Comp 410/510

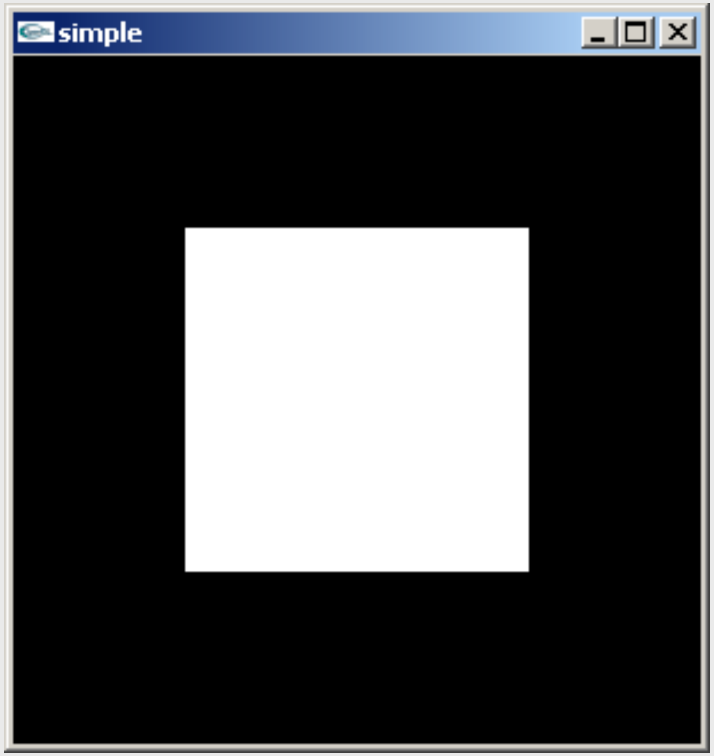
Computer Graphics

Spring 2023

Programming with OpenGL
Part 2: First Program

Objectives

- Look into the first program
  - Introduce a standard program structure
  - Initialization



Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions:
  - **`main()`**:
    - defines the callback functions
    - opens one (or more) windows with the required properties
    - enters event loop
  - **`init()`**:
    - sets the state (viewing, attributes, etc)
    - loads shaders
  - **`display()`**:
    - specifies what to display
  - **callbacks**
    - user input
    - window functions
    - animation (specify what changes with time)

`main()` - using GLFW

```
#include <gl3.h>
#include <glfw3.h>


int main(int argc, char** argv){

    initWindowAPI();
    GLFWwindow* window = glfwCreateWindow(500, 500, "Simple", NULL, NULL);
    glfwMakeContextCurrent(window);

    init();

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

user-defined function; initiates a session with windowing system (`glfwInit()`, etc.)

creates a window and its associated OpenGL context

user-defined; set OpenGL state and shaders

user-defined; render the scene into back buffer

swap back and front bufffers

check the event queue and execute the associated callbacks

initWindowAPI()

```
void initWindowAPI()
{
 if (!glfwInit())
           exit(EXIT_FAILURE);

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_RESIZABLE, GL_TRUE);

}
```

User-defined function to initiate the session and set the properties of the current window and associated openGL context.

Let's look into `init()` and `display()`

```
#include <gl3.h>
#include <glfw3.h>


int main(int argc, char** argv){

   initWindowAPI();
   GLFWwindow* window = glfwCreateWindow(500, 500, "Simple", NULL, NULL);
   glfwMakeContextCurrent(window);

   init();

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

Immediate vs Retained Mode Graphics

- Geometry specified in terms of vertices
  - Locations in space (2 or 3 dimensional)
  - Points, lines, circles, polygons, curves, surfaces, etc
- Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Old style used **`glVertex`**
  - Creates bottleneck between CPU and GPU
  - Removed from OpenGL 3.1 (deprecated in 3.0)
- Retained mode
  - Put all vertex and attribute data in array
  - Send array over and store on GPU for multiple renderings

display()

- Once we get data to GPU, we can initiate the rendering with a simple callback

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}
```

How to send data to GPU: Vertex Arrays

- Vertices can have many attributes such as
  - Position
  - Color
  - Texture Coordinates
  - Application data

- An array can hold this data
  - e.g., can be defined using types in `vec.h`

```
GLfloat  vertices[NumVertices][3] = {
        { -0.5, -0.5, 0.0 },
        { -0.5,  0.5, 0.0 },
        {  0.5,  0.5, 0.0 },
        {  0.5, -0.5, 0.0 },
        { -0.5, -0.5, 0.0 },
        {  0.5,  0.5, 0.0 }
    };
```

Vertex Array Object (VAO)

- VAO bundles all vertex attribute data (positions, colors, etc.)

- First get unused name(s) for vertex array object(s) then bind:

# of VAOs

get names for VAOs

```
Gluint vao;
glGenVertexArrays(1,&vao);
glBindVertexArray(vao);
```

- `glBindVertexArray` creates the vertex array object but yet with no content

- Subsequent calls of `glBindVertexArray` let us switch between multiple VAOs

Vertex Buffer Object (VBO)

- Need to associate a VBO with each VAO
- VBOs allow us to transfer large amounts of data to GPU
- Hold attribute data for vertex array on the GPU
- Need to create, bind and identify data

# of VBOs

```
Gluint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices), vertices, GL_STATIC_DRAW);
```

get names for VBOs

create VBO with no content

- Data associated with current vertex array is sent to GPU

Initialization

- Vertex arrays and associated buffer objects can be set up on `init()`

- Also set OpenGL state (such as clear color, projection, transformation, etc.)

- Also set up **shaders** as part of initialization
  - Read
  - Compile
  - Link

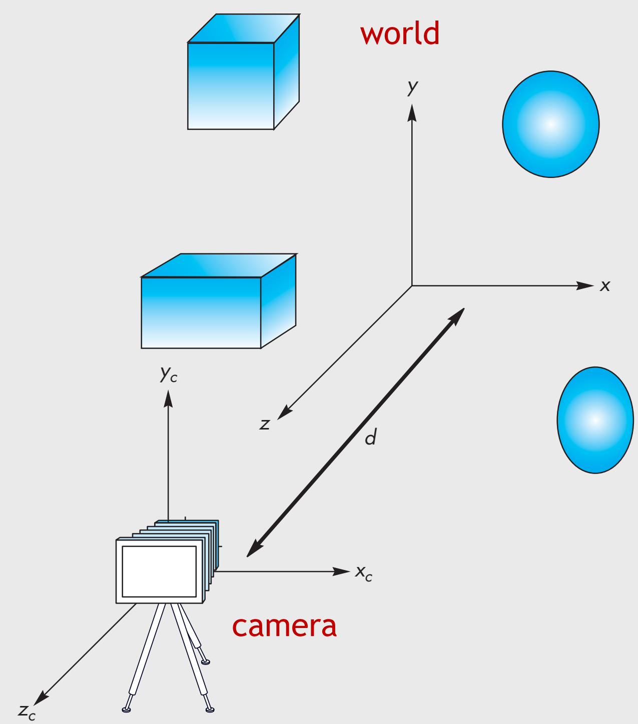
  (See `InitShader.cpp`)

init()

```
void init()
{
    // Set vertex array object(s)
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    GLfloat vertices[NumVertices][3] = {
        { -0.5, -0.5, 0.0 },
        { -0.5,  0.5, 0.0 },
        {  0.5,  0.5, 0.0 },
        {  0.5, -0.5, 0.0 },
        { -0.5, -0.5, 0.0 },
        {  0.5,  0.5, 0.0 }
    };

    // Set vertex buffer object(s) and send data to GPU
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Load shaders and use the resulting shader program
    GLuint program = InitShader( ”vshader_simple.glsl", ”fshader_simple.glsl" );
    glUseProgram( program );

    // Associate attribute variable(s) in the shader with the vertex attribute data
    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
}
```

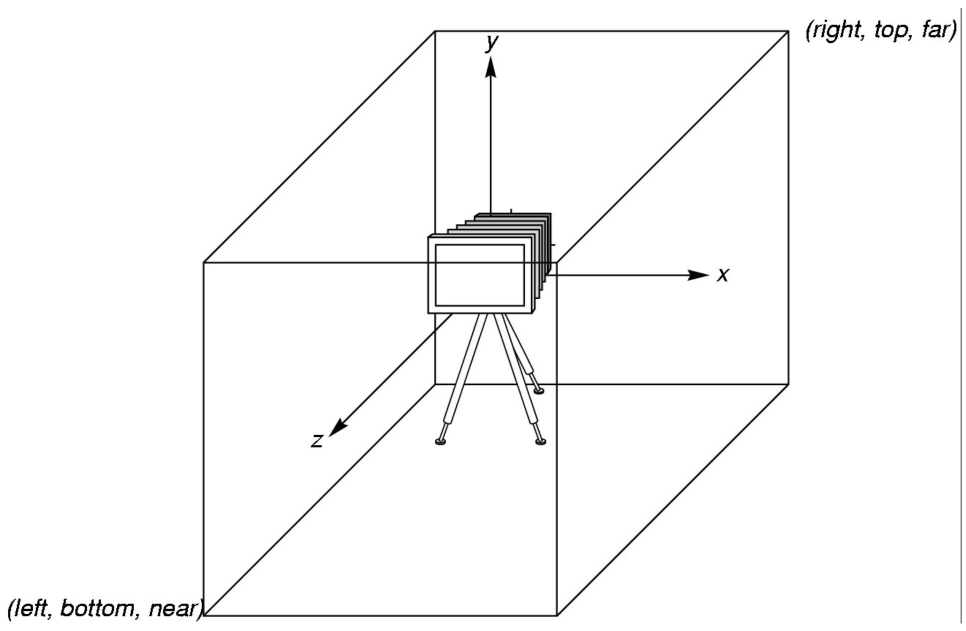
Coordinate Systems

- The units in `vertices` are determined by the application in world (or object) coordinates
- The viewing specifications are also in world coordinates, and it is the size of the viewing volume that determines what will appear in the image
- World coordinates are then converted to camera coordinates and eventually to window (or image) coordinates



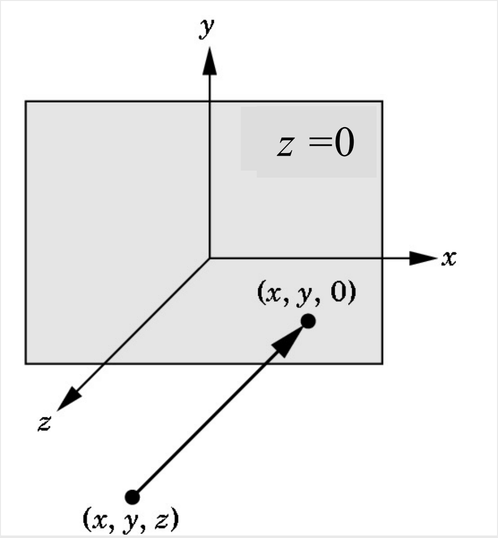
OpenGL Camera Default

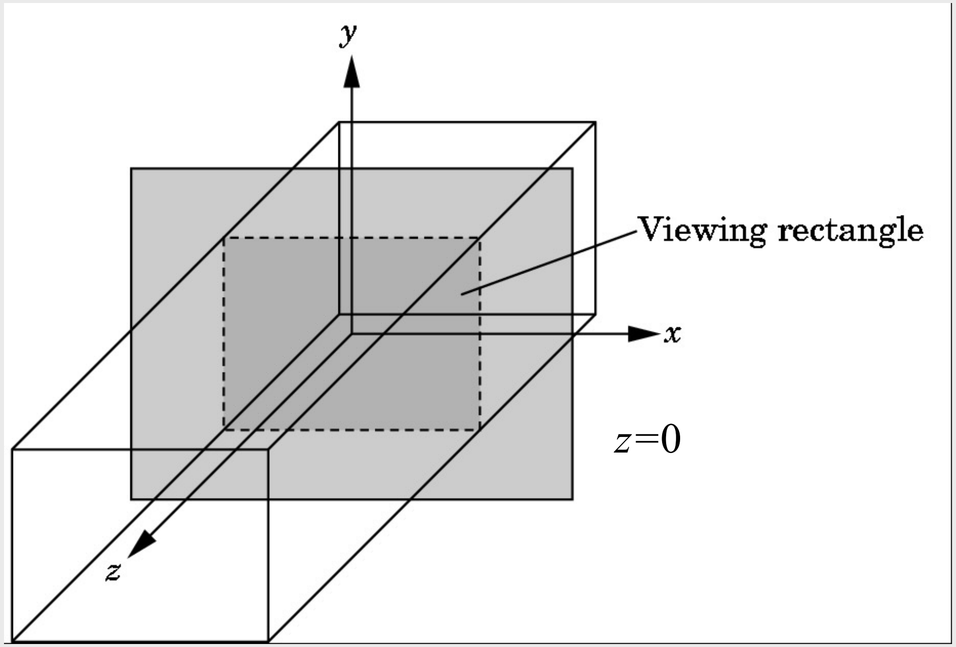
- OpenGL initially places a camera at the origin pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2
- The default projection is orthographic
- Initially, the world and camera coordinate systems are the same



Orthographic Viewing

In the default orthographic view, points are projected forward along the z axis onto the plane $z = 0$.



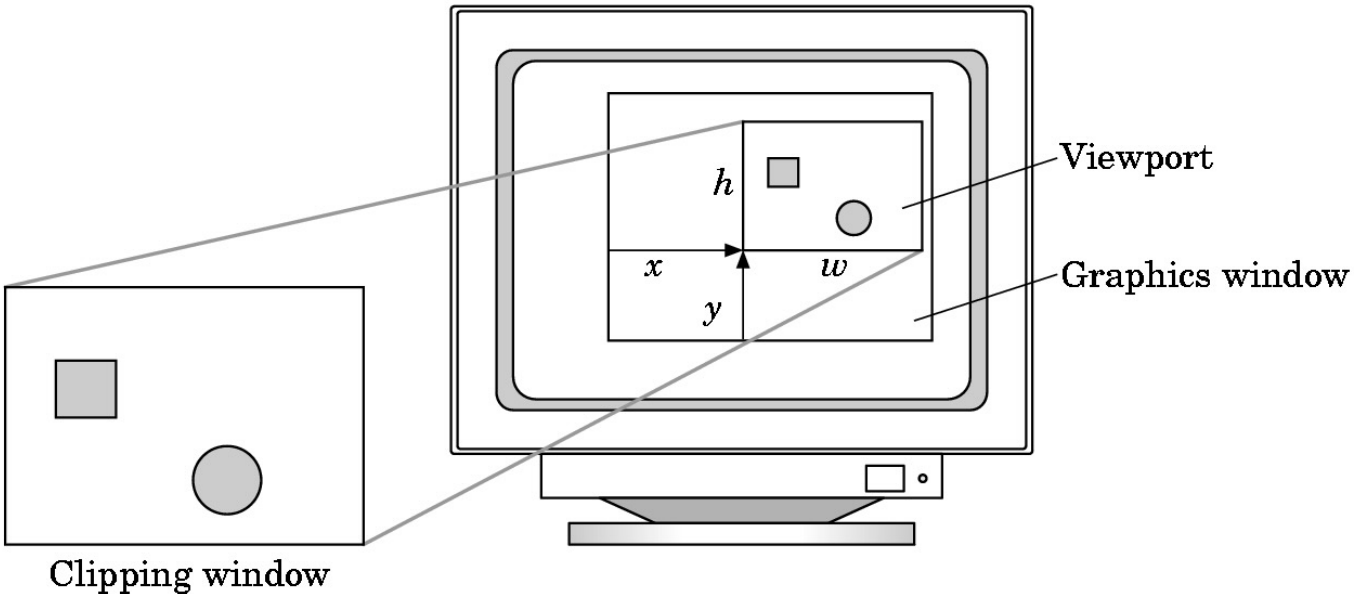


Viewport

- Do not have to use the entire window for the image:

  `glViewport(x,y,w,h)` (can be used in `init()`)

- Values are in pixels (window coordinates)



Transformations and Viewing

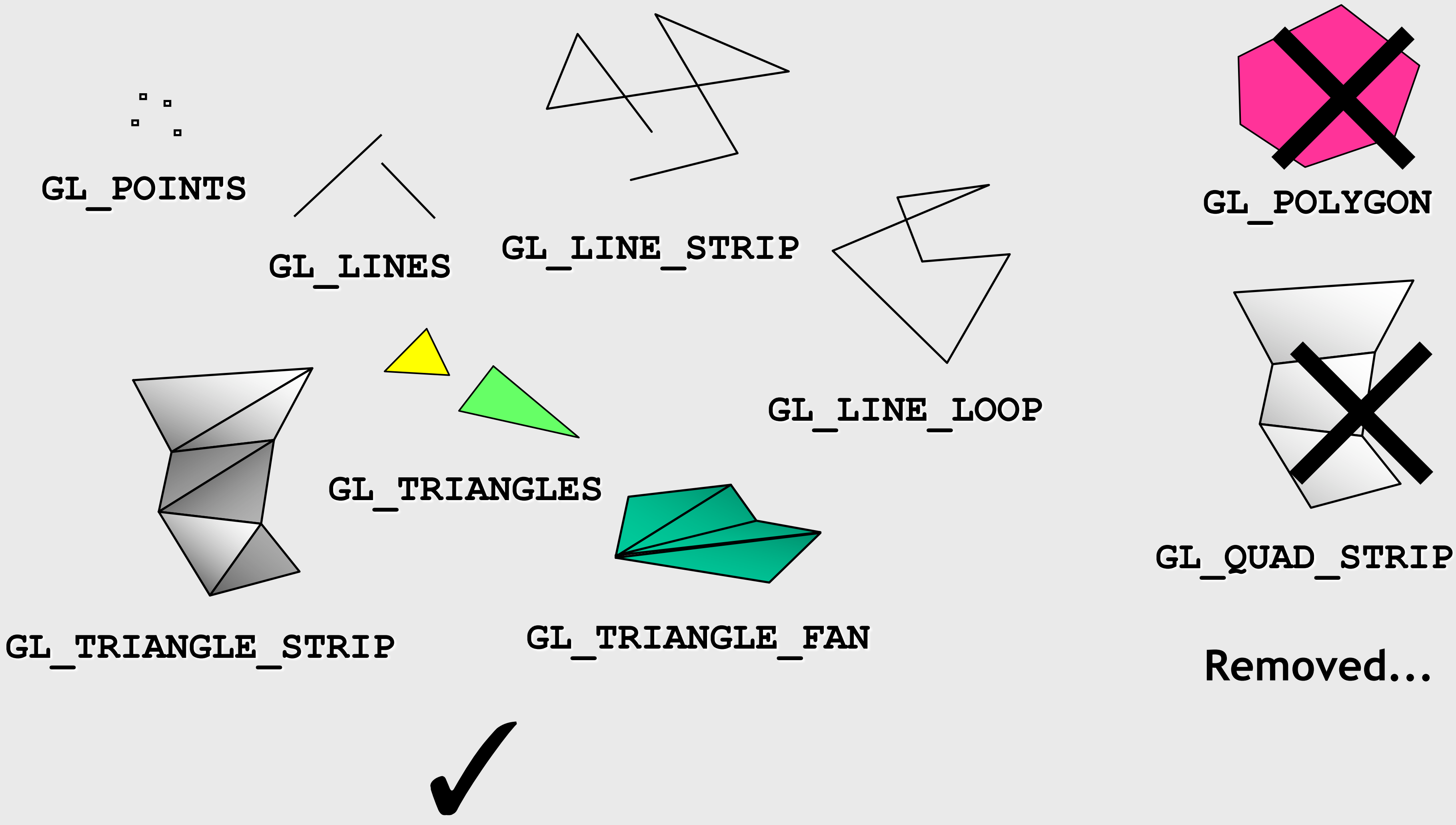
- In OpenGL, projection is carried out by a 4x4 projection matrix
- 4x4 matrices can also be used to specify transfomations such as rotation, translation, scaling, etc.

$$\boldsymbol{p}' = \boldsymbol{M}_{\text{projection}} \boldsymbol{M}_{\text{transform}} \boldsymbol{p}$$

M is 4x4;
p and ***p′*** are points

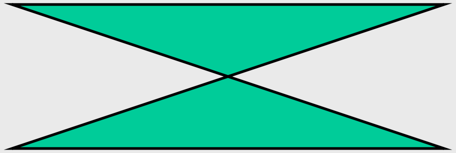
- Pre-OpenGL 3.0 had a set of built-in state variables (4x4 matrices) and functions (such as glRotate(), glTranslate()) to manipulate these matrices
- Deprecated (and then removed by 3.1)
- We have now three choices to set these matrices
  - via application code
  - via shader code using GLSL functions for vector and matrix operations
    - User defined vec.h and mat.h (provided with the textbook code)

OpenGL Primitives



Polygon Issues

- OpenGL will display only triangles
- If needed, application program must tessellate a polygon into triangles to display (triangulation)
- OpenGL 4.1 contains a tessellator that can be carried out in shaders

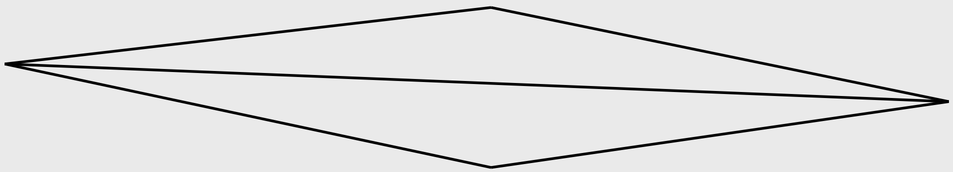


Non-simple polygon

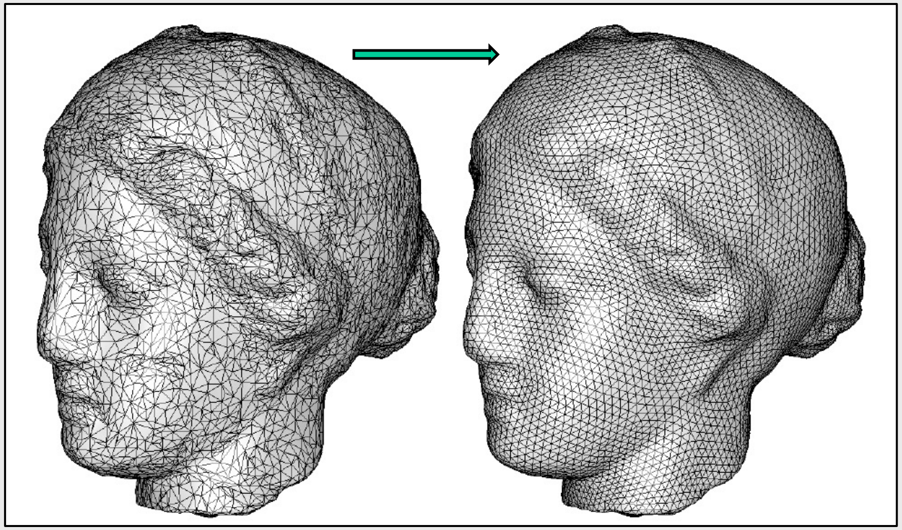
Non-convex polygon

Good and Bad Triangles

- Long thin triangles render badly



- Equilateral triangles render well
- So maximize minimum angle while modeling



Attributes

- Attributes determine the appearance of objects
  - Color (points, lines, triangles)
  - Size and width (points, lines)
  - Polygon mode
    - Display as filled: solid color
    - Display as edges

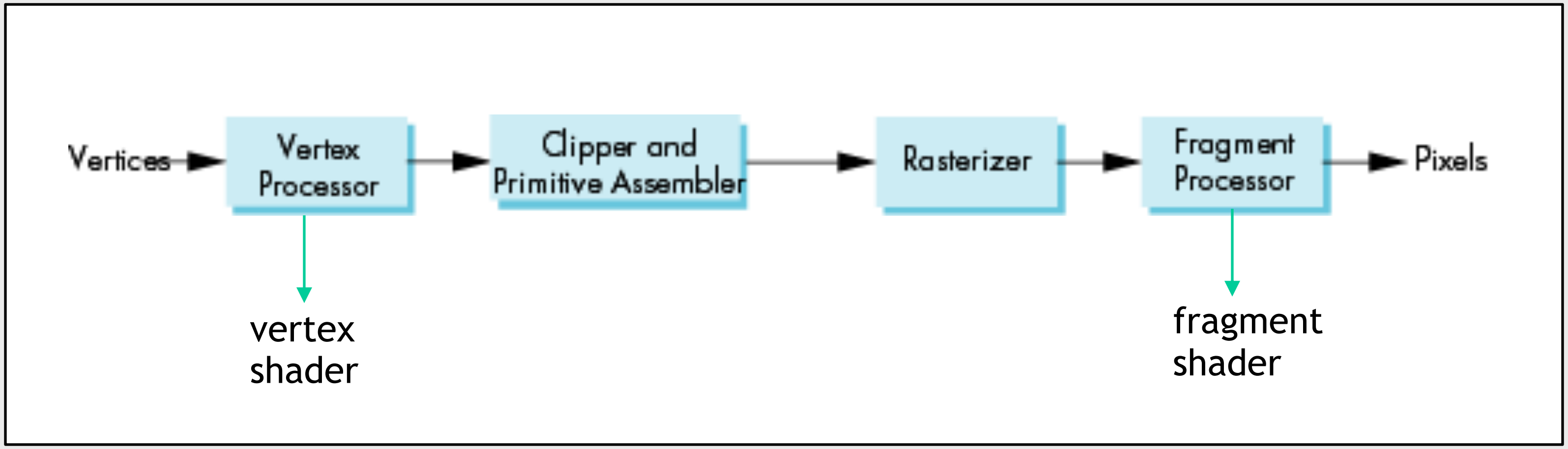
- Only a few attributes (such as `glPointSize`, `glPoygonMode`) are supported by modern OpenGL functions

RGB(A) color

- “**A**” channel encodes transparency.
- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values range from 0.0 (none) to 1.0 (all) using floats, or over the range from 0 to 255 using unsigned char.

Setting Colors

- Colors of pixels are ultimately set in the fragment processor
- Can be specified in either shaders or in the application
- **Application color:** Passed to vertex shader as a vertex attribute or as a uniform variable (next lectures)
- **Vertex shader color:** Passed to fragment shader as varying variable (next lectures)
- **Fragment color:** Can also alter color via fragment shader code



Smooth Color

By default, OpenGL interpolates vertex colors across visible triangles (rasterization)