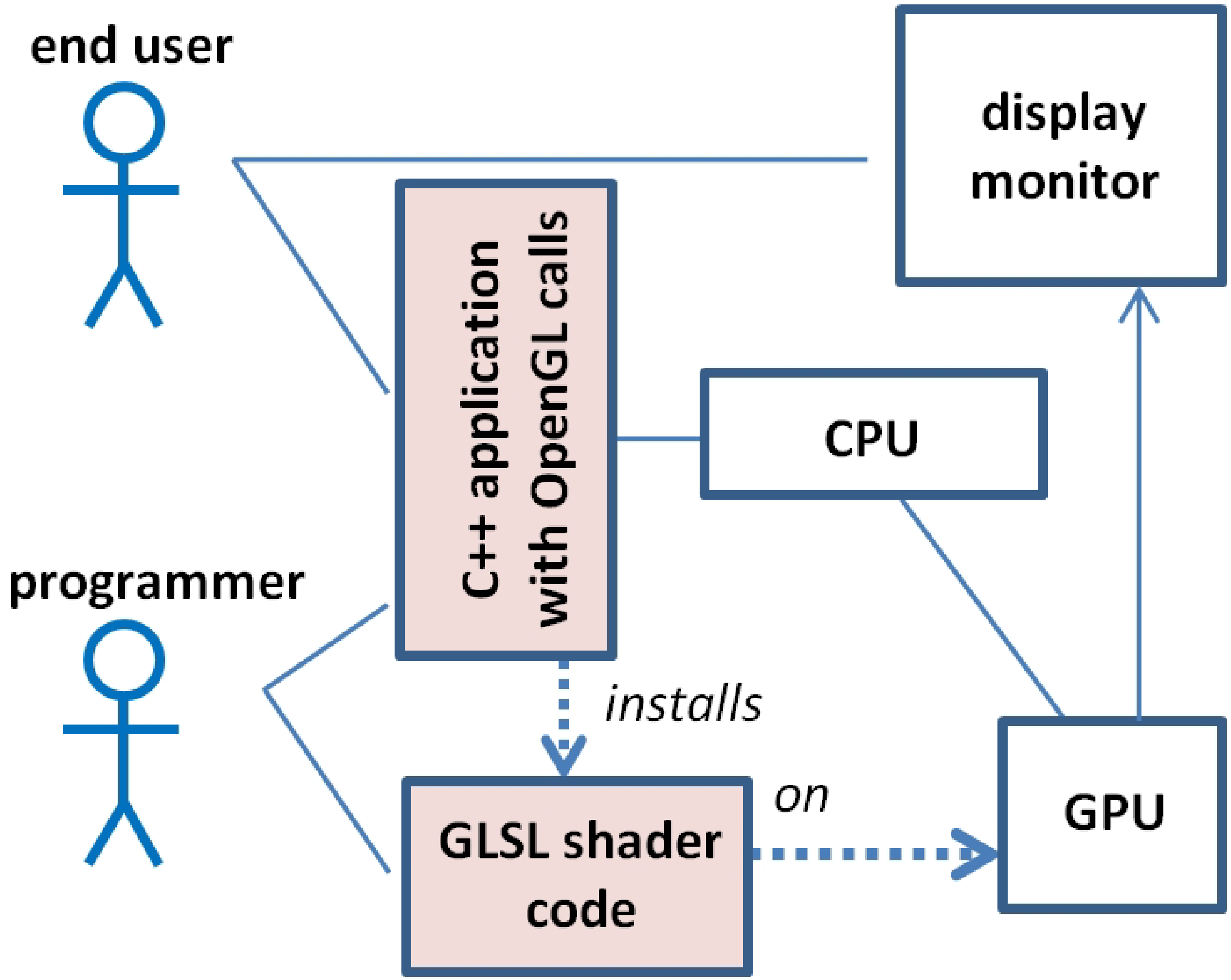
OpenGL Graphics Pipeline

Prof. George Wolberg
Dept. of Computer Science
City College of New York

OpenGL

- Multiplatform 2D and 3D graphics API
- Incorporates hardware
  - Provides a multi-stage graphics pipeline that is partially programmable using a language called GLSL (OpenGL Shading Language)
- Incorporates software
  - Written in C; compatible with C/C++
  - Programmer writes code that runs on CPU and includes OpenGL calls: C++/OpenGL application
  - Programmer's GLSL code is installed on GPU

Components of a C++/OpenGL Application



Software components shown in pink

Overview

- Some of the code we write will be in C++ with OpenGL calls
- Some of the code will be written in GLSL
- Our C++/OpenGL application will work with GLSL modules, and the hardware, to create our 3D graphics output
- Once the application is complete, the end user will interact with the C++ application

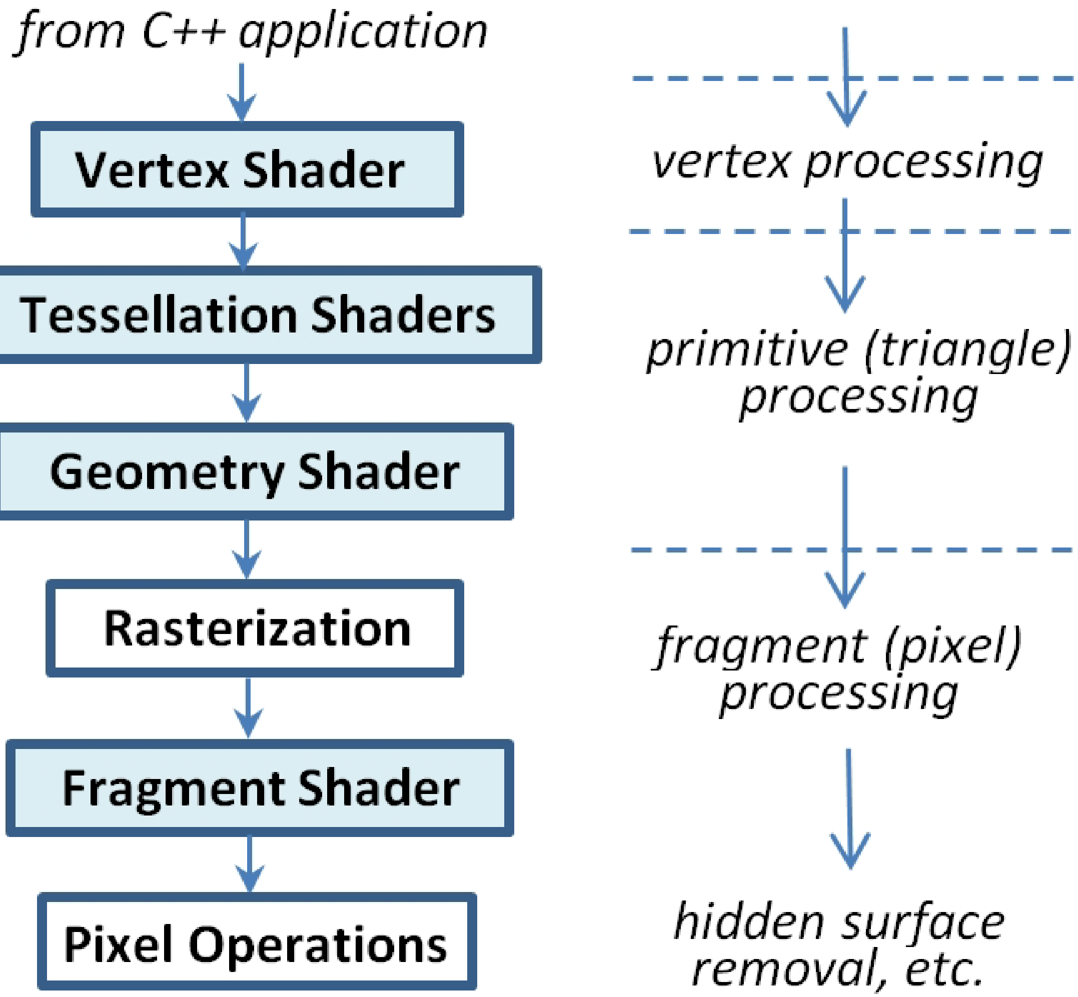
GLSL

- GLSL is an example of a shader language
- Shader languages run on a GPU in the context of a graphics pipeline
- There are other shader languages
  - HLSL: works with Microsoft's 3D framework DirectX
- GLSL is the specific shader language compatible with OpenGL
- We will write shader code in GLSL, in addition to our C++/OpenGL application code

OpenGL Pipeline

- Modern 3D graphics programming uses a pipeline to convert a 3D scene into a 2D image
- The C++/OpenGL application sends graphics data into the vertex shader
- Processing proceeds through the pipeline and pixels emerge for display on the monitor

OpenGL Pipeline Overview



Stages shaded in blue are programmable

Programmable Stages

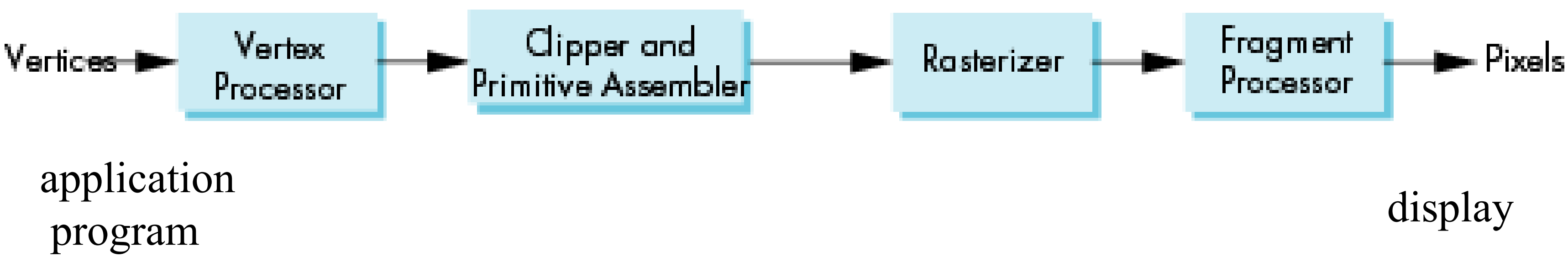
- The vertex, tessellation, geometry, and fragment stages are programmable in GLSL
- It is one of the responsibilities of the C++/OpenGL app to load GLSL programs into these shader stages as follows:
  - It uses C++ to obtain the GLSL shader code, either from text files or hardcoded as strings
  - It then creates OpenGL shader objects and loads the GLSL code into them
  - Finally, it uses OpenGL commands to compile and link objects and install them on the GPU

Programmable Stages

- GLSL code for the vertex and fragment stages is required
- The tessellation and geometry stages are optional

Pipeline with Required Stages Only

- Process 3D objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture



- All steps can be implemented in hardware on the graphics card

Following the Graphics Pipeline: Vertex Processing

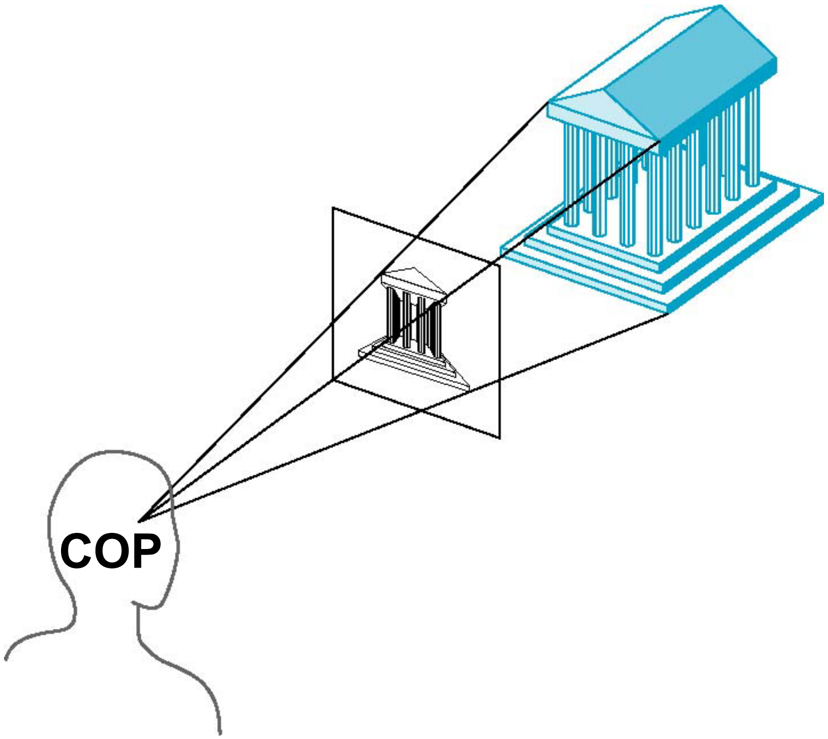
- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

Projection

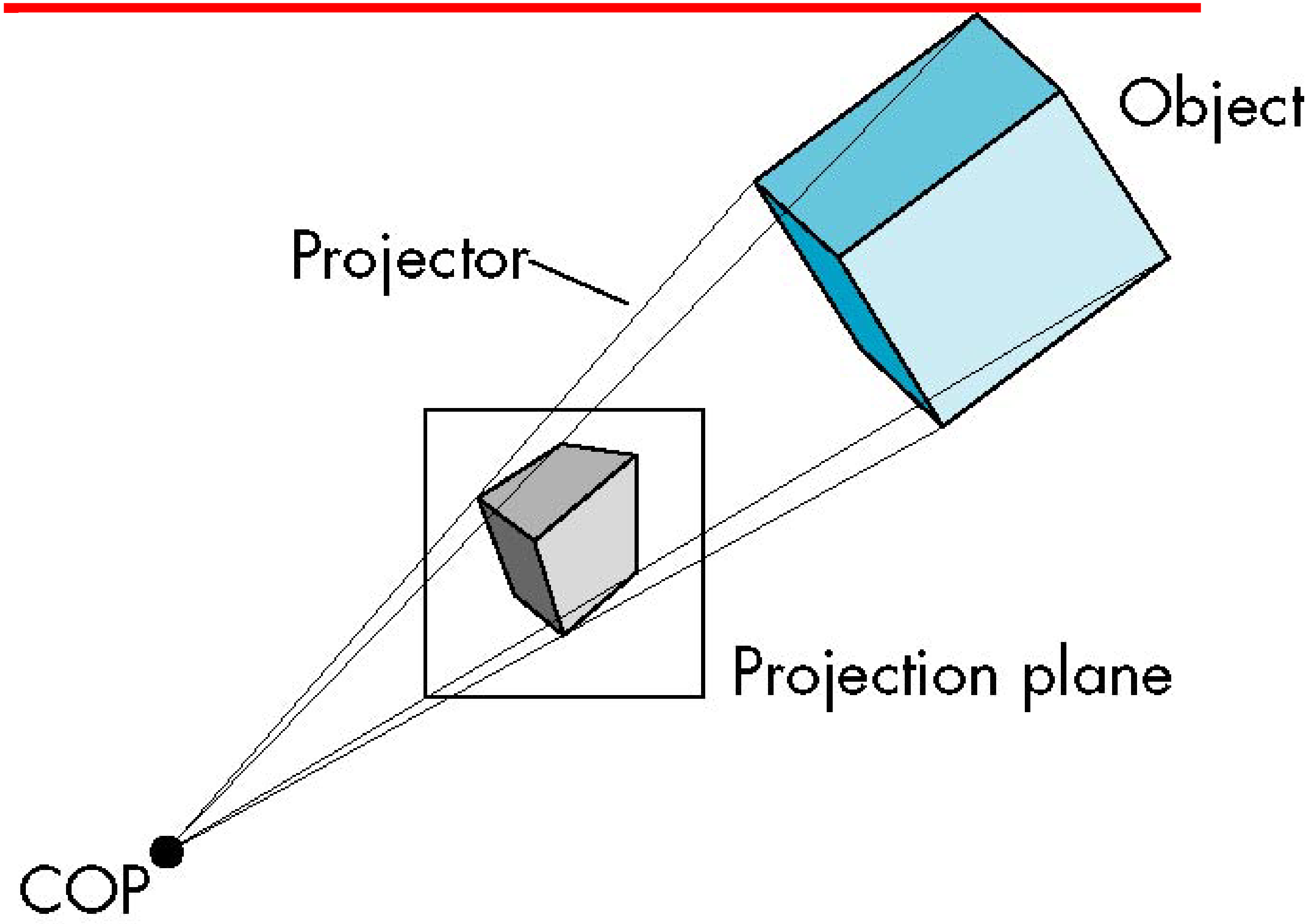
- *Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image
  - Perspective projections: all projectors meet at the center of projection
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection

Perspective Projection

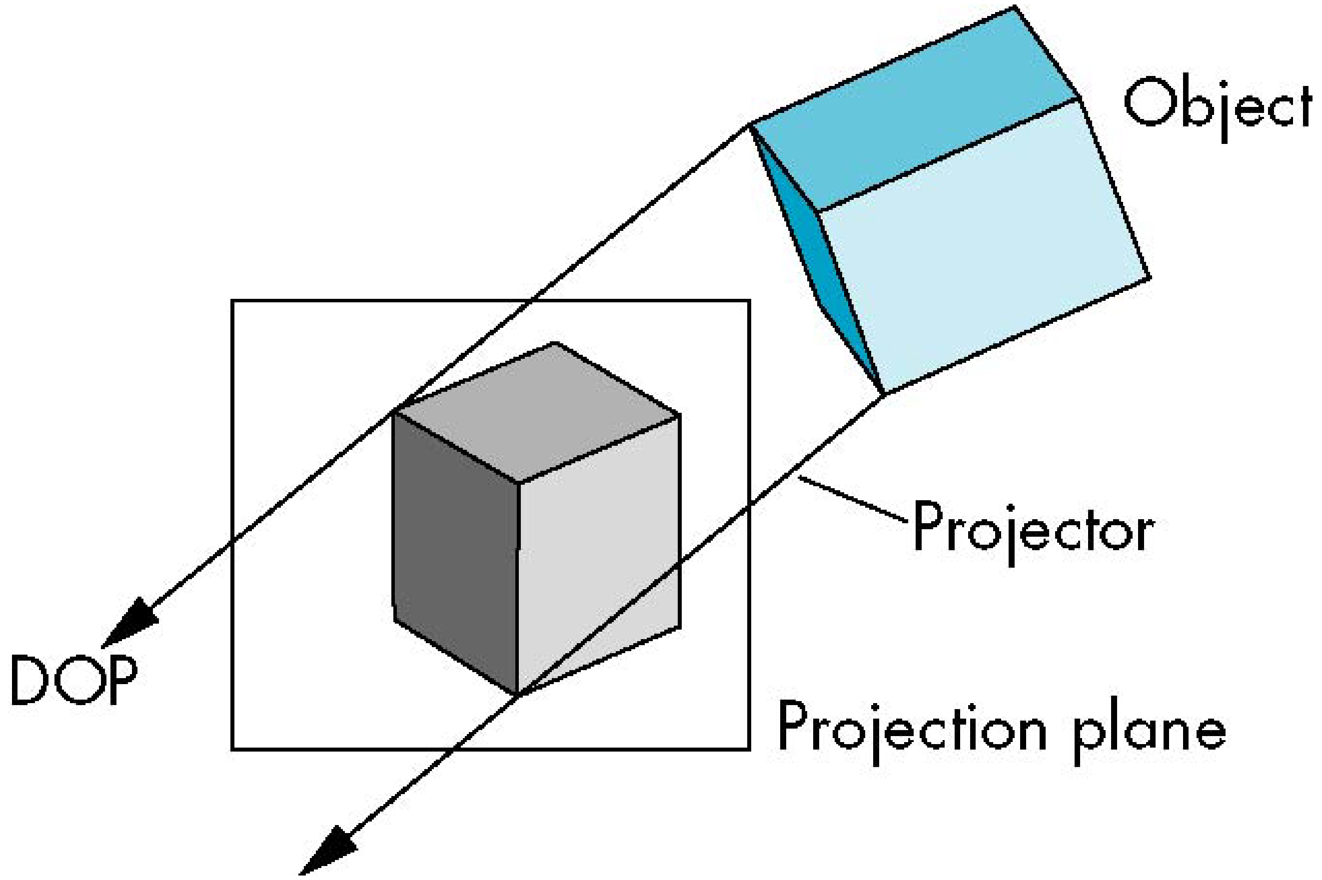
Projectors converge at center of projection (COP)



Perspective Projection



Parallel Projection



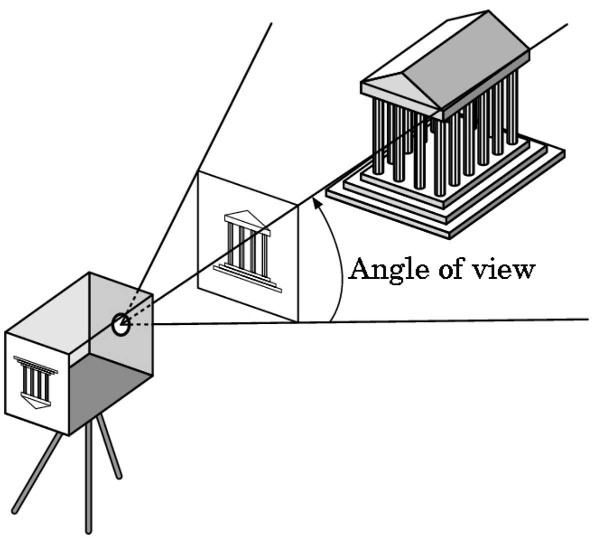
Primitive Assembly

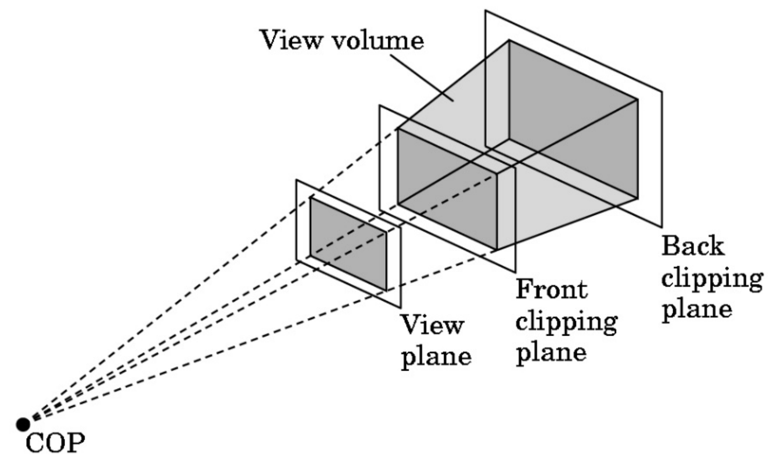
- The fundamental unit of rendering in OpenGL is known as the *primitive*.
- The three basic primitive types are points, lines, and triangles.
- Vertices must be collected into primitives before clipping and rasterization can take place

Clipping

Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space

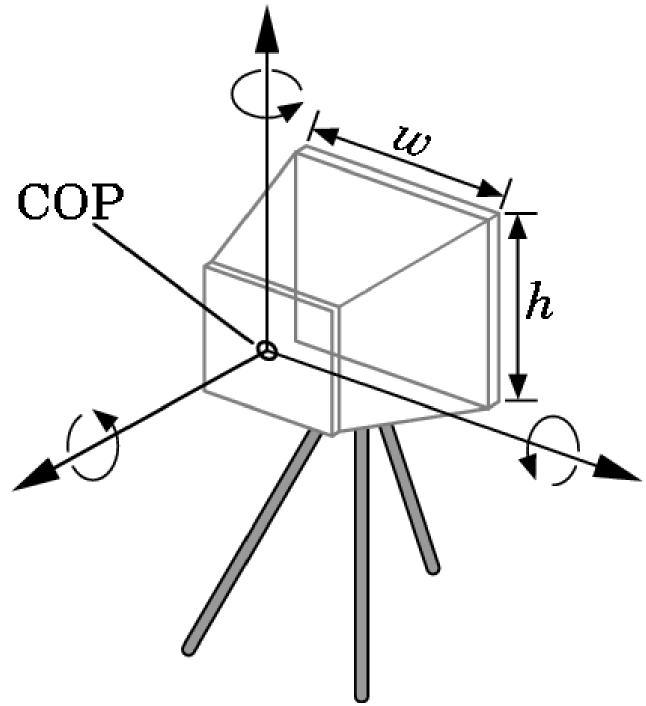
- Objects that are not within this volume are said to be *clipped* out of the scene





Specification of Virtual Camera

- Six degrees of freedom
  - Position of center of lens
  - Orientation
- Lens
- Film size
- Orientation of film plane

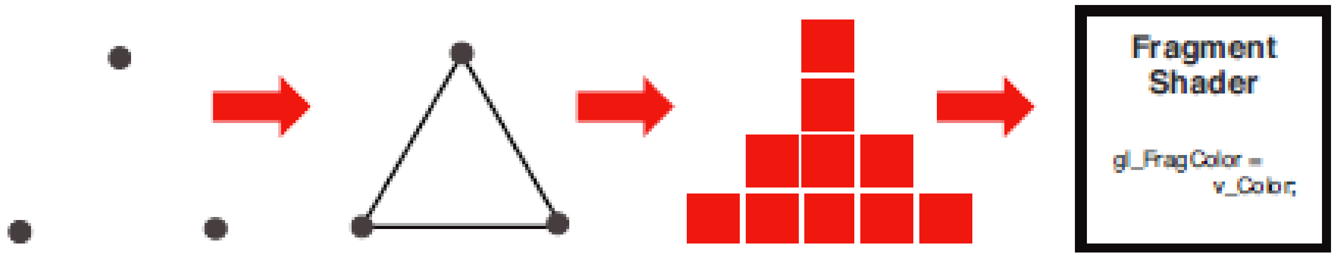


Rasterization

- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are "potential pixels"
  - Have a location in frame buffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer

Putting It All Together

- Vertices stream into vertex processor and are transformed into new vertices
- These vertices are collected to form primitives
- Primitives are rasterized to form fragments
- Fragments are colored by fragment processor



Fragment Processing

- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal

C++/OpenGL Application

- Bulk of graphics application is written in C++
- The application may interact with the end user using standard C++ libraries
- For 3D rendering tasks, it uses OpenGL calls
- Several additional libraries may be used:
  - GLEW (OpenGL extension wrangler)
  - GLM (OpenGL Math library)
  - SOIL2 (Simple OpenGL Image Loader)
  - GLFW (OpenGL Framework)

C++/OpenGL Application

- GLFW library includes a class called GLFWwindow on which we can draw 3D scenes
- In the next example, we use main() to:
  - Call glfwInit() to initialize the GLFW library
  - Call glfwCreateWindow() to instantiate a GLFWwindow
  - Call glewInit() to initialize the GLEW library
  - Call init() once for application-specific tasks
  - Call display() repeatedly to draw to the GLFWwindow
- glClearColor() specifies the background color
- glClear() clears window with background color

Simple C++/OpenGL Application

```
void init(GLFWwindow* window) { }

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

int main(void)  {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 - program1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);

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

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

(#includes and namespace not shown)

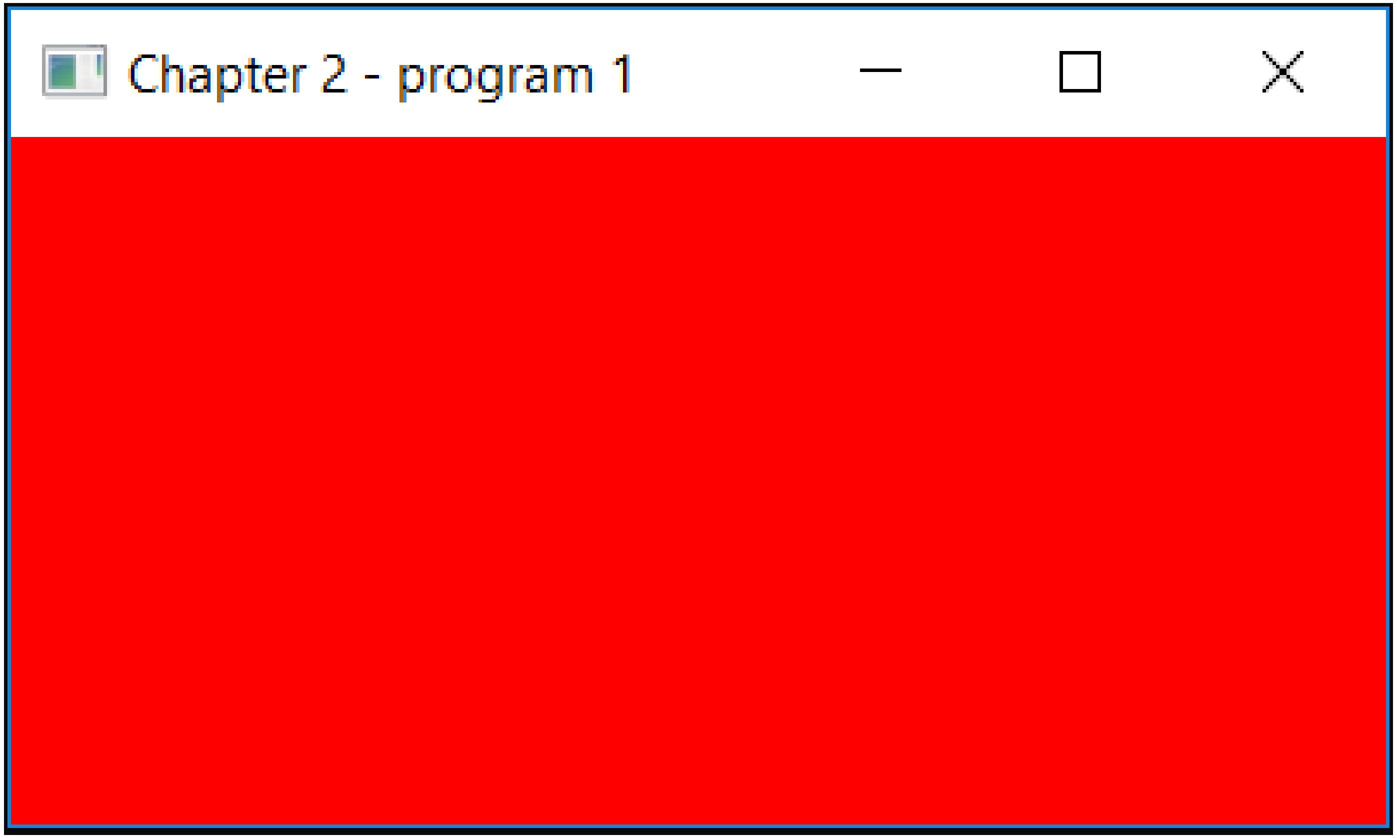
C++/OpenGL Application

- The window hints specify that the machine must be compatible with OpenGL version 4.3
- The parameters of glfwCreateWindow() specify the width and height of the window (in pixels) and the title placed at the top of the window
- The additional two parameters (NULL) allow for full screen mode and resource sharing
- Vertical synchronization (VSync) is enabled by using glfwSwapInterval() and glfwSwapBuffers()

C++/OpenGL Application

- By default, GLFW windows are double buffered
- Creating the GLFW window doesn’t automatically make the OpenGL context current
  - We must therefore call glfwMakeContextCurrent()
- glfwSwapBuffers() paints the screen
- glfwPollEvents() handles other window-related events such as a key being pressed
- The loop terminates when GLFW detects an event that should close the window (such as clicking the X in the upper right corner)

Running the Application



Primitives

- OpenGL can only draw a few simple things:
  - points, lines, or triangles
- These simple things are called *primitives*
- Most 3D models are made up of lots of primitives, usually triangles
- Primitives are made up of vertices
  - For example, a triangle consists of three vertices
- Vertices can come from a variety of sources
  - Read from files and loaded into buffers
  - Hardcoded in the C++ or GLSL code

C++/OpenGL program

- The C++/OpenGL program must first compile and link appropriate GLSL vertex and fragment shader programs, and load them into the pipeline
- The C++/OpenGL program also tells OpenGL to construct triangles:
  - glDrawArrays(GLenum mode, GLint first, GLsizei count)
  - The mode is the type of primitive (GL_TRIANGLES)
  - first indicates which vertex to start with (vertex 0 is first)
  - The count specifies total number of vertices to be drawn
- When glDrawArrays() is called, the GLSL code in the pipeline starts executing

Adding Vertex and Fragment Shaders

- All vertices pass through the vertex shader
- The shader is executed once per vertex
- Vertex shader may execute millions of times for large models
- To display vertex, we also need to provide a fragment shader
- For simplicity, we will declare the two shader programs as arrays of strings

Adding Vertex and Fragment Shaders

```
#define numVAOs 1
GLuint renderingProgram;
GLuint vao[numVAOs];

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

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

(continued)

```
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); }";
    GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
    GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(vShader, 1, &vshaderSource, NULL);
    glShaderSource(fShader, 1, &fshaderSource, NULL);
    glCompileShader(vShader);
    glCompileShader(fShader);
    GLuint vfProgram = glCreateProgram();
    glAttachShader(vfProgram, vShader);
    glAttachShader(vfProgram, fShader);
    glLinkProgram(vfProgram);
    return vfProgram;
}
```

Notes

- GLuint refers to the *unsigned int* data type
- init() now calls createShaderProgram() to read two hard-coded strings for the vertex and fragment shaders
  - vshaderSource is the character string that stores vertex shader code
  - fshaderSource is the character string that stores fragment shader code
- We call glCreateShader() twice to create the two shader objects and return an integer ID for each that is an index for referencing it later
  - vShader and fShader are the two integer IDs
- glShaderSource() loads the GLSL code from the strings into the empty shader objects indexed by the integer IDs.
  - The number of lines of code in each shader is listed as one.
- The shaders are then compiled using glCompileShader()

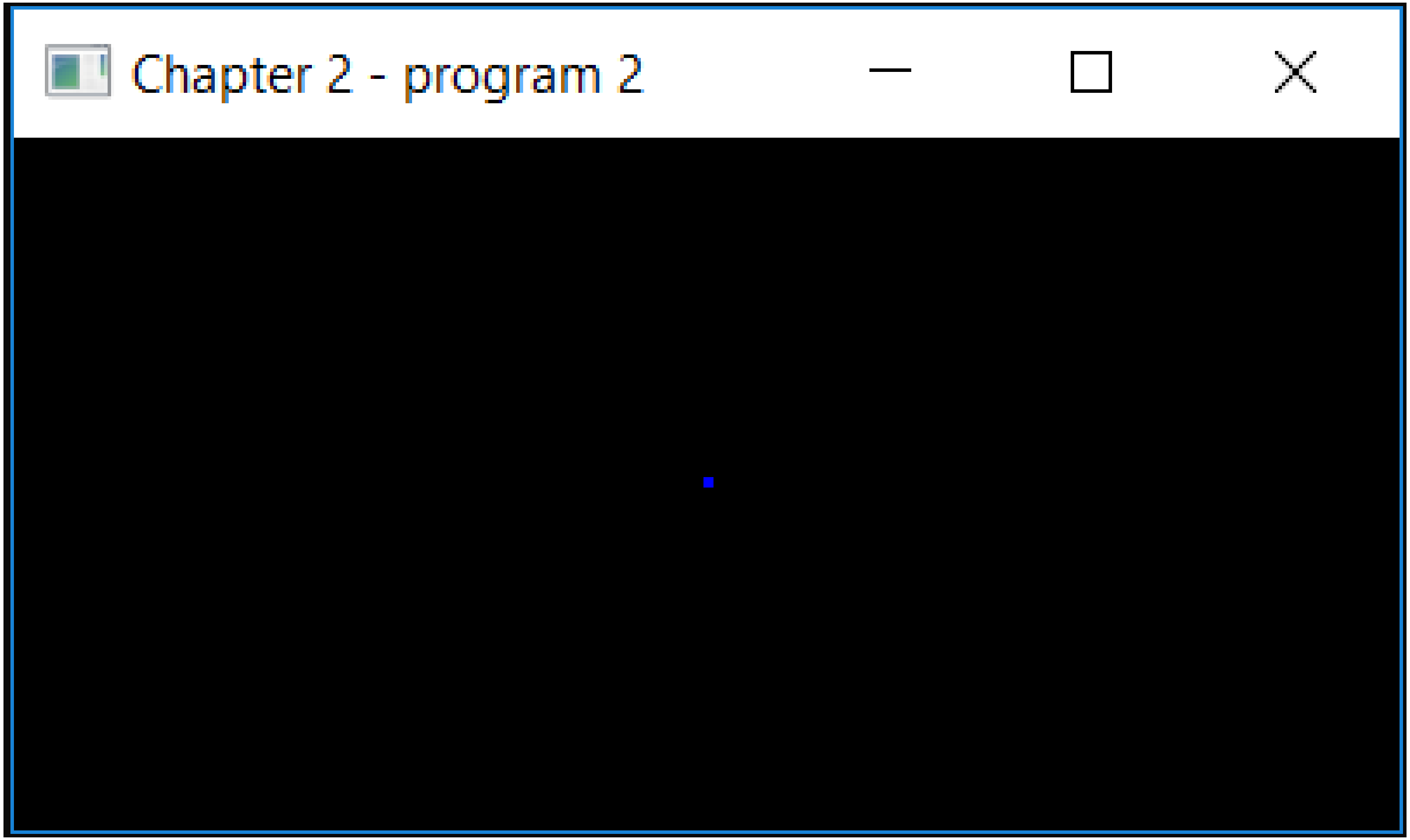
Notes

- An empty OpenGL program object is created using glCreateProgram() to hold a series of compiled shaders
- glAttachShader() is called twice to attach the compiled vertex and fragment shaders
- glLinkProgram() is called to request the GLSL compiler to ensure that the attached shaders are compatible

display()

- After init(), the display() function is called repeatedly
- It calls glUseProgram(), which loads the program containing the two compiled shaders into the OpenGL pipeline stages (onto the GPU)
- glUseProgram() doesn't run the shaders; it just loads them onto the hardware
- glDrawArrays() is called to initiate pipeline processing
- GL_POINTS (points) is the primitive type to be displayed
- Only a single point is displayed in this simple example

Running the Application



Vertex Shader

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

- The vertex shader is run once for each vertex
- The first line indicates the OpenGL version: 4.3
- The built-in variable gl_Position is used to set vertex position
- The GLSL datatype vec4 holds a 4-tuple (e.g., (0,0,0,1))
- The vertices move through the pipeline to the rasterizer where they are transformed into pixel locations (fragments)
- These pixels (fragments) reach the fragment shader

Fragment Shader

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

- The fragment shader is run once for each fragment
- Its purpose is to set the RGBA color of pixel to be displayed
- In this case, the color is blue (0,0,1) and the opacity is 1
- The out tag indicates that the variable color is an output
- It wasn't necessary to specify an out tag for gl_Position in the vertex shader because it is a predefined output variable

Vertex Array Buffer

- init() contained the following two lines:

```
glGenVertexArrays(numVAOs, vao);  // numVAOs = 1
glBindVertexArray(vao[0]);
```

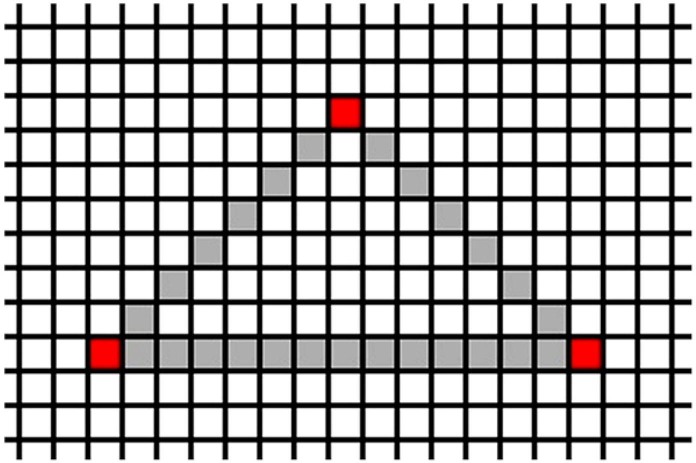
- Data is organized into buffers when sent down the pipeline
- Those buffers are organized into Vertex Array Objects (VAOs)
- We didn't need any buffers since we only displayed one point
- However, OpenGL still requires at least one VAO be created whenever shaders are being used, even if the application isn't using any buffers

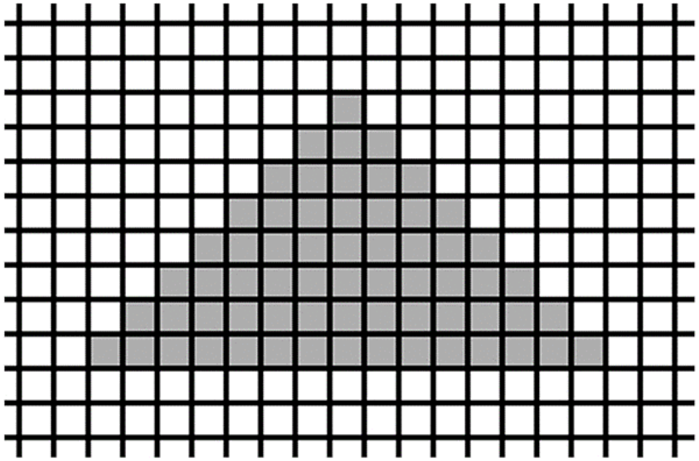
Rasterization

- How does the vertex that comes out of the vertex shader become a pixel in the fragment shader?
- The rasterization stage between the vertex and fragment shaders is responsible for converting primitives into pixels
- The default size of an OpenGL point is one pixel, so that is why our single point was rendered as a single pixel
- If we add glPointSize(30.), then point is rendered as 30 pixels

Rasterization

- When a 3D object is rasterized, OpenGL converts the primitives in the object (usually triangles) into fragments
- A fragment holds the information associated with a pixel
- Rasterization determines the pixel locations to be drawn in order to produce the triangle specified by its three vertices
- The process starts by interpolating, pairwise, between the three vertices of the triangle

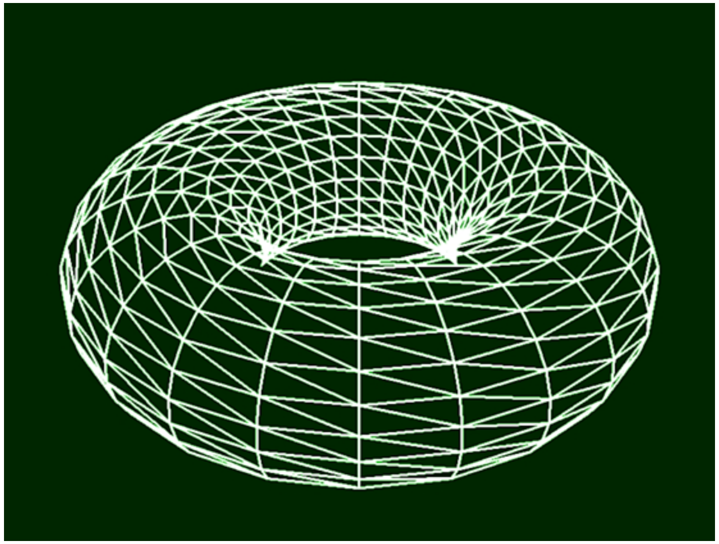




Wire Frame

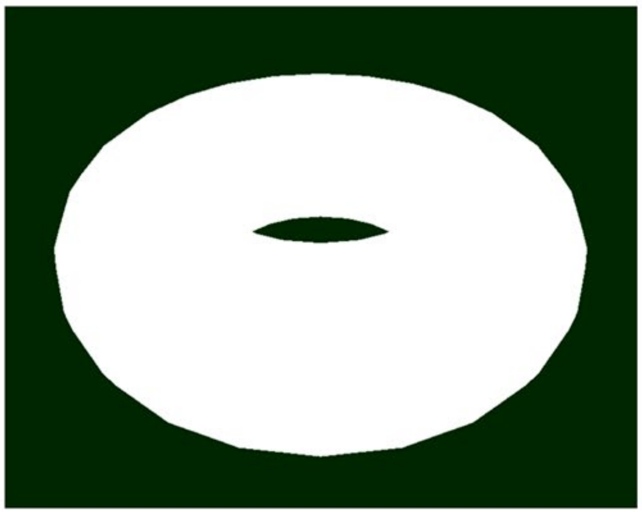
- Instead of filling with rasterization, we can draw a wireframe:

  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)

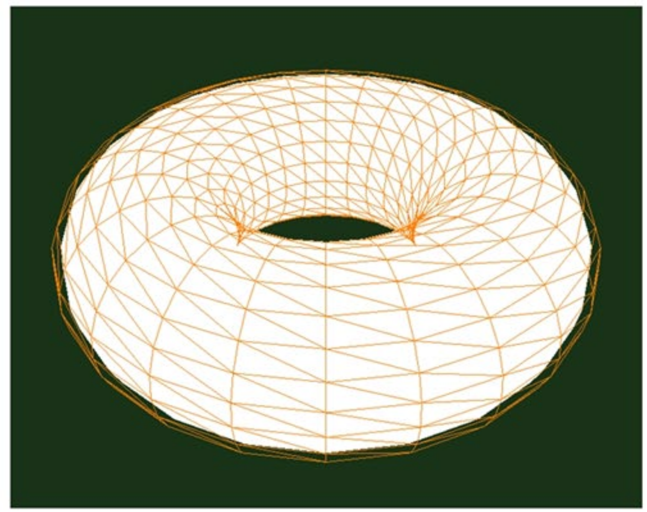


Example

- Torus with fully rasterized primitives (left) and with wireframe grid superimposed (right)



Torus with fully rasterized primitives



Wireframe grid superimposed

Pixel Operations

- We expect to see objects in front to block our view of objects behind them
- We expect to see the front of an object, but not its back
- To achieve this, we need *hidden surface removal* (HSR)
- This phase is not programmable, but we need to understand how it works and how to configure it
- It will be useful later when including shadows in our scene

Hidden Surface Removal

- Accomplished through the cleverly coordinated use of two buffers: the color buffer and the depth (Z) buffer
- There is an entry in each buffer for every pixel on the screen
- As various objects are drawn in a scene, pixel colors are generated by the fragment shader and placed in the color buffer, which is ultimately written to the screen
- When multiple objects occupy the same pixels in the color buffer, a determination must be made as to which pixel colors are retained, based on which object is nearest the viewer

Hidden Surface Removal

- Hidden surface removal is done as follows:
- Fill the depth buffer with values representing maximum depth
- As a pixel color is output by the fragment shader, its distance from the viewer is calculated
- If the computed distance is less than the distance stored in the depth buffer for that pixel, then
  - (a) the pixel color replaces the color in the color buffer, and
  - (b) the computed distance replaces the value in the depth buffer
  - (c) otherwise the pixel is discarded
- This procedure is called the Z-buffer algorithm

Hidden Surface Removal (HSR) (Z-Buffer Algorithm)

```
Color   [ ] [ ]  colorBuf = new  Color   [pixelRows][pixelCols];
double [ ] [ ] depthBuf = new double [pixelRows][pixelCols];

for (each row and column)  {     // initialize color and depth buffers
        colorBuf  [row][col] = backgroundColor;
        depthBuf [row][col] = far away;
}

for (each shape) {     // update buffers when new pixel is closer
        for (each pixel in the shape) {
                if (depth at pixel < depthBuf value) {
                        depthBuf [pixel.row][pixel.col] = depth at pixel;
                        colorBuf  [pixel.row][pixel.col] =  color at pixel;
                }
        }
}

return colorBuf;
```

Building Objects from Vertices

- Consider drawing objects of more than just a single point
- We now extend our code to draw objects of many vertices
- Begin with a simple example: define three vertices and use them to draw a triangle
  - Our vertex shader will be modified to output three different vertices to subsequent stages of the pipeline
  - glDrawArrays() will be modified to specify that we are using three vertices
- In the glDrawArrays() function in the C++/OpenGL code, we specify GL_TRIANGLES instead of GL_POINTS, and also specify that there are three vertices sent through the pipeline
- This causes the vertex shader to run three times, and at each iteration, the built-in variable gl_VertexID is automatically incremented (it is initially set to 0)

Building Objects from Vertices

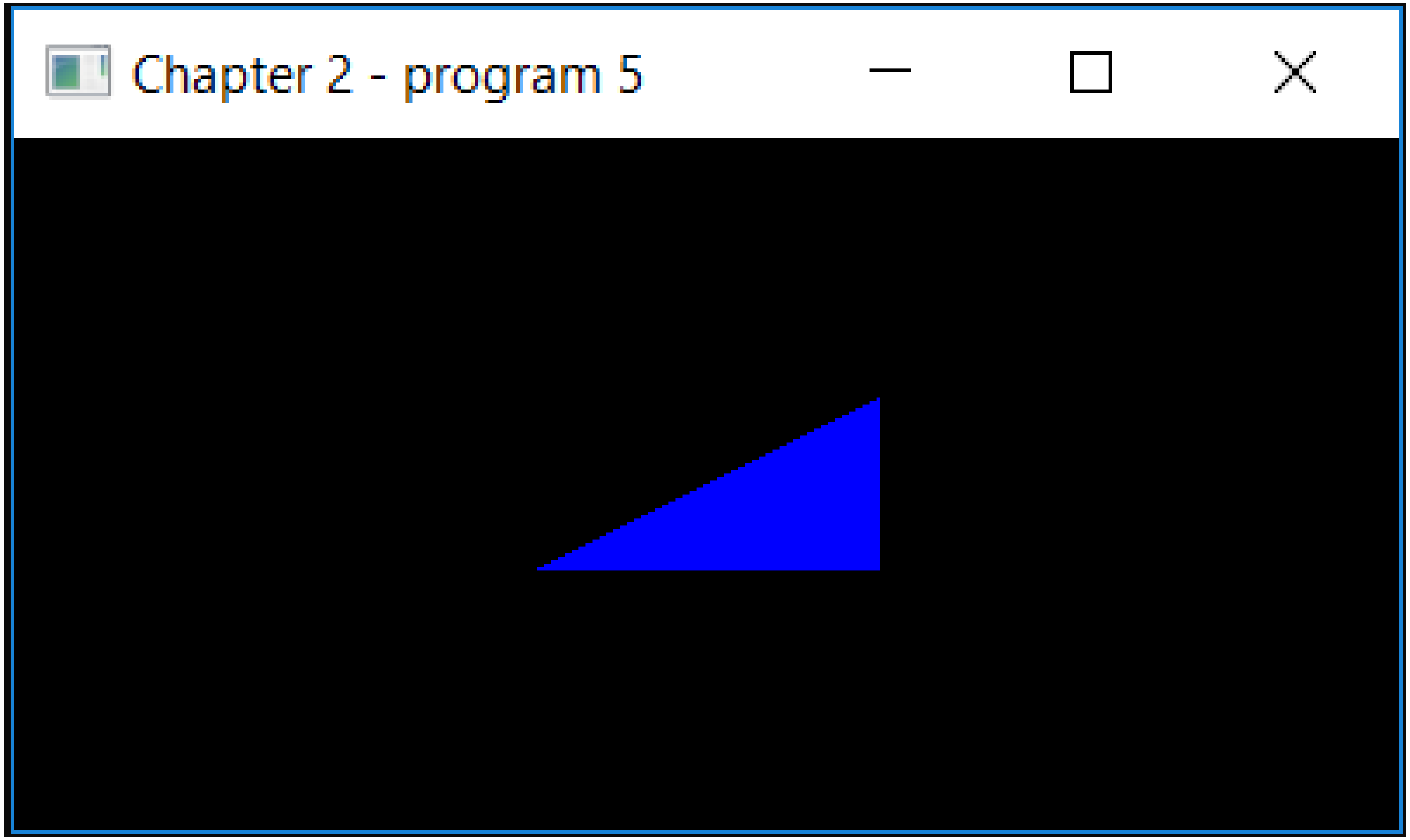
Vertex Shader

```
#version 430
void main(void) {
        switch(gl_VertexID) {
        case 0:     gl_Position = vec4( 0.25, -0.25, 0.0, 1.0); break;
        case 1:     gl_Position = vec4(-0.25, -0.25, 0.0, 1.0); break;
        default:    gl_Position = vec4( 0.25,  0.25, 0.0, 1.0); break;
        }
}
```

C++/OpenGL application -- in display()

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

The Application Draws a Triangle



Adding Animation

- We constructed main() to make a single call to init() and call display() repeatedly
- While preceding examples may have appeared to be a single fixed rendered scene, in actuality the loop in main() was causing it to be drawn over and over again
- Our main() is already structured to support animation
- We simply design our display() to alter what it draws over time
- Each rendering of our scene is called a *frame,* and the frequency of the calls to display() is called the *frame rate*

Adding Animation

<u>*in C++/OpenGL application:*</u>

```
. . .
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, each time

        glUseProgram(renderingProgram);

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

        GLuint offsetLoc = glGetUniformLocation(renderingProgram, "offset");       // get pointer to "offset"
        glProgramUniform1f(renderingProgram, offsetLoc, x);                          // send value in "x" to "offset"
        glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

(continued)

Adding Animation

in 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);
}
```