2. OpenGL and Shaders

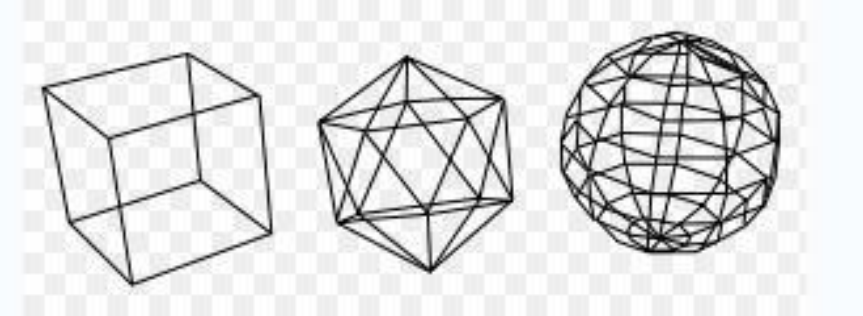
- What is openGL?
- OpenGL, open Graphics Library, is a multi-platform 3D graphics API that incorporates both hardware and software.
  - consists of about 150 commands to be used to create interactive 3D applications.
  - maintained by the company The Khronos Group (since 2006)
  - All Windows versions support openGL (run the software ***GLView*** to find out the version of openGL your machine supports)

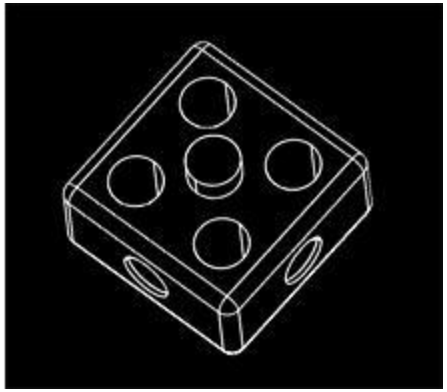
Advantages:

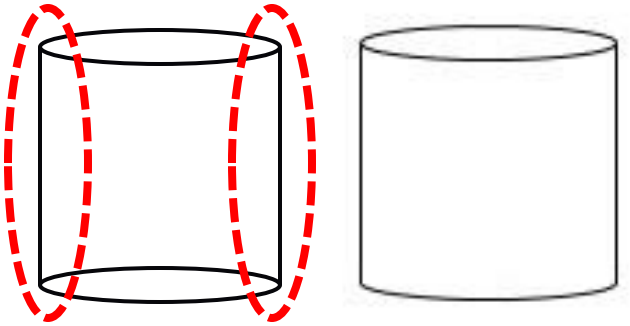
- Developed on top of X Windows so it inherits many good features of X Windows such as
  - **Device-independence, API to graphics hardware**
  - **Event driven (when the user presses a particular button of the mouse, say the left button, the event (left button pressed) and the measure (location of the cursor) are put into a queue)**
- High end
  - You don't have to write your code for most of the applications (OpenGL can do most of them for you
- 3D-oriented
  - Developed for 3D applications, 2D is a special case of 3D (in what sense? e.g., z=0)

Things OpenGL can do:

- Wireframe models (2D & 3D wireframe drawings)



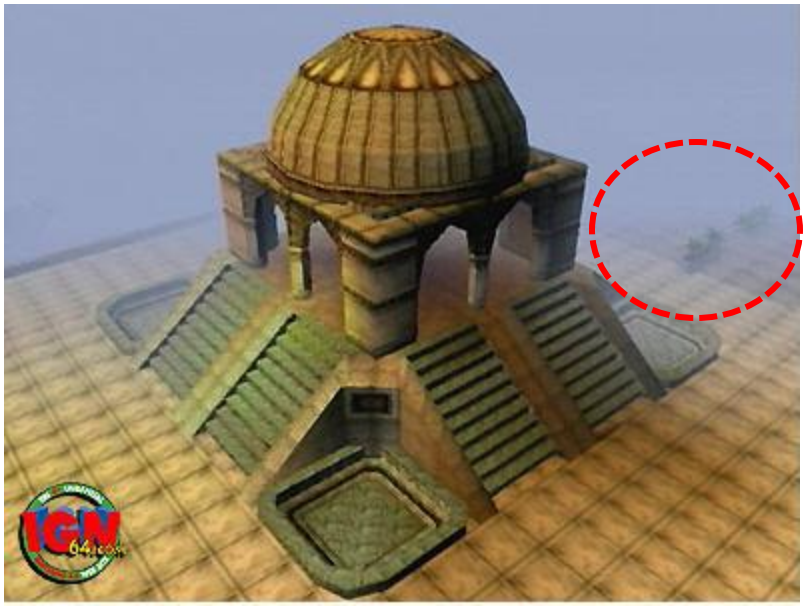




Multiplied by a percentage value depending on its distance from the viewer

Things Open[illegible]an do:

- Depth-cuing effect
  - lines farther from the eye are dimmer; when we do scan conversion of a line/polygon, the intensity of a point considers the effect of depth



CS Dept, UK

Has less zigzag effect

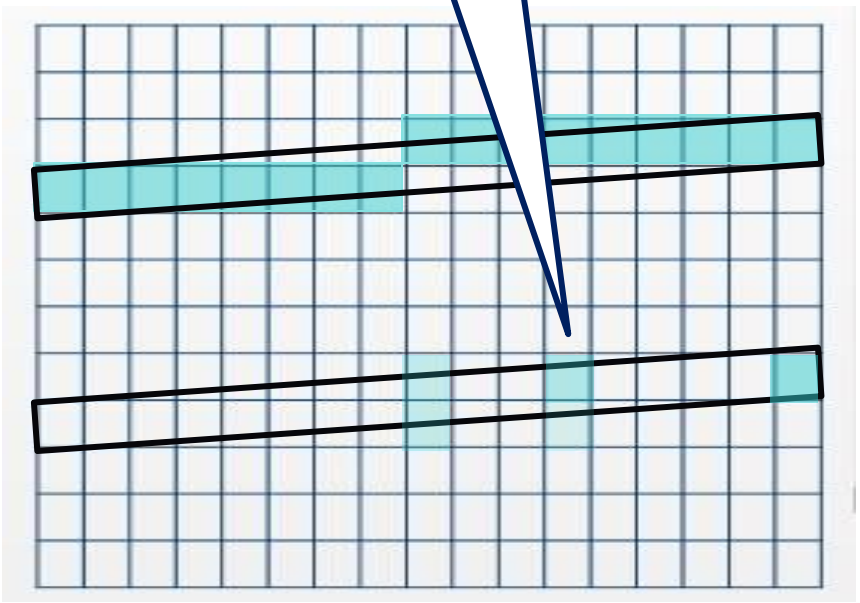
Bresenham's Line Algorithm

Things OpenGL can do:

- Anti-aliased lines
  - the intensity of a point is proportional to the area of the pixel covered by the line polygon)

Not anti-aliased

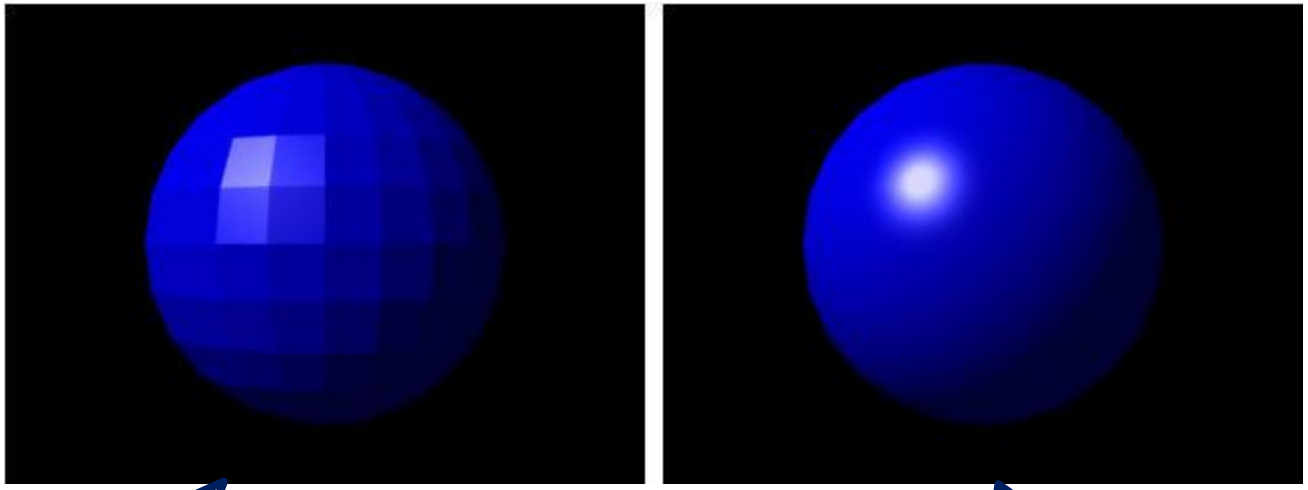
Anti-aliased



Geometrical structures are the same

Things OpenGL can do:

- Flat-shaded vs smooth-shaded polygons



Each polygon has only one intensity

Intensity is interpolated

Things OpenGL can do: (conti)

- Shadows and textures (2D or 3D)

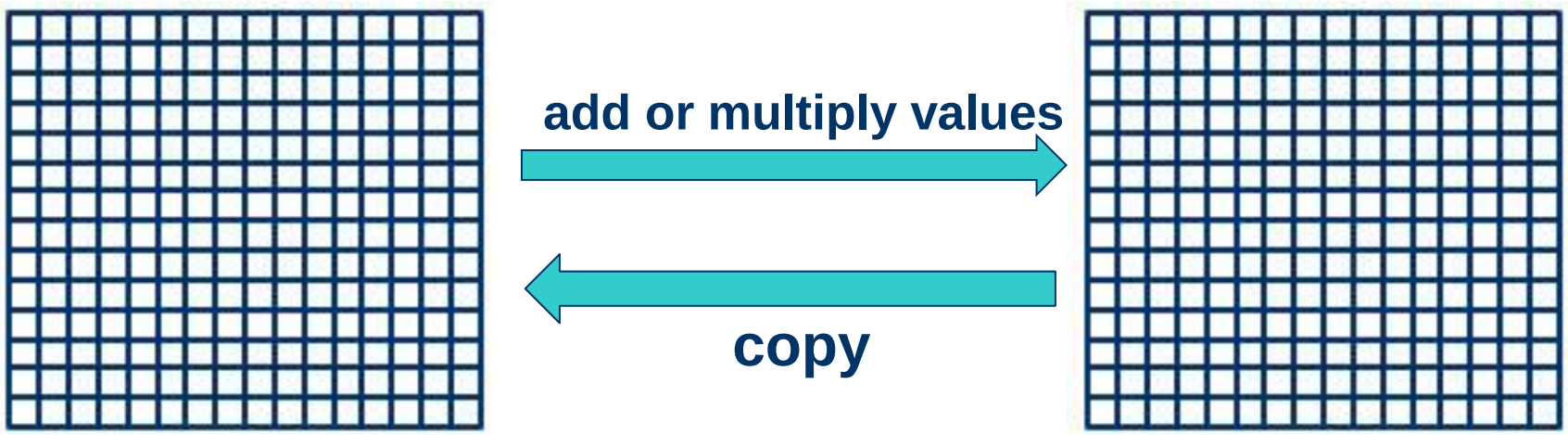
A point is in shadow if it is visible to the view point, but not to the light source

For each visible point, find out which value of the given texture to use

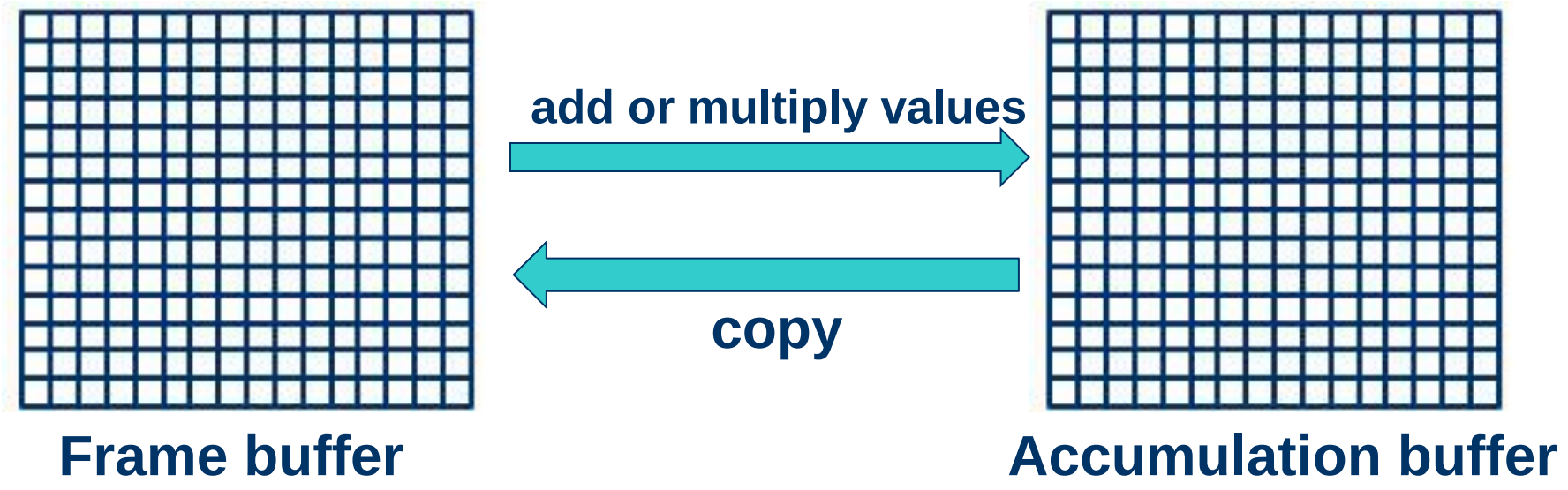
openGL has 4 buffers: color buffer (frame buffer), depth buffer (Z-buffer), accumulation buffer, stencil buffer

Things OpenGL can do: (conti)

- Motion-blurred objects
  - OpenGL has a special buffer called the accumulation buffer, it is used to compose a sequence of images needed to blur the moving object (double buffering, good for animation)




```
glAccum(GL_LOAD, value);
glAccum(GL_RETURN, value);
glAccum(GL_ACCUM, value);
glAccum(GL_ADD, value);
glAccum(GL_MULT, value);
```



For instance:

```
glClear(GL_ACCUM_BUFFER_BIT);
for (i=0; i < num_mages; i++)
{
    glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT)
    display_image(i);
    glAccum(GL_ACCUM, 1.0/(float) num_images);
}
glAccum(GL_RETURN, 1.0);
```

// show average of the 5 previous frames
// if num_images == 5

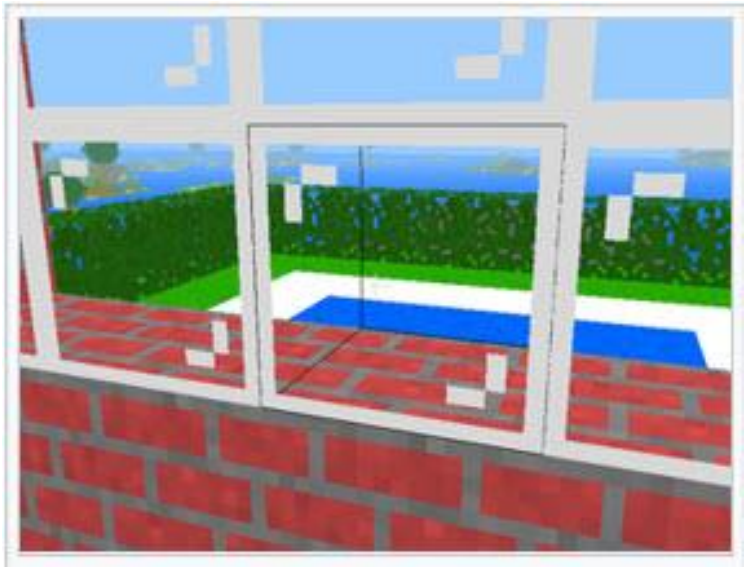
Things OpenGL can do: (conti)

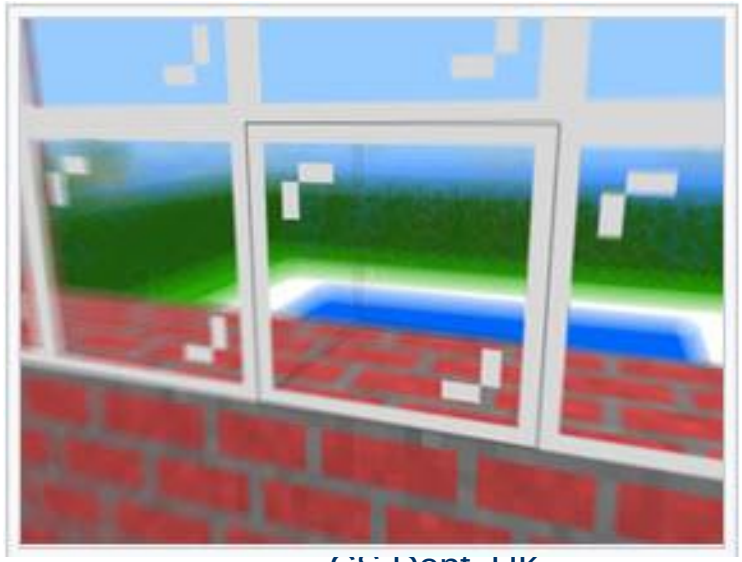
Using a technique similar to depth-cuing

- Atmospheric effect (fog)
  - to simulate a smoke-filled room/space

Things OpenGL can do: (conti)

- Depth-of-the-field effect
  - Objects drawn with jittered viewing volumes into the accumulation buffer for a depth-of-the-field effect





Things OpenGL can do: (conti)

Note that in this class,

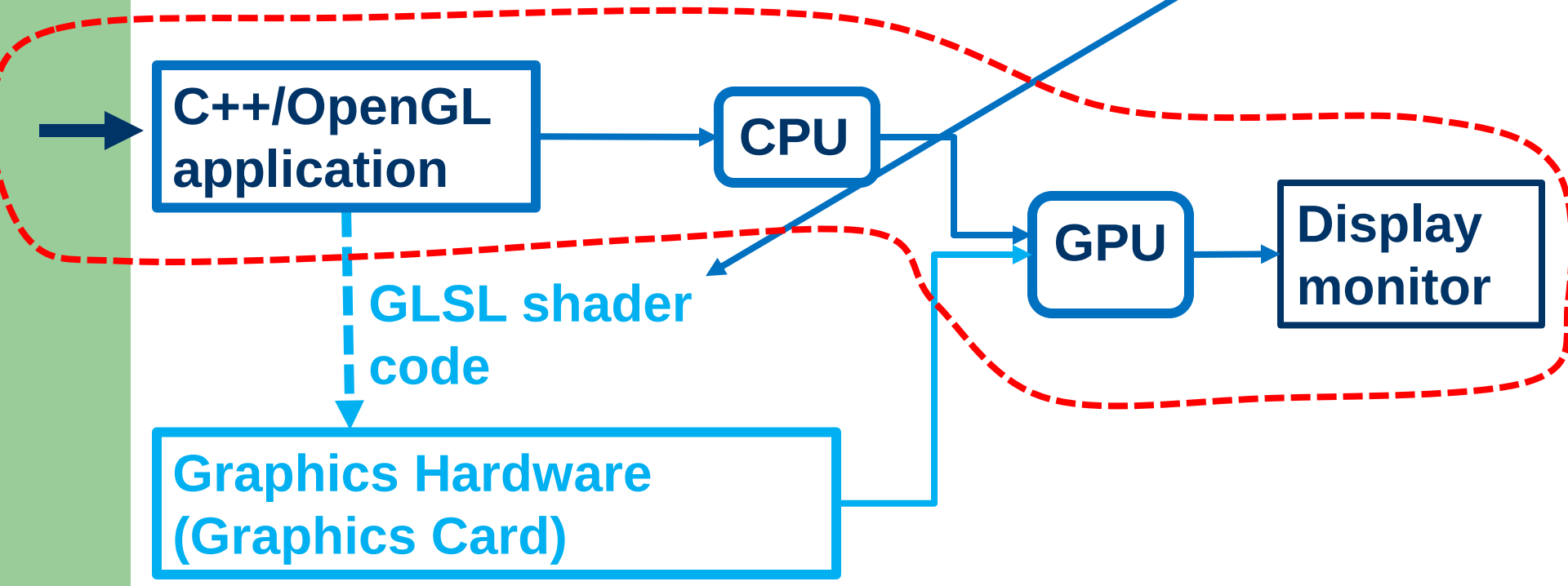
- we use openGL calls for tasks related to 3D rendering
- We will also use several additional libraries:
  - GLEW (openGL extension Wrangler (to determine which openGL extensions are supported on the target platform))
  - GLM (openGL Mathematics)
  - SOIL2 (Simple OpenGL Image Loader (to do texture mapping))
  - GLFW (Graphics Library Framework (to create glfw window to draw 3D scenes))

2.1 Overview

- OpenGL has been extensively extended
- On the hardware side, modern openGL provides a multi-stage *graphics pipeline* that is partially programmable using a language called GLSL (*OpenGL Shading Language*)
- On the software side, openGL's API is written in C, and thus the calls are directly compatible with C and C++. C++ is the most popular language choice to include openGL calls (referred as a *C++/OpenGL application*)

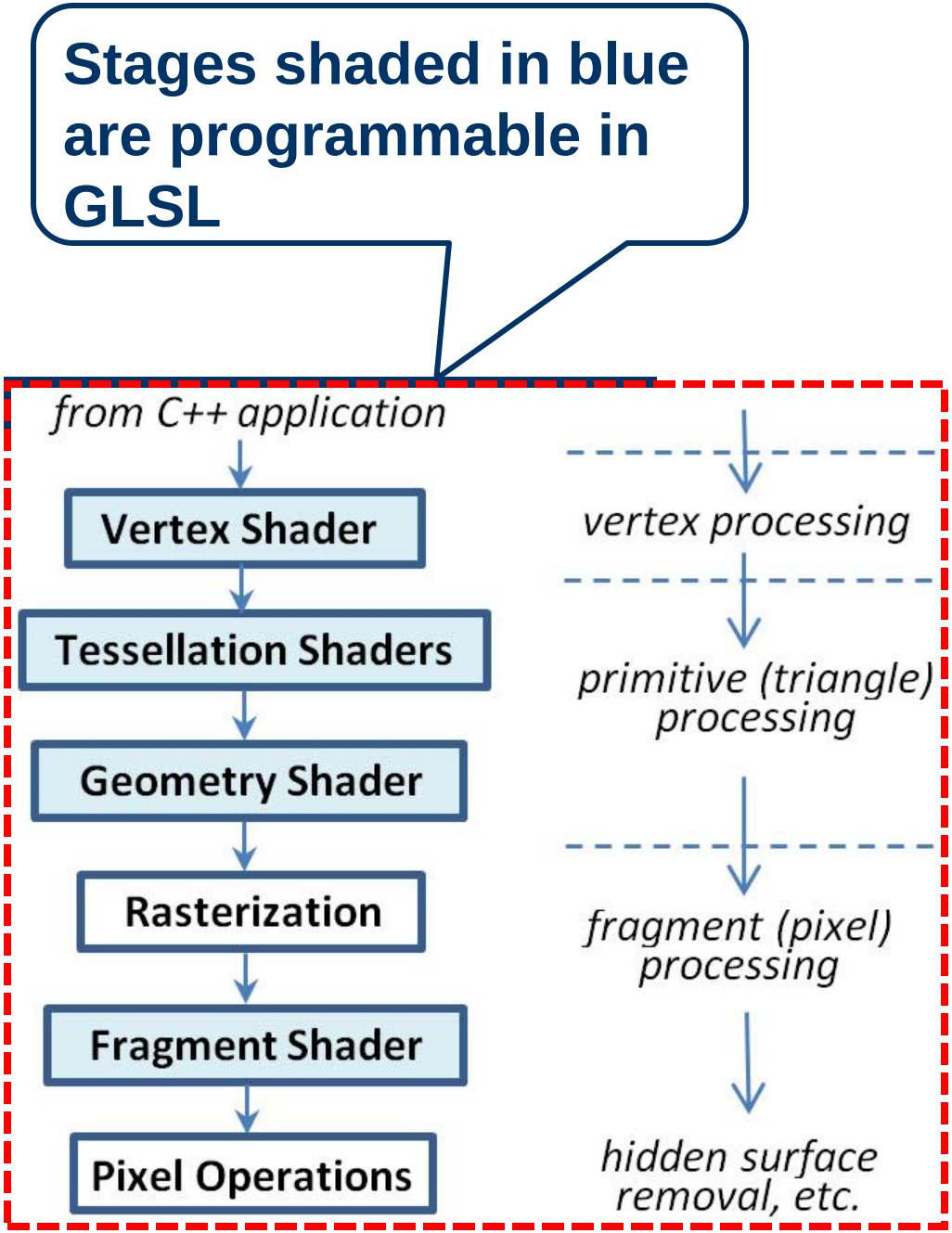
An overview of a modern C++/OpenGL application

The task of a *C++/OpenGL application* that contains GLSL code to install the GLSL code onto the graphics card



2.1 Overview

- Modern 3D graphics programming utilizes a *pipeline.*
- The process of converting a 3D scene to a 2D image is broken into a series of steps, as the figure shown on the right side.

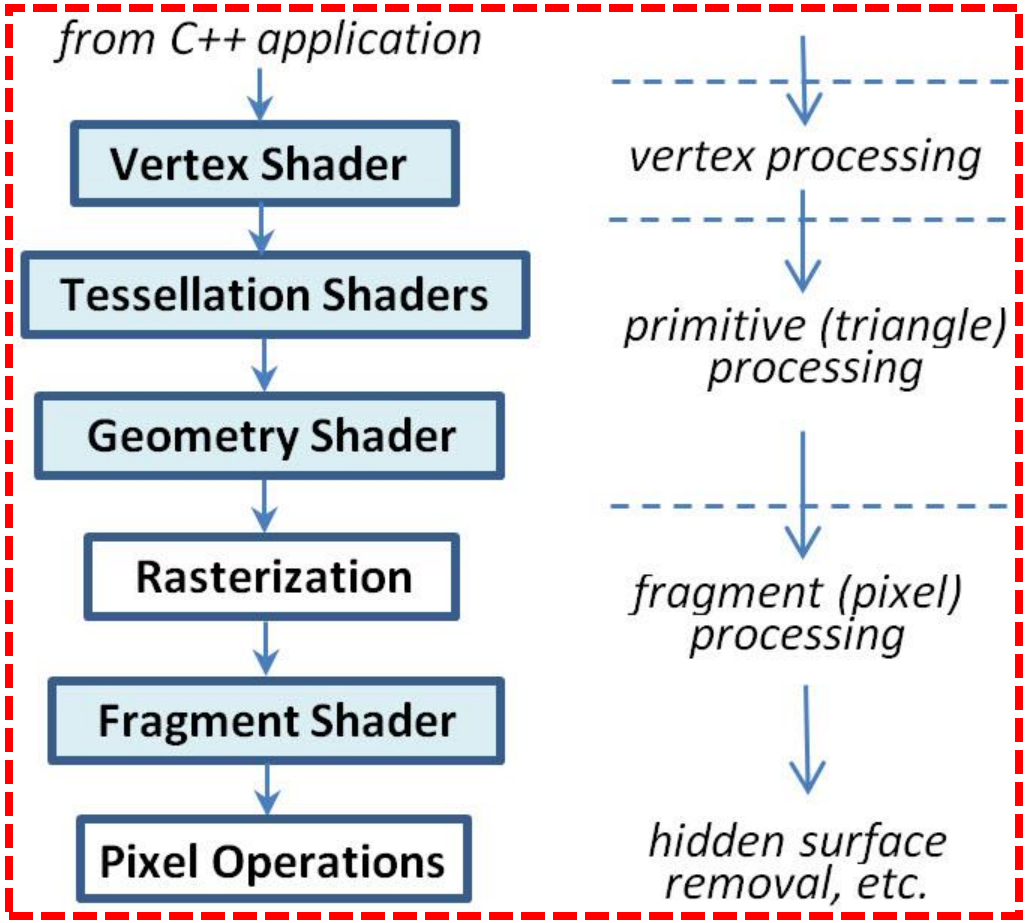


2.1 Overview

- Loading GLSL programs into the shaders is the job of the C++/OpenGL application and is done as follows:
  - It uses C++ to obtain the GLSL shader code, either from text file or hard-coded as strings
  - It then creates openGL shader objects and loads the GLSL shader code into them
  - Finally, it uses OpenGL commands to compile and link objects and install them on the GPU

2.1 Overview

- Usually it is necessary to provide GLSL code for the *vertex* and *fragment* shaders
- The *tessellation* and *geometry* stages are optional.



2.2 Basic Structure of OpenGL Programs

Initialization Procedures

Display and event handling Functions

```
int main (void)
{
```

Window and coordinate system creation

Libraries Initialization

Event handling functions registration

(Infinite) Event Handling Loop

Terminate and Exit

```
}
```

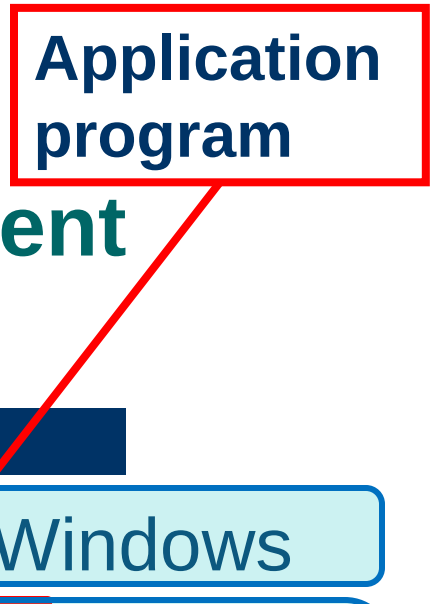
Structure of Old OpenGL Programs

- Event handling functions
- Initialization
  - window and coordinate system creation
  - libraries initialization
- Registration
  - Let the system know which event handling function should be invoked when an event occurs
- Infinite Event Handling Loop
  - Entering (infinite) event handling loop
- Terminate and Exit

Classical (X Windows based) event handling approach:

```
void main ( ) {
    ....
    while ( 1 ) {
        XNextEvent ( display, &event );
        switch  ( event.type ) {
            case  KeyPress:
                { event handler for a keyboard event }
                break;
            case  ButtonPress:
                { event handler for a mouse event }
                break;
            case Expose:
                { event handler for an expose event }
                break;
            ....
        }
    }
}
```

Classical (X Windows based) event handling approach



- Event queue is maintained by the X Windows
- But handling of the events is your job
  - A statement like "case KeyPress" is like a **callback func**tion registration
- The entire structure then is replaced with one instruction:

  glutMainLoop( )

  But you don't see it.

2.3 A Modern OpenGL Example

```
////////////////////////////////////////////////////////////////////////////////
// Program 2.1
// A GLFWwindow is instantiated and the background is set to red
////////////////////////////////////////////////////////////////////////////////
```

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window) { }

void display(GLFWwindow* window, double currentTime) {
    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

Application-specific initialization tasks are placed here

We place code that draws to the GLFWwindow here

2.3 A Modern OpenGL Example

Initialize libraries and create glfw window

Means the machine must be compatible with openGL version 4.3

```
int main(void) {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 400,
                    "Chapter 2 - program 1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);
```

2.3 A Modern OpenGL Example

```
while (!glfwWindowShouldClose(window)) {
    display(window, glfwGetTime());
    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwDestroyWindow(window);
glfwTerminate();
exit(EXIT_SUCCESS);
}
```

Output of the program:

2.3 A Modern OpenGL Example (conti)

- glClearColor(1.0, 0.0, 0.0, 1.0) : set background color to red and the opacity component to 1
- glClear(GL_COLOR_BUFFER_BIT) : fill all the color buffers with the specified clear (background) color
- glfwWindowHints( ) : sets hints for the next call to glfwCreateWindow()
- gefwCreateWindow(w, h, xxxx, m, s) : creates a glfw window with width w, height h, title “xxxx”, ...

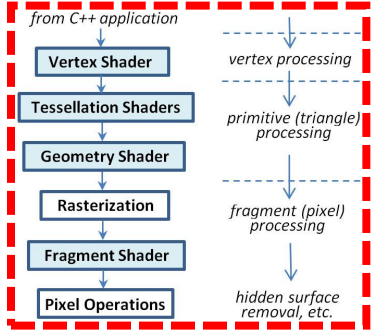
2.3 A Modern OpenGL Example (conti)

All drawing happens in the current context.

Number of screen updates

- glfwMakeContextCurrent(window) : makes the OpenGL or OpenGL ES context of the specified window current on the calling thread
- glfwSwapInterval(1) : sets the swap interval for the current OpenGL or OpenGL ES context to 1
- glfwSwapBuffers(window) : swaps the front and back buffers of the specified window
- glfwPollEvents() : processes all pending events

2.4 OpenGL Graphics Pipeline
2.4.1 Vertex and Fragment Shaders



- Our 1st openGL program didn't actually draw anything. To actually draw something, we need to include a *vertex shader* and a *fragment shader*
- The C++/openGL application must compile and link appropriate GLSL vertex and fragment shader programs, and load them into the pipeline
- All vertices of openGL primitives (points, lines, and triangles) pass through the *vertex shader*
- To display vertices and triangles, we need to provide a *fragment shader*

2.4.1 Vertex and Fragment Shaders

```
////////////////////////////////////////////////////////////////////////////////
// Program 2.2  Using shaders to draw a point
// …. #includes are the same as before
////////////////////////////////////////////////////////////////////////////////
#define numVAOs 1;

GLuint renderingProgram;
GLuint vao[numVAOs];

GLuint createShaderProgram() {
    const char *vshaderSource =
        "#version 430    \n"
        "void main(void) \n"
        "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";
    const char *fshaderSource =
        "#version 430    \n"
        "out vec4 color; \n"
        "void main(void) \n"
        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";
```

new declarations

Declares a vertex shader as a char string

Declares a fragment shader as a char string

2.4.1 Vertex and Fragment

Generate a shader object

```
GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
GLuint vfprogram = glCreateProgram();

glShaderSource(vShader, 1, &vshaderSource, NULL);
glShaderSource(fShader, 1, &fshaderSource, NULL);
glCompileShader(vShader);
glCompileShader(fShader);

glAttachShader(vfprogram, vShader);
glAttachShader(vfprogram, fShader);
glLinkProgram(vfprogram);

return vfprogram;
}
```

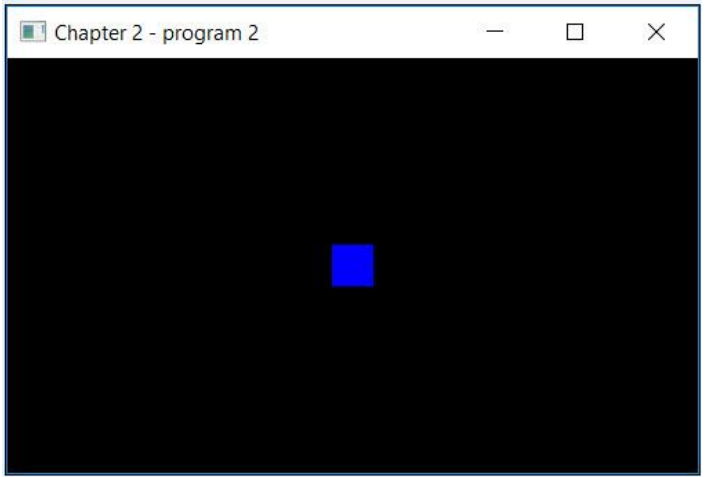
2.4.1 Vertex and Fragmen

See slide 31

```
void init(GLFWwindow* window) {
    renderingProgram = createShaderProgram();
    glGenVertexArrays(numVAOs, vao);
    glBindVertexArray(vao[0]);
}

void display(GLFWwindow* window, double currentTime) {
    glUseProgram(renderingProgram);
    glPointSize(30.0f);
    glDrawArrays(GL_POINTS, 0, 1);
}
```

….. Main( ) same as before



Output of the program:

2.4.1 Vertex and Fragment Shaders

Look at the shaders themselves now

The vertex shader (declared as arrays of strings):

```
#version 430
void main(void)
{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0);  }
```

gl_Position: built-in GLSL variable, set a vertex's coordinate position in 3D space

vec4: GLSL datatype, hold a 4-tuple, representing X, Y, Z and the homogeneous component

2.4.1 Vertex and Fragment Shaders

The fragment shader (declared as arrays of strings):

```
#version 430
out vec4 color;
void main(void)
{ color = vec4(0.0, 0.0, 1.0, 1.0);  }
```

Purpose of a fragment shader: to set the RGB color of a pixel to be displayed

The "out" tag: indicates the variable color is an output

It wasn't necessary to specify an "out" tag for gl_Position in the vertex shader b/c gl_Position is a predefined output variable.

2.4.1 Vertex and Fragment Shaders

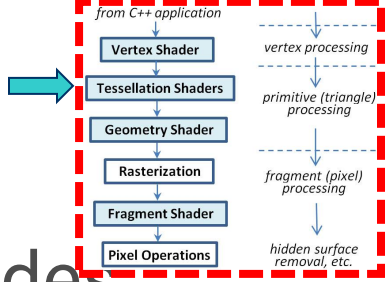
One thing about the init( ) function

```
void init(GLFWwindow*  window) {
    renderingProgram = createShaderProgram();
    glGenVertexArrays(numVAOs, vao);
    glBindVertexArray(vao[0]);
}
```

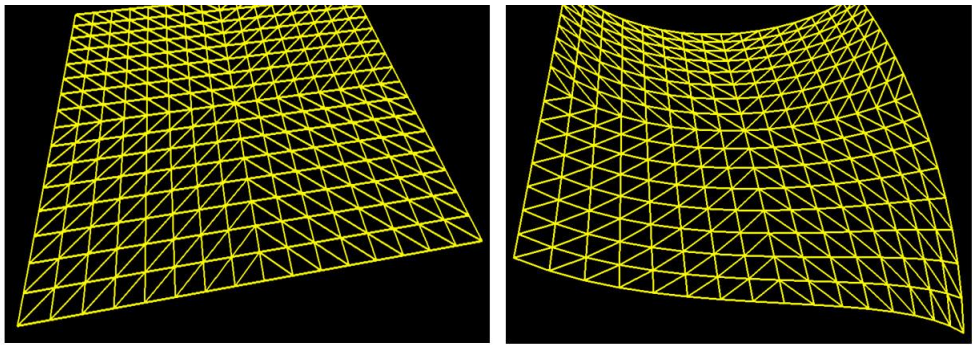
In openGL, when sets of data are prepared for sending down the pipeline, they are organized into *buffers* first, and then in turn organized into *Vertex Array Objects* (VAOs).

Even if *buffers* are not needed, one still needs to create at least one *VAO* whenever shaders are being used. The last two lines are for that purpose.

2.4.2 Tessellation

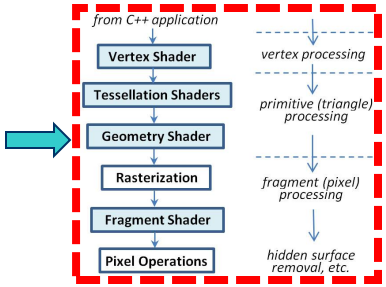


- The programmable tessellation stage provides
  - a *tessellater* that can generate a large number of triangles, usually as a grid
  - some tools to manipulate those triangles
- Useful for generating complex terrain
- Efficient to generate a triangle mesh in GPU hardware than doing it in C++

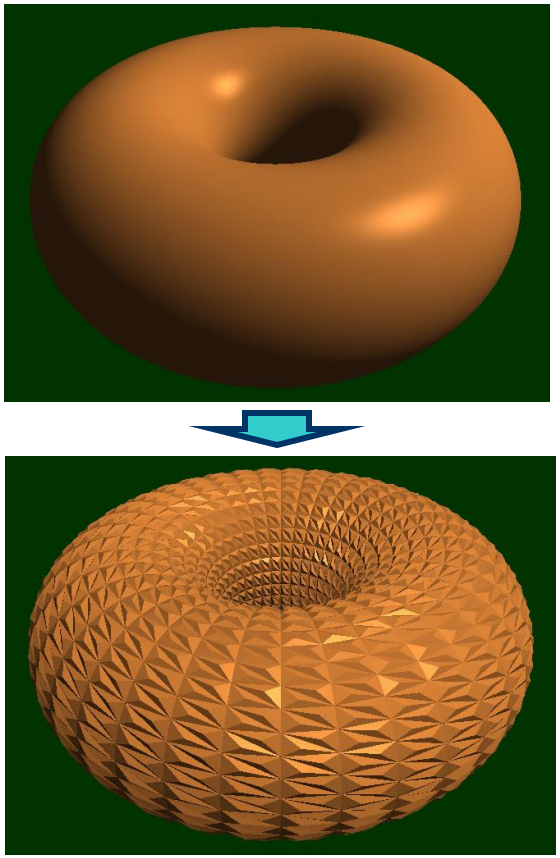


2.4.3 Geometry Shader

Lighting effects included

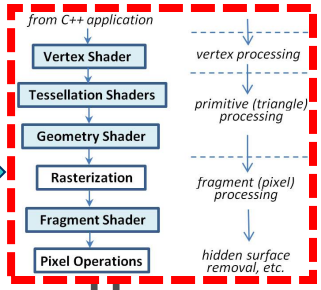


- A *primitive* *is processed at a time instead of a vertex or a pixel*
- Triangle primitives could be
  - altered (stretched or shrunken)
  - deleted (to create “holes” in an object)
- The geometry shader can
  - generate additional primitives
  - add surface texture (bumps, scales, …) to an object



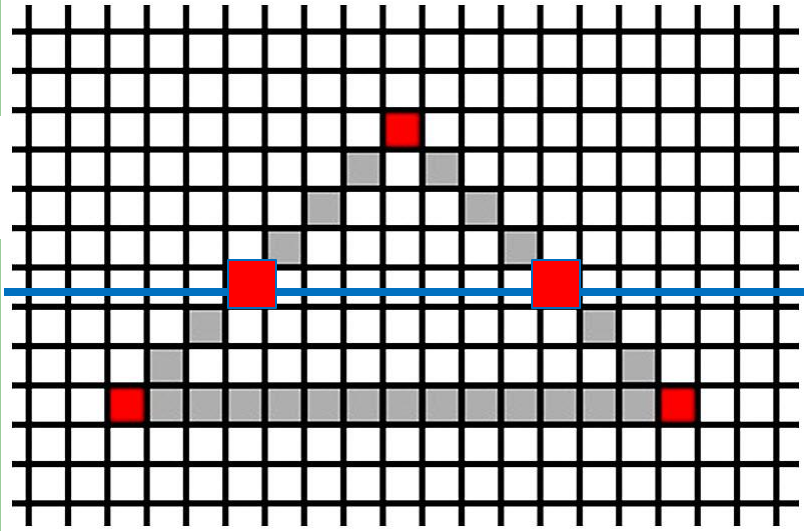
2.4.4 Rasterization

Holds the information associated with a pixel

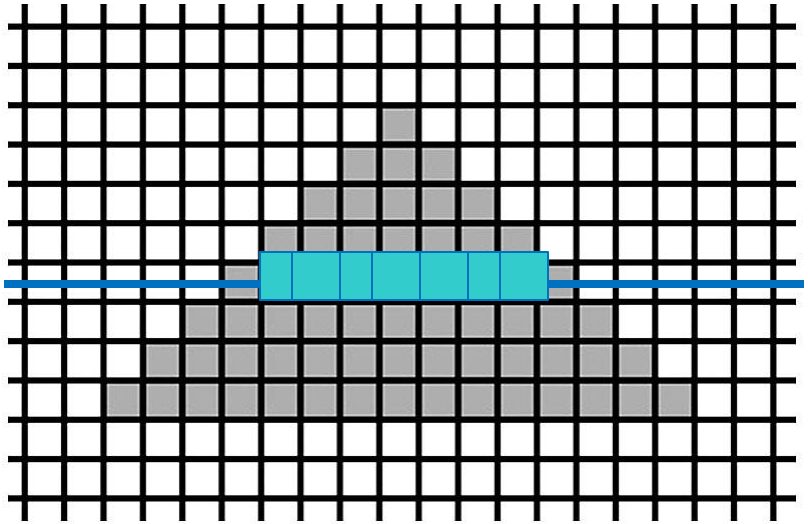


- To have a 3D object displayed on a 2D monitor, the *primitives* in the object (usually triangles) have to be *rasterized* into *fragments* first.
- *Rasterization* determines the locations of pixels that need to be drawn in order to produce the triangle specified by its three vertices

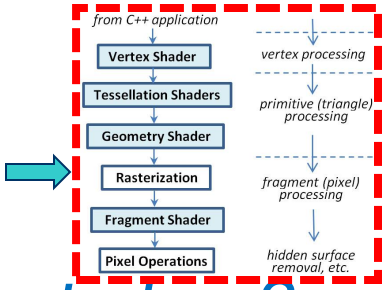
Step 1



Step 2

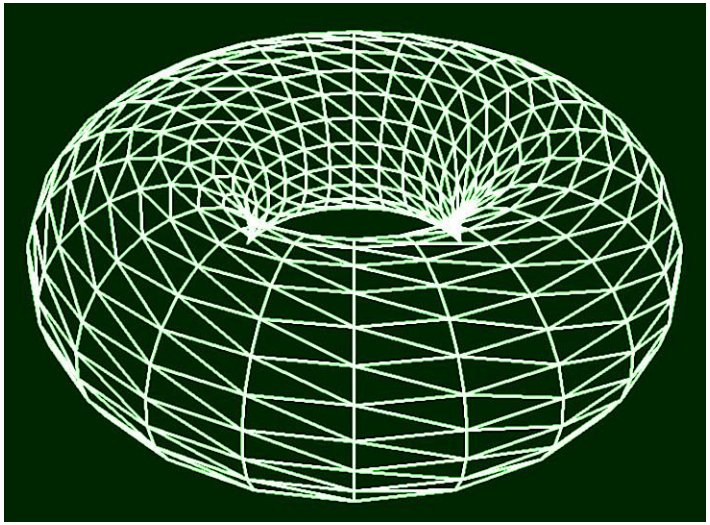


2.4.4 Rasterization

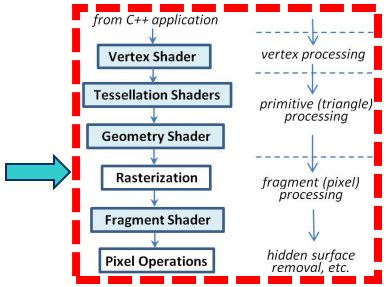


- Rasterization has two options, *with or without step 2 performed*
- Without step 2 performed: only bounding edges of the triangle primitives are rasterized

```
void display ( ) {
      glUseProgram( );
      glPolygonMode(GL_FRONT_AND_BACK,
                          GL_LINE);
      glDrawArrays(  );
}
```

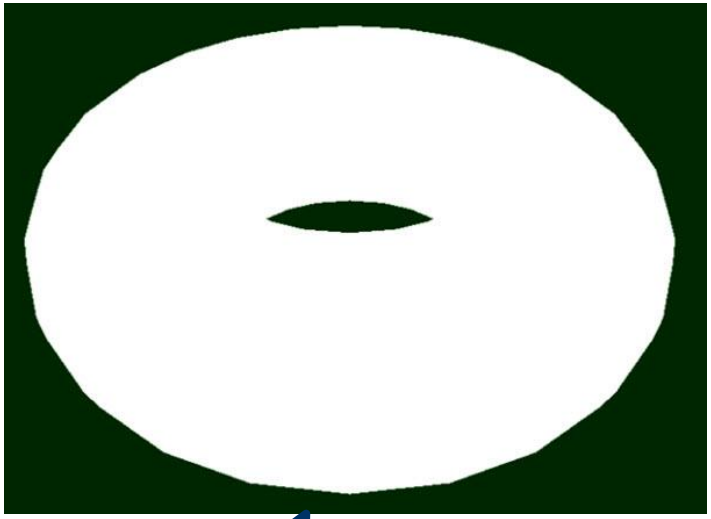


2.4.4 Rasterization



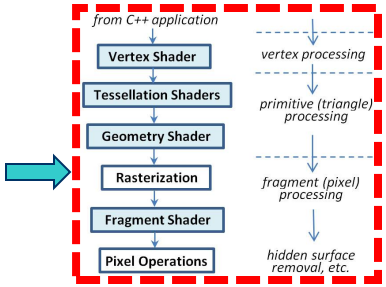
- With step 2 performed: bounding edges and interior of the triangle primitives are both rasterized

```
void display ( ) {
      glUseProgram( );
      glPolygonMode(GL_FRONT_AND_BACK,
                          GL_FILL);
      glDrawArrays(  );
}
```



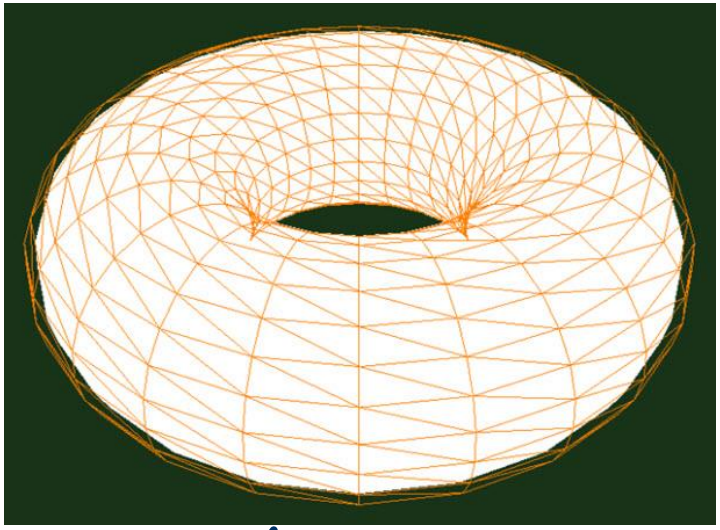
Fully rasterized torus but lighting effects not included

2.4.4 Rasterization



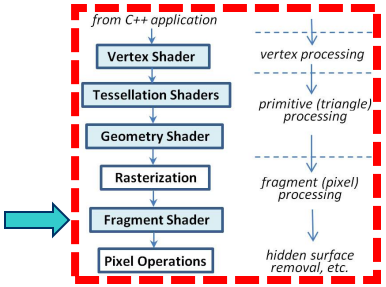
- With and without step 2 performed: with wireframe grid superimposed on fully rasterized primitives

```
void display ( ) {
      glUseProgram( );
      glPolygonMode(GL_FRONT_AND_BACK,
                          GL_FILL);
      glDrawArrays( );
      glPolygonMode(GL_FRONT_AND_BACK,
                          GL_LINE);
      glDrawArrays( );
}
```



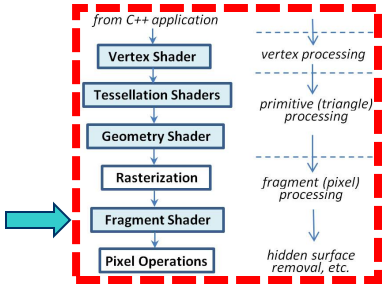
Lighting effects not included

2.4.5 Fragment Shader



- Purpose of fragment shader: to assign colors to the rasterized pixels
- GLSL affords us virtually limitless creativity to calculate colors
- One can use the predefined GLSL variable *gl_FragCoord*, for instance, to set a pixel's color based on its location

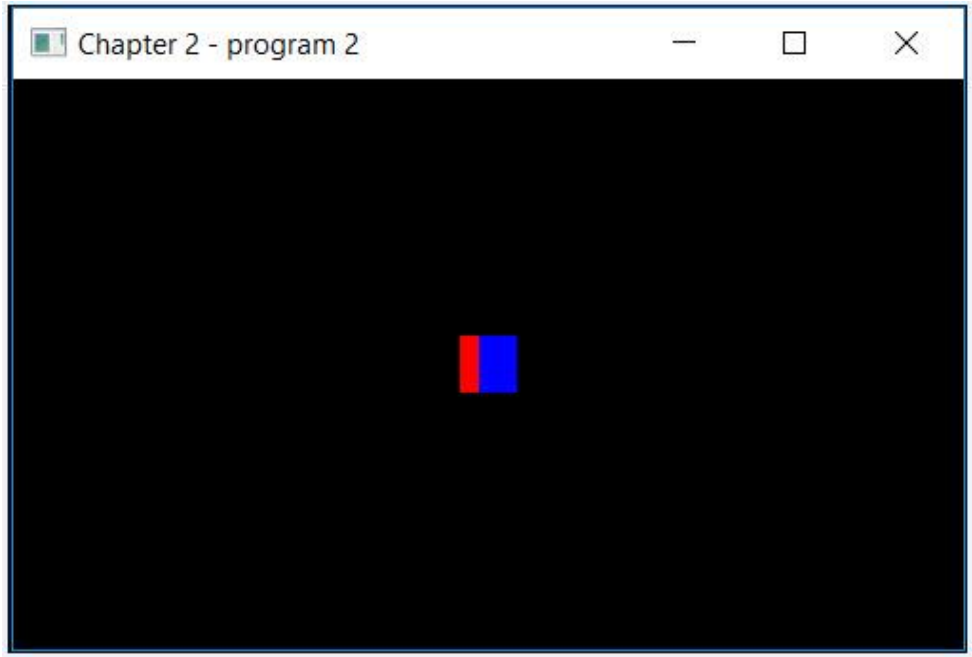
2.4.5 Fragment Shader



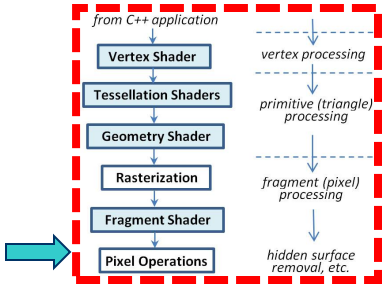

```
#version 430
out vec4 color;
void main(void)
{ if (gl_FragCoord.x < 200) color = vec4(1.0, 0.0, 0.0, 1.0);
  else color = vec4(0.0, 0.0, 1.0, 1.0);
}
```

By modifying example program 2.2

Output of the program:

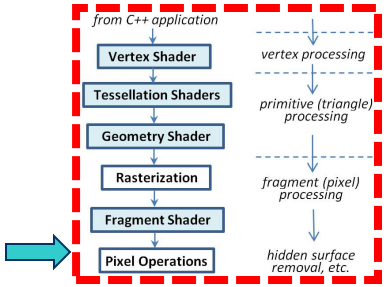


2.4.6 Pixel Operations



- This stage of the openGL graphics pipeline is to determine a *color/intensity* for each entry (pixel) of the *color buffer*
- If a pixel is the projection of several points in the 3D scene, then the color/intensity of that pixel should be set to the color/intensity of the point closest to the viewer
- Several methods (*hidden surface removal algorithms*) can be used to determine which point is the closest point for each pixel. *Z-buffer method* is used by openGL

2.4.6 Pixel Operations



Color buffer

d1

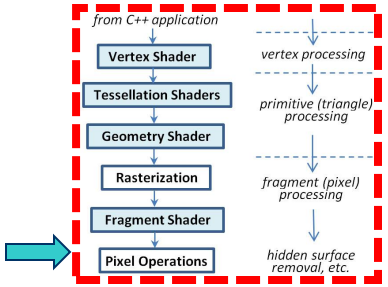
d2

d3

View point

Color/intensity of pixel (x, y) should be computed using information from d1

2.4.6 Pixel Operations



Z-Buffer Method:

- Initially set all entries of the *color buffer* to the back-ground color and all entries of the *depth buffer* to *1*
- When a *pixel color* is output by the *fragment shader*, its distance from the viewer is computed
- If the computed distance is *less than* the distance in the *depth buffer* for that pixel, then the pixel color replaces the color in the color buffer for that pixel, ar the distance replaces the current distance value in th *depth buffer*. Otherwise the pixel is discarded.

2.5 Detecting OpenGL and GLSL Errors

- Debugging GLSL errors is difficult because
  - *GLSL compilation happens at C++ runtime*
  - GLSL code doesn't run on CPU (it runs on GPU), so *the operating system cannot always catch openGL runtime errors*.
- How to catch/display GLSL errors? build utilities to
  - Check the openGL error flag for the occurrence of an openGL error
  - Display the contents of openGL's log when GLSL compilation failed
  - Display the contents of openGL's log when GLSL linking failed

2.5 Detecting OpenGL and GLSL Errors

//

// Program 2.3

// An OpenGL program with modules to catch GLSL errors

//

```
void printShaderLog(GLuint shader) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetShaderInfoLog(shader, len, &chWrittn, log);
        cout << "Shader Info Log: " << log << endl;
        free(log);
    }
}
```

2.5 Detecting OpenGL and GLSL Errors

```
void printProgramLog(int prog) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);
        glGetProgramInfoLog(prog, len, &chWrittn, log);
        cout << "Program Info Log: " << log << endl;
        free(log);
    }
}
```

2.5 Detecting OpenGL and GLSL Errors

```
bool checkOpenGLError() {
    bool foundError = false;
    int glErr = glGetError();
    while (glErr != GL_NO_ERROR) {
        cout << "glError: " << glErr << endl;
        foundError = true;
        glErr = glGetError();
    }
    return foundError;
}
```

2.5 Detecting OpenGL and GLSL Errors

```
GLuint createShaderProgram() {
    GLint vertCompiled;
    GLint fragCompiled;
    GLint linked;
    ...
    // catch errors while compiling shaders

    glCompileShader(vShader);
    checkOpenGLError();
    glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
    if (vertCompiled != 1) {
        cout << "vertex compilation failed" << endl;
        printShaderLog(vShader);
    }

    glCompileShader(fShader);
    checkOpenGLError();
    glGetShaderiv(fShader, GL_COMPILE_STATUS, &fragCompiled);
    if (fragCompiled != 1) {
        cout << "fragment compilation failed" << endl;
        printShaderLog(fShader);
    }
```

2.5 Detecting OpenGL and GLSL Errors

```
// catch errors while linking shaders

glAttachShader(vfprogram, vShader);
glAttachShader(vfprogram, fShader);

glLinkProgram(vfprogram);
checkOpenGLError();
glGetProgramiv(vfprogram, GL_LINK_STATUS, &linked);
if (linked != 1) {
    cout << "linking failed" << endl;
    printProgramLog(vfprogram);
}

return vfprogram;
}
```

2.6 Reading GLSL Code from Files

- Storing *GLSL shader code* inline in strings is not practical for complicated cases. *Shader code* should be stored in **text files** and read in by our C++/openGL program
- The following sample program shows how to read the *vertex* and *fragment shader code* stored in text files "vertShader,glsl" and "fragShader.glsl", respectively, to build *vShader* and *fShader* for the pipeline.

2.6 Reading GLSL Code from Files

//

// Program 2.4 Reading GLSL source from files

// …. #includes, main(), display(), init() same as before, plus the following…

//

```
#include <string>
#include <iostream>
#include <fstream>

string readShaderSource(const char *filePath) {
    string content;
    ifstream fileStream(filePath, ios::in);
    string line = "";
    while (!fileStream.eof()) {
        getline(fileStream, line);
        content.append(line + "\n");
    }
    fileStream.close();
    return content;
}
```

e.g. "C:\\MyDirectory\\MyFile.bat"

This function reads content of a text file (specified by "filePath") into a string object one line a time and return that string object

2.6 Reading GLSL Code from Files.

```
GLuint createShaderProgram() {
    (... as before plus ...)
    string vertShaderStr = readShaderSource("vertShader.glsl");
    string fragShaderStr = readShaderSource("fragShader.glsl");

    const char *vertShaderSrc = vertShaderStr.c_str();
    const char *fragShaderSrc = fragShaderStr.c_str();

    glShaderSource(vShader, 1, &vertShaderSrc, NULL);
    glShaderSource(fShader, 1, &fragShaderSrc, NULL);

    (... etc, building rendering program as before)
}
```

2.7 Building Objects from Vertices

- How to draw objects constructed of many vertices?
  - How should the *vertex shader* in Program 2.2 be modified?
  - How should the *glDrawArrays()* call be modified?
- A sample program that draws a triangle is shown in Program 2-5. Note that in

  glDrawArrays(GL_TRIANGLES, 0, 3)

  the primitive type is specified as *GL_TRIANGLES* and the number of vertices is set to *3*. This means the vertex shader will run 3 times, once for each vertex.

2.7 Building Objects from Vertices

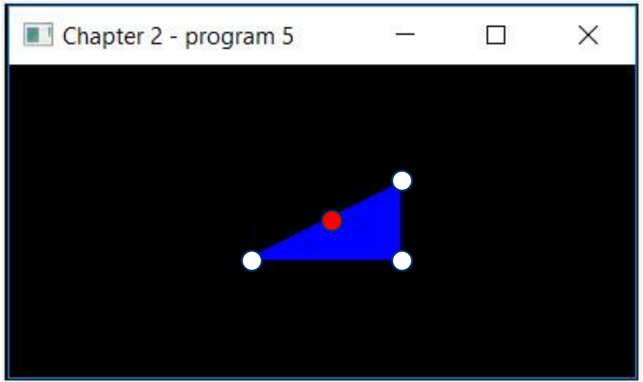
```
//////////////////////////////////////////////////////////////////////////////////////////////////
// Program 2.5   Draw a Triangle
// same as Program 2.4 except the vertex shader and glDrawArrays()
//////////////////////////////////////////////////////////////////////////////////////////////////
```

Vertex Shader

```
#version 430
void main(void)
{ if (gl_VertexID==0) gl_Position = vec4(0.25,-0.25, 0.0, 1.0);
  else if (gl_VertexID==1) gl_Position = vec4(-0.25,-0.25, 0.0, 1.0);
  else gl_Position = vec4( 0.25, 0.25, 0.0, 1.0);
}
```

C++/OpenGL application – in display()

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```



2.8 Animating a Scene

- How to arrange/structure a C++/openGL application so that *animation* is supported?
- **THE ANSWER**: our main() is already structured to do so. **WHY?**
- *Main()* calls the function *display()* repeatedly. Each time *display()* is called, a new *rendering* of the scene is shown on the screen with a 60-120 *frame rate* per second.
- All we need to do is to design the *display()* properly to alter what it draws over time.

2.8 Animating a Scene

//

// Program 2.6 A Simple Animation Example

// same *#includes* and *declarations* as before, plus the following…

//

```
float x = 0.0f;     // location of triangle on x axis
float inc = 0.01f; // offset for moving the triangle

void display(GLFWwindow* window, double currentTime) {
  // glClear(GL_DEPTH_BUFFER_BIT);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);  // clear the background to black

    glUseProgram(renderingProgram);

    x += inc;   // move the triangle along x axis
    if (x > 1.0f) inc = -0.01f;  // switch to moving triangle to left
```

2.8 Animating a Scene

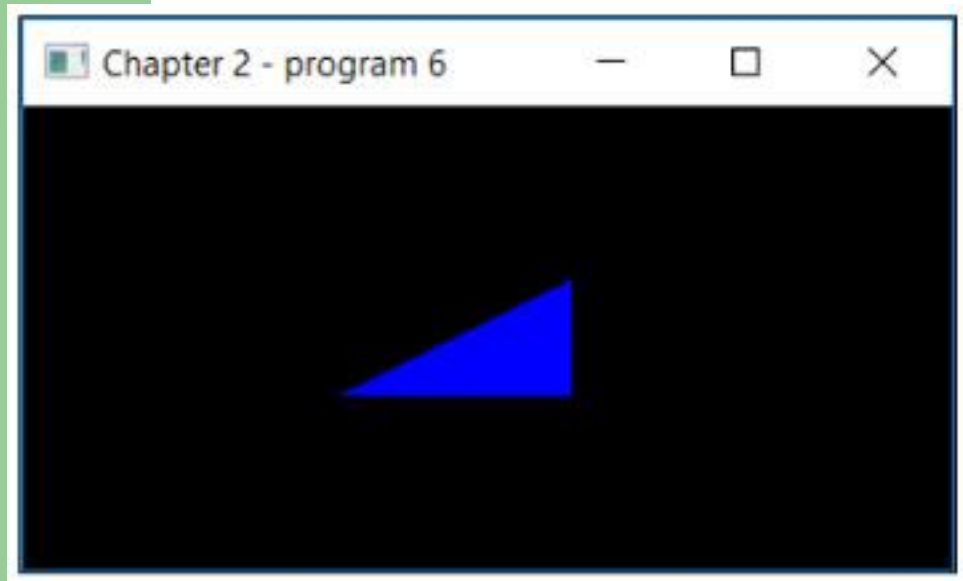
```
    if (x < -1.0f) inc = 0.01f;  // switch to moving triangle to right
    offsetLoc = glGetUniformLocation(renderingProgram, "offset");
    // get ptr to "offset"
    glProgramUniform1f(renderingProgram, offsetLoc, x);
    // send value in "x" to "offset"
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

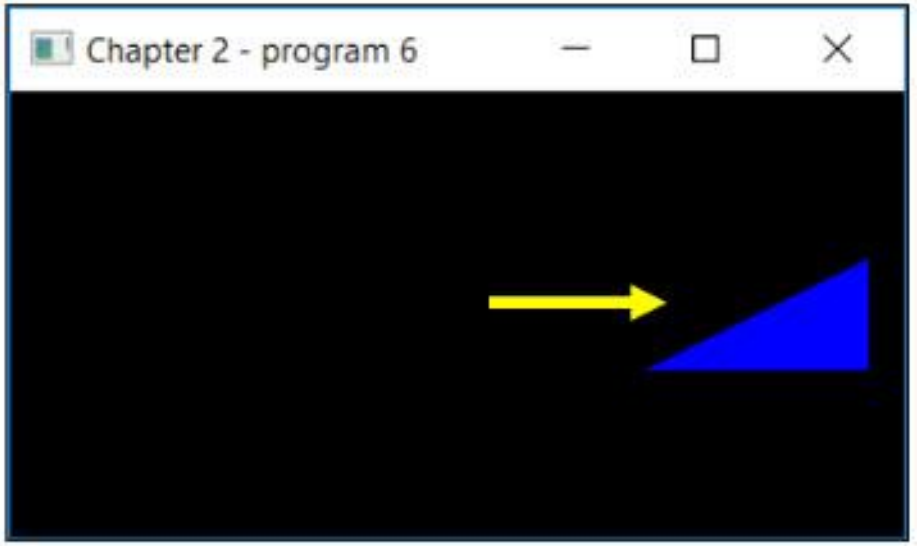
// remaining functions same as before

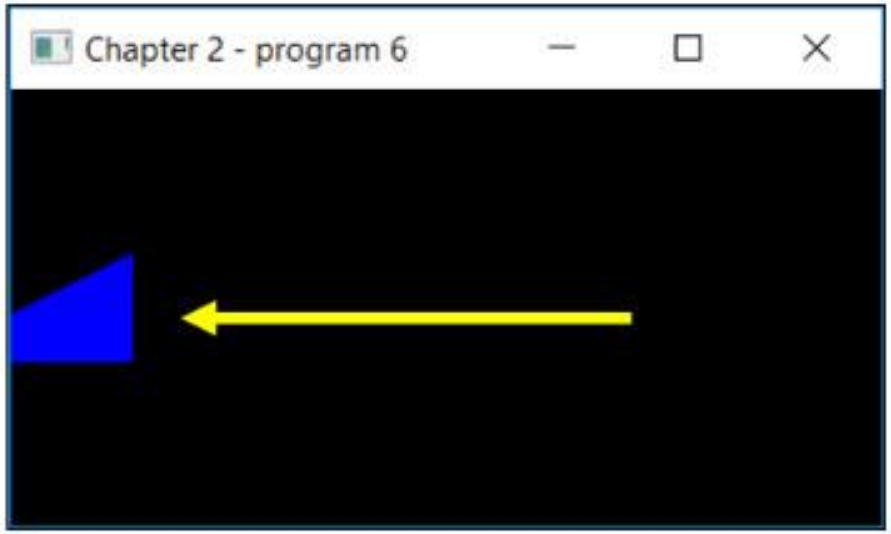
Vertex Shader:

```
#version 430
uniform float offset;
void main(void)
{ if (gl_VertexID == 0) gl_Position = vec4( 0.25+offset,-0.25, 0.0, 1.0);
  else if (gl_VertexID == 1) gl_Position = vec4(-0.25+offset,-0.25, 0.0, 1.0);
  else gl_Position = vec4( 0.25+offset, 0.25, 0.0, 1.0);
}
```

2.8 Animating a Scene







End