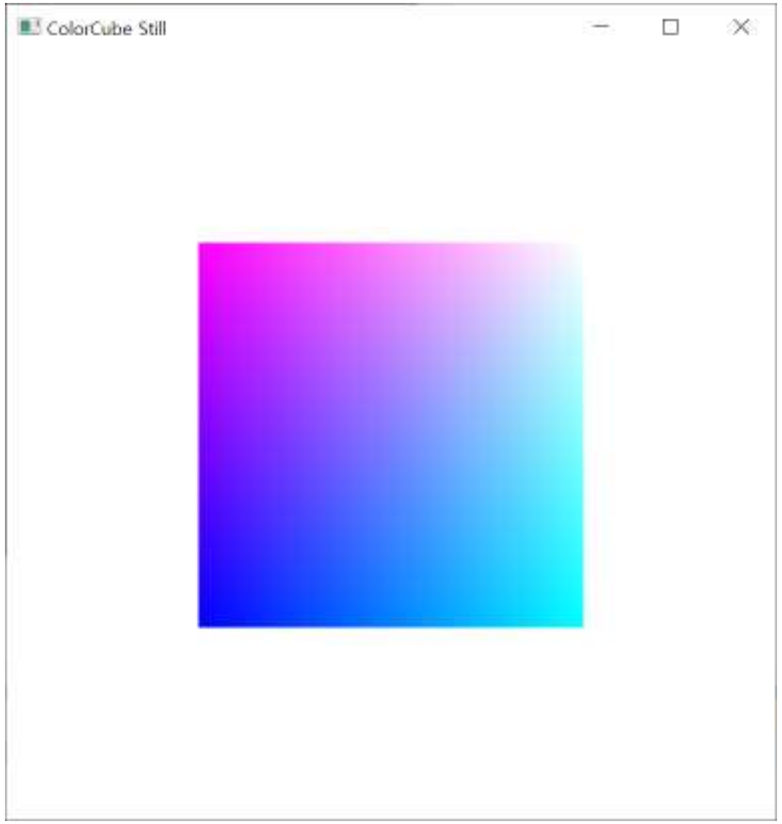
- Modify the application codes accordingly!

Let’s Run Program!



Exercise

- Draw a color cube?
  - 8 vertices are assigned with different colors



Summary

- Shader Programming
  - Creating data (in an array)
  - Sending the data to GPU
    - VAO – vertex array object
    - VBO – vertex buffer object
  - Loading the shaders (vertex/fragment)
  - Draw it with `glDrawArrays(…)`

In Next Practice

- OpenGL GLSL syntax
- More callback functions on GLUT
- Apply transformation to objects