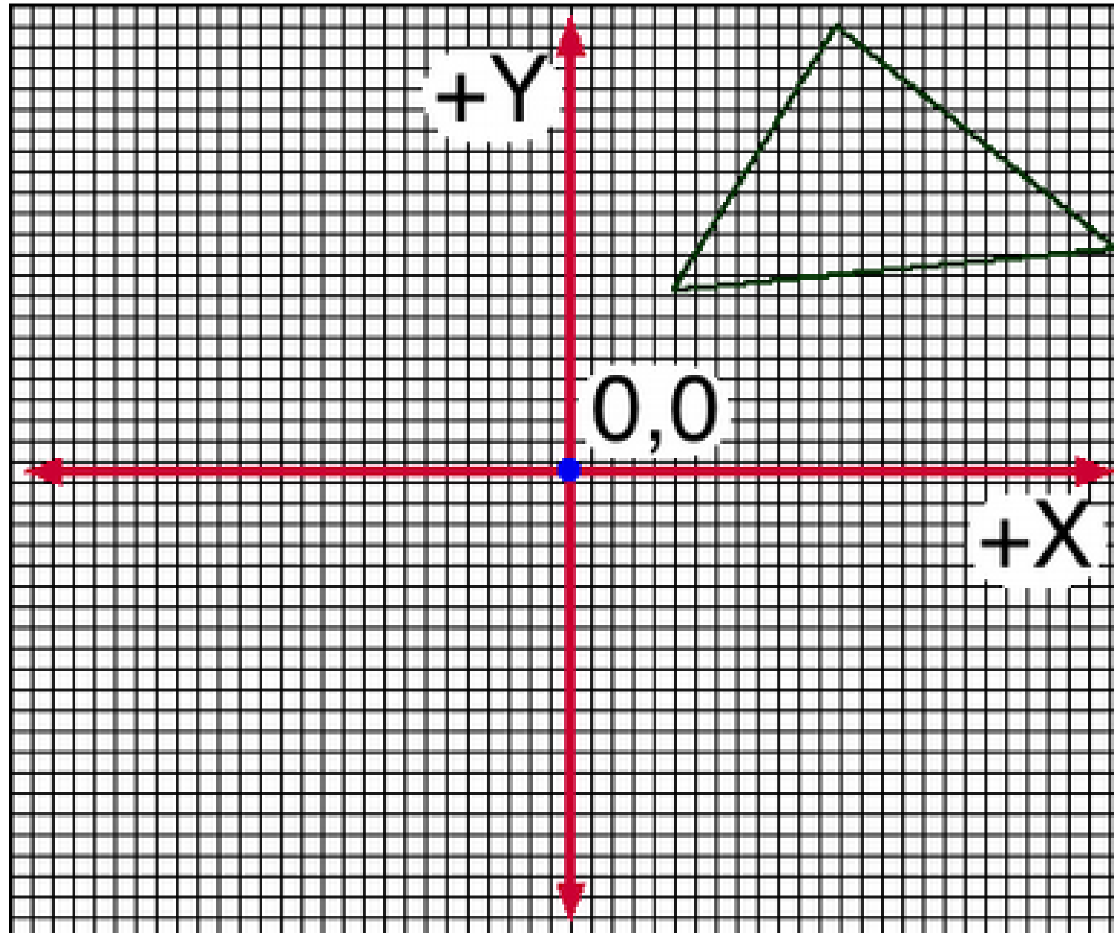
# Graphics Programming

---

527970
Fall 2020
9/17/2020

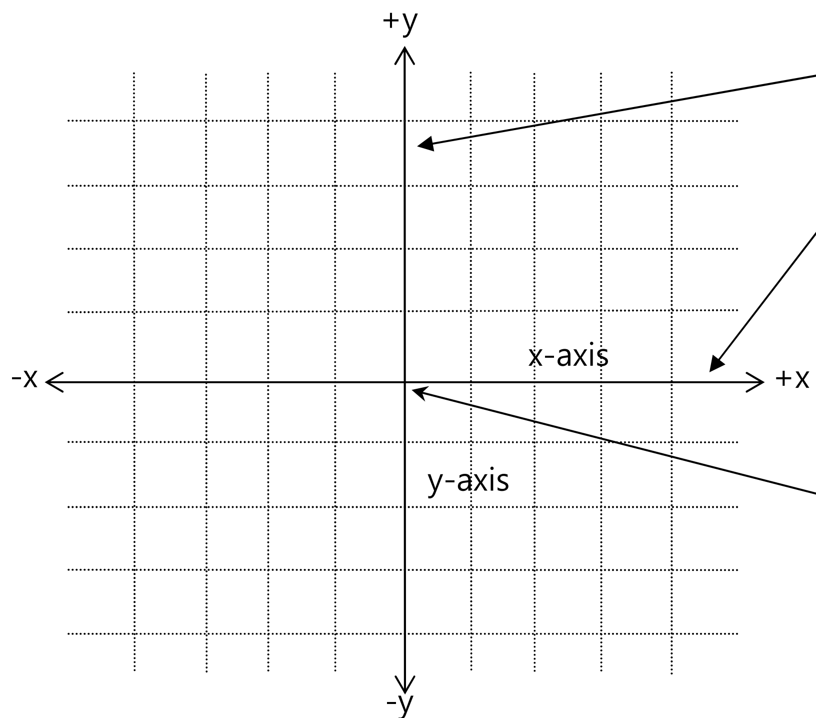
Kyoung Shin Park
Computer Engineering
Dankook University

# Coordinate Systems



# 2D Cartesian Coordinate Systems

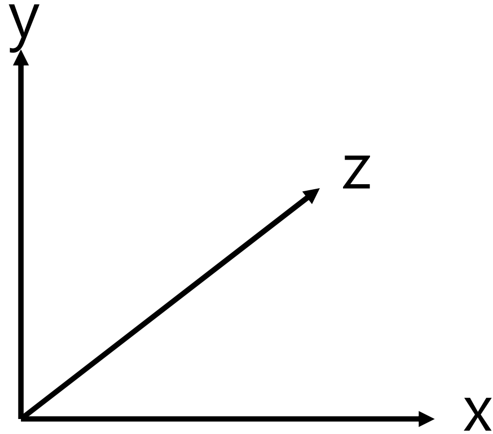
- Cartesian Coordination Systems

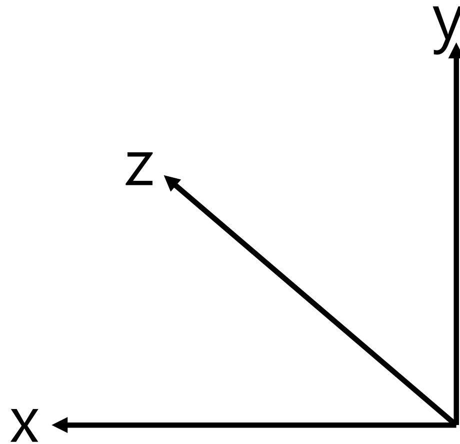



Two axes: **x-axis** and **y-axis**, two straight lines perpendicular to each other, both pass through origin and extends infinitely in two opposite directions

The origin is located in the center of the coordinate system and its value is (0, 0).

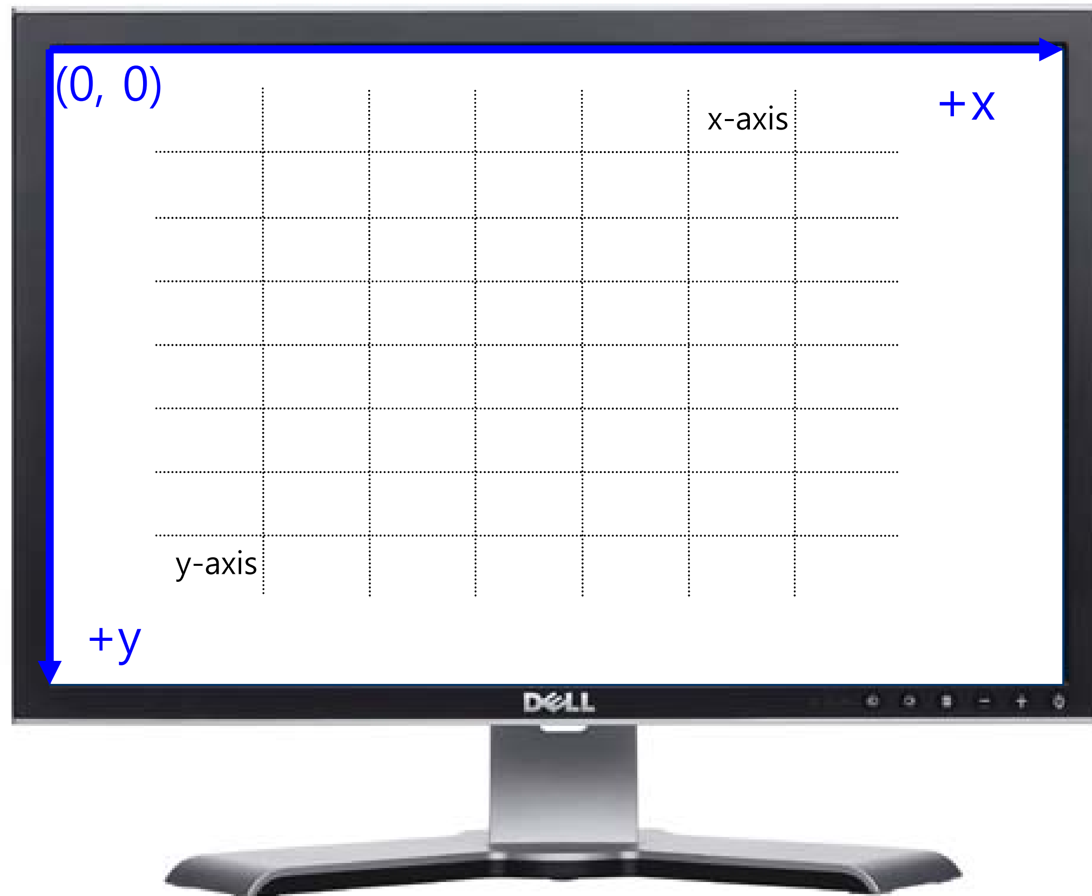
# 3D Cartesian Coordinate Systems



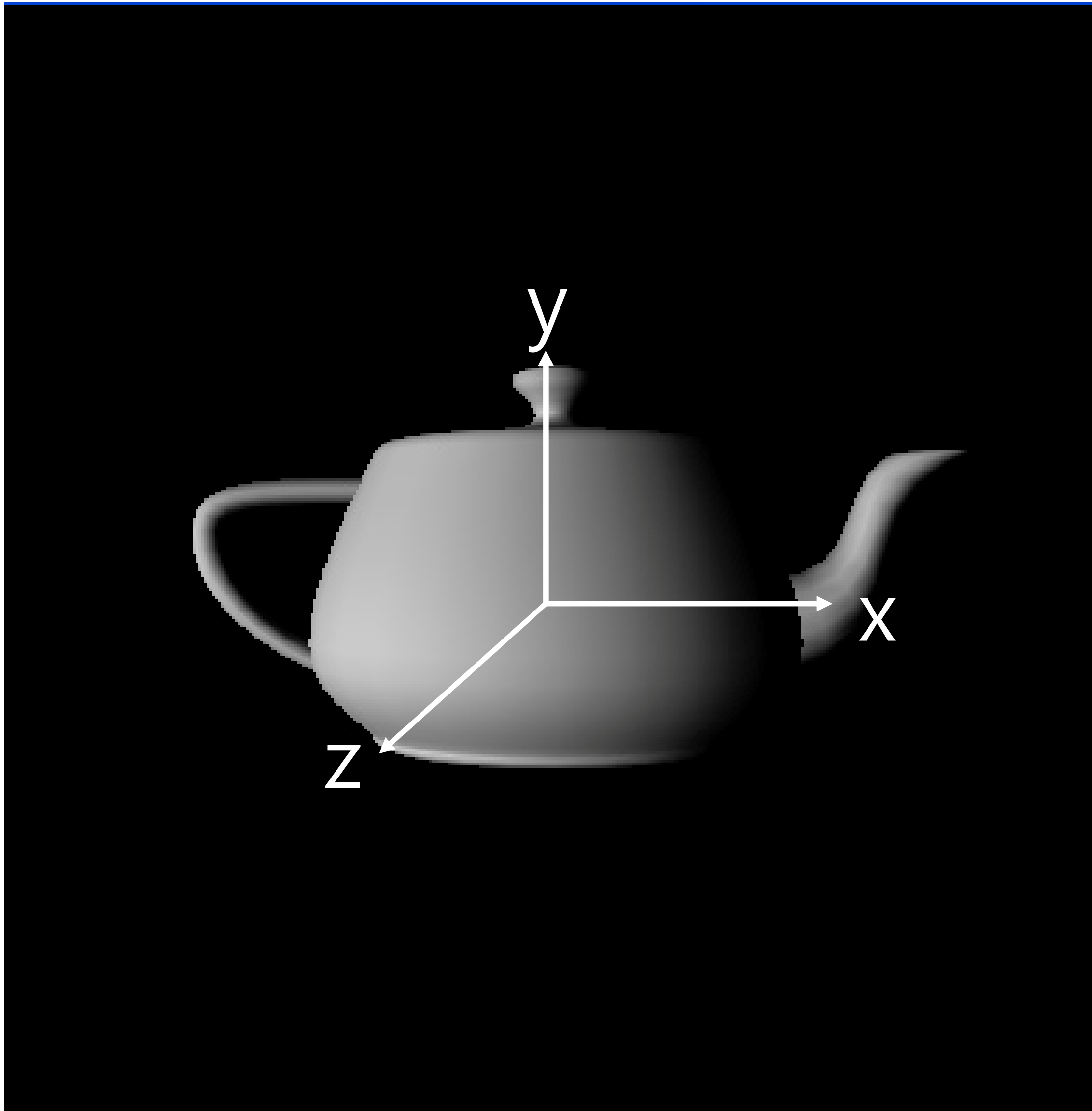

- In left-handed coordinate system, x+ is right, y+ is up, z+ is inside the screen.
- In right-handed coordinate system, x+ is left, y+ is up, z+ is inside the screen.

# Screen Coordinate System




- In screen coordinate system, the origin is located at the top left of the screen and the value is (0, 0). x+ is right. y+ is down.
- 1 unit = 1 pixel

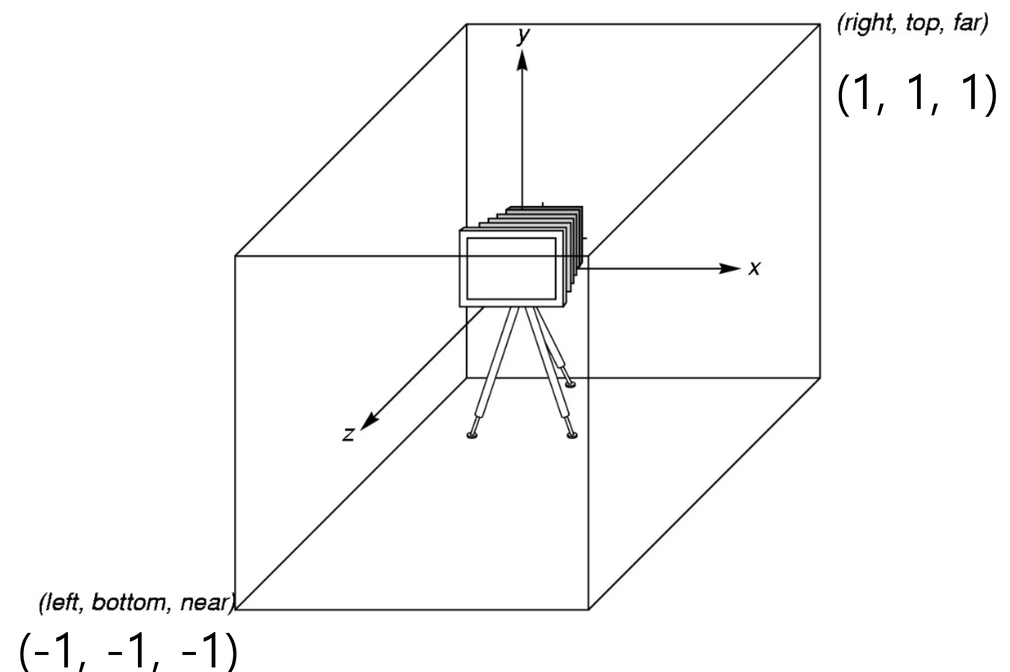
# 3D Coordinate Systems




- OpenGL use a right-handed coordinate system
- x+ is right, y+ is up, z+ is out of the screen.

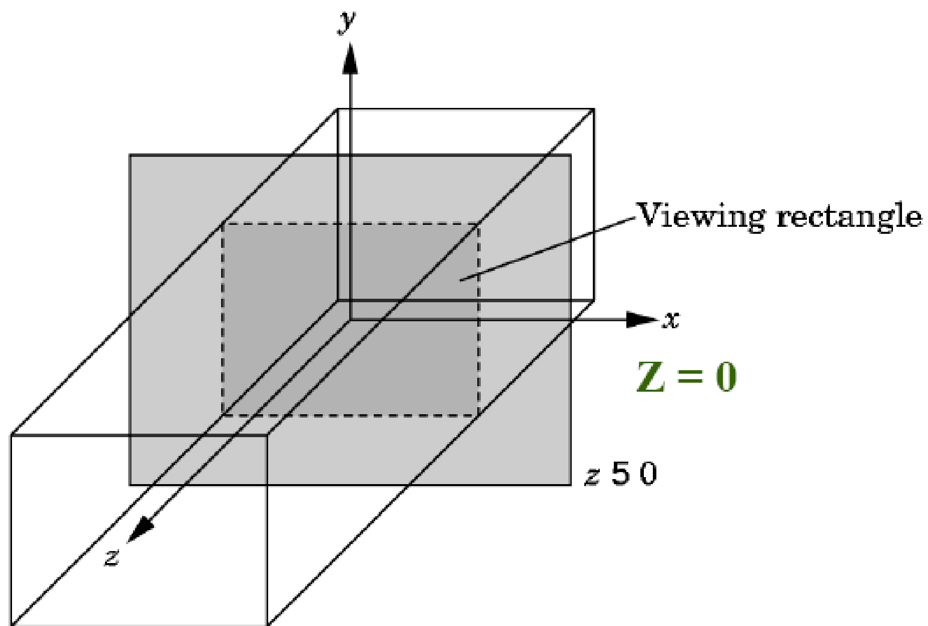
# OpenGL Camera

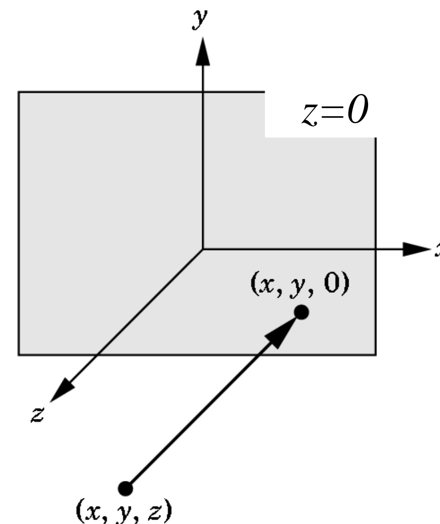
- In OpenGL, the camera is located at the origin of the object's coordinate system and is point at the z-direction.
- By default, a 2x2x2 viewing volume is used.



# Orthographic Viewing

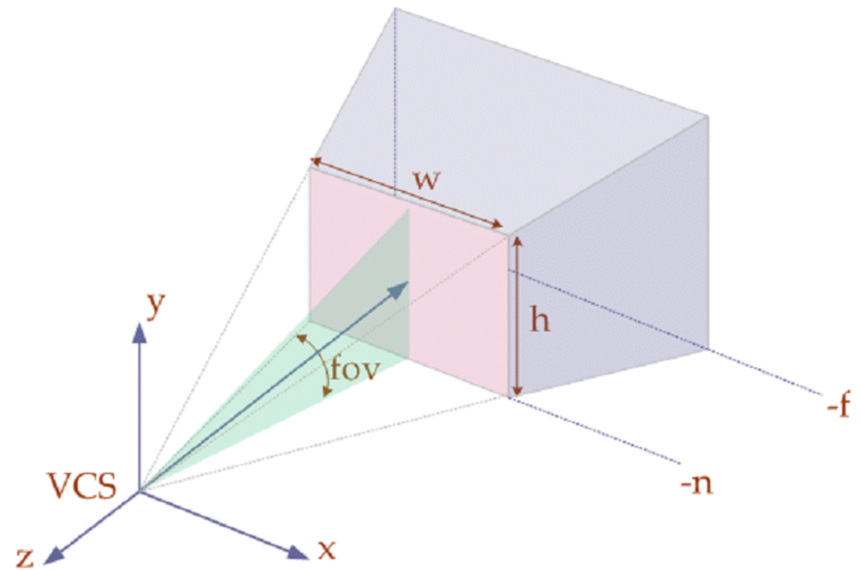
- Orthographic parallel projection
  - Ortho(left, right, bottom, top, zNear, zFar);
  - In orthographical projection, points are projected onto the z=0 plane towards the z- axis.



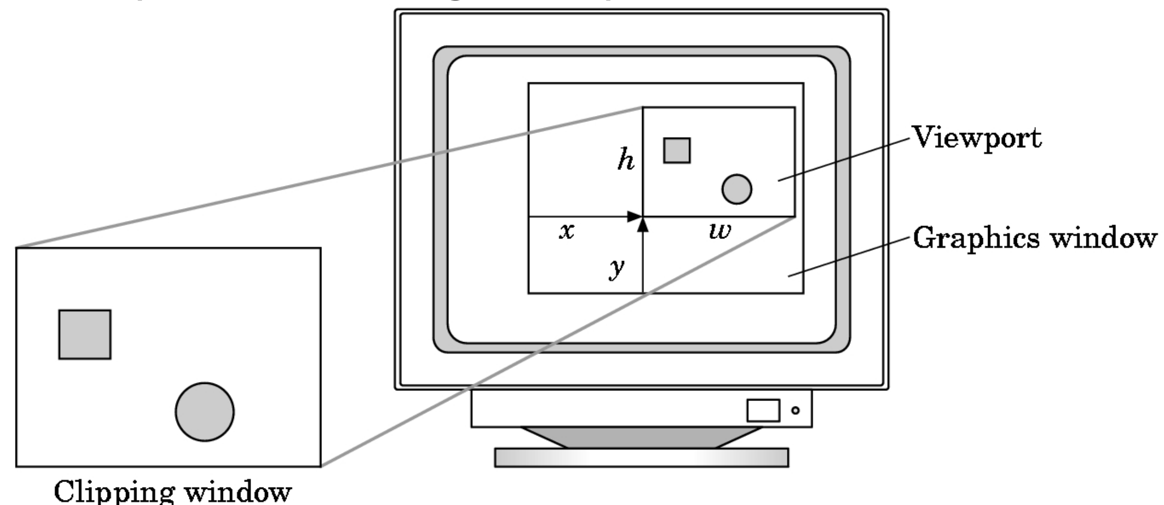
# Perspective Viewing

- Perspective projection
  - Frustum(left, right, bottom, top, zNear, zFar);
  - Perspective(fovy, aspect, zNear, zFar); - Instead of setting up/down/left/right, it uses the y-direction viewing angle (FOV) and the aspect ratio (the value of the width of the nearest clipping plane divided by the height)



# Viewport

- Viewport
  - The space set inside the window. Drawing is restricted to inside the viewport.
- glViewport(x, y, width, height)
  - When the window is first created, the pixel area corresponding to the entire window is set as the viewport; To set a smaller area as a viewport, use glViewport(). Typically the entire window is used as a viewport.
  - In the GLUT reshape function, glViewport() must be included.

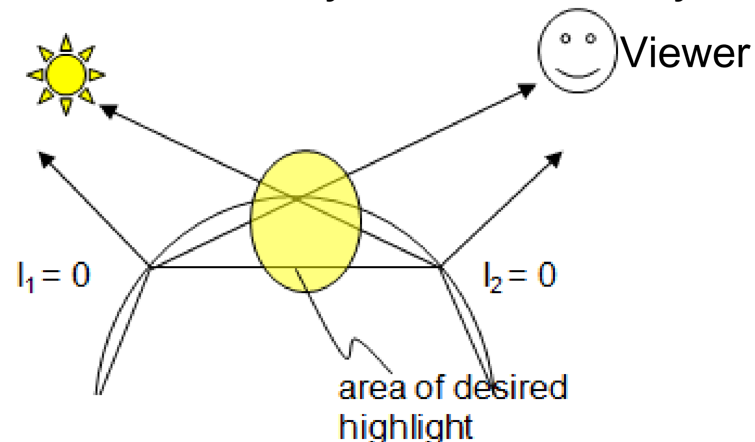


# Transformations and Viewing

- In OpenGL, the transformation matrix is used for transformation.
- Transformation function was used to transform the coordinate system.
- However, transformation functions prior to OpenGL 3.0 are deprecated (recommended not to use anymore).
- 3 choices
  - Application code
  - GLSL functions
  - GLM (OpenGL Mathematics) vector, matrix

# Conventional OpenGL Rendering Pipeline

- It is inefficient in low-spec HW because it determines whether it is applied by examining the options and state variables supported by OpenGL.
- Modified Phong Illumination Model supports only the fixed lighting calculation.
- Fixed shading supports only the Gouraud Shading.
  - After vertex color calculation, it interpolates the vertex color to determine the pixel color.
  - Mach Band may appear or pixel values may be incorrectly calculated.

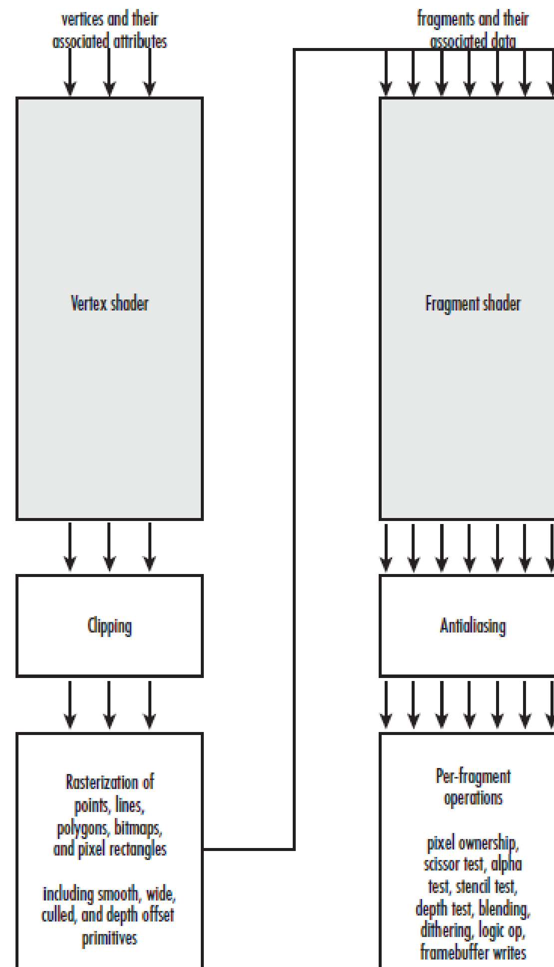


# Extending OpenGL

- Need to support functions to apply advanced graphics techniques with the advent of graphics hardware
- OpenGL supports functions added in new versions as extensions.
  - Maintains backward compatibility by not modifying the previous versions' API.
  - Name the function or macro constant name with a suffix ro identify the extension.
    - _ARB, _EXT, _NV, _ATI, etc
- API to support programmable hardware is provided as an extension function.
  - Instead of using a fixed pipeline, it is possible to use a programmable pipeline that can shade according to the code written by the user.

# Programmable Pipeline

- Various rendering techniques can be applied by creating Vertex Shader and Fragment Shader.



# OpenGL Shader

- Basic Shaders
  - Vertex shader
  - Fragment shader

# Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

# Fragment Shader Applications

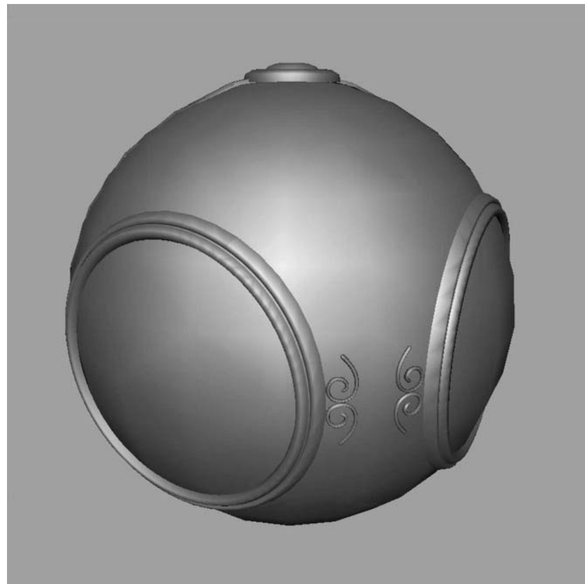
- Per-fragment lighting calculations

per vertex lighting

per fragment lighting

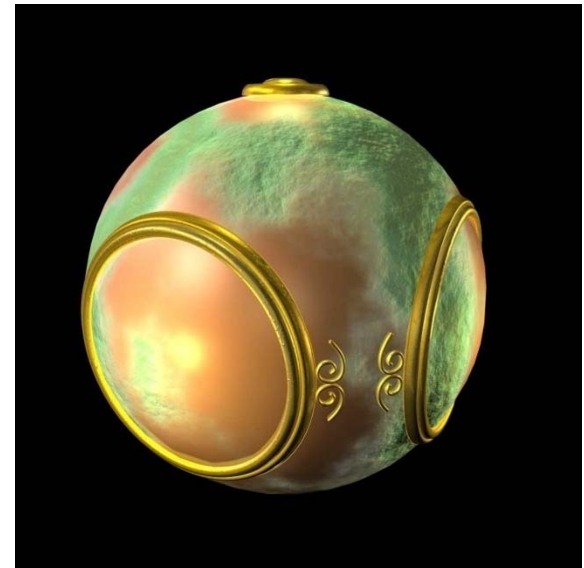
# Fragment Shader Applications

- Texture mapping



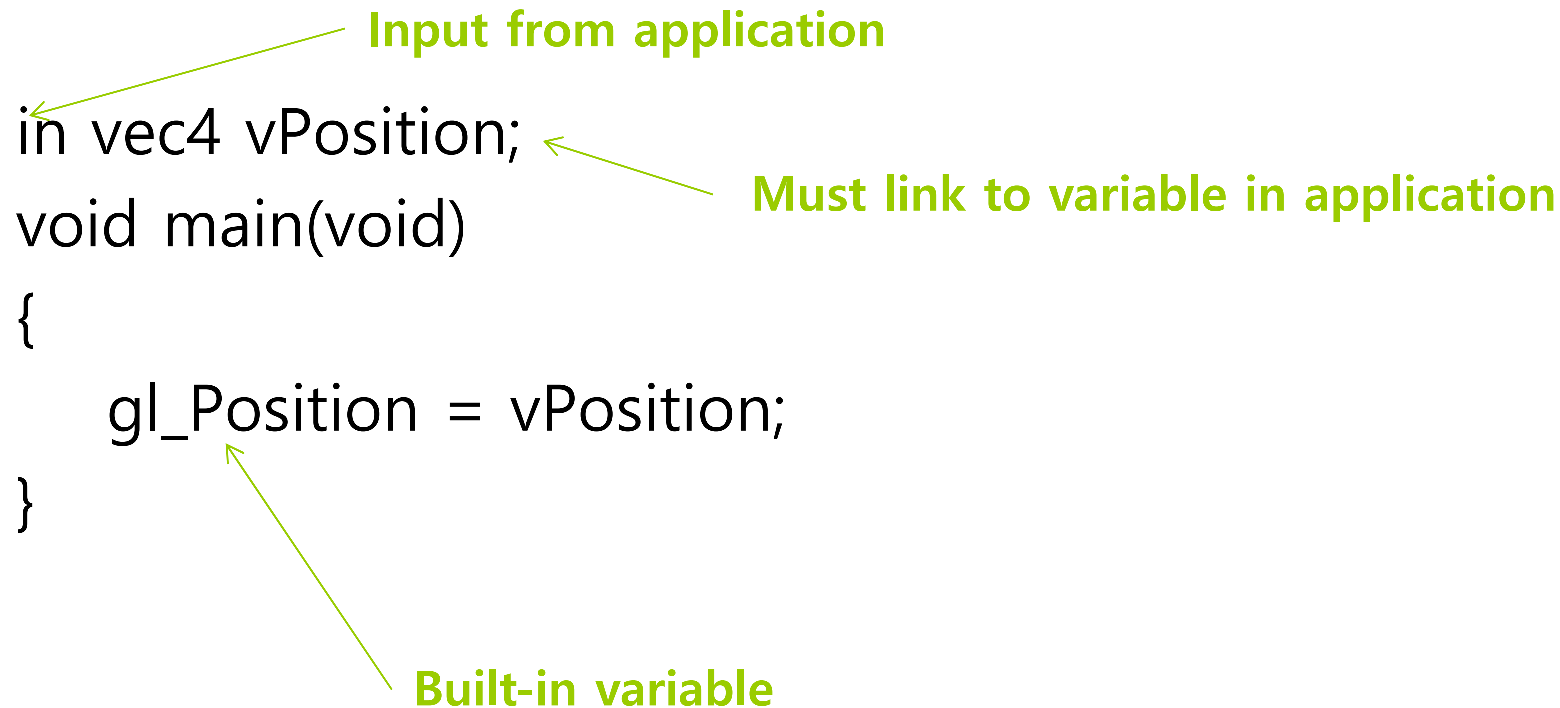
Smooth shading

Environment mapping

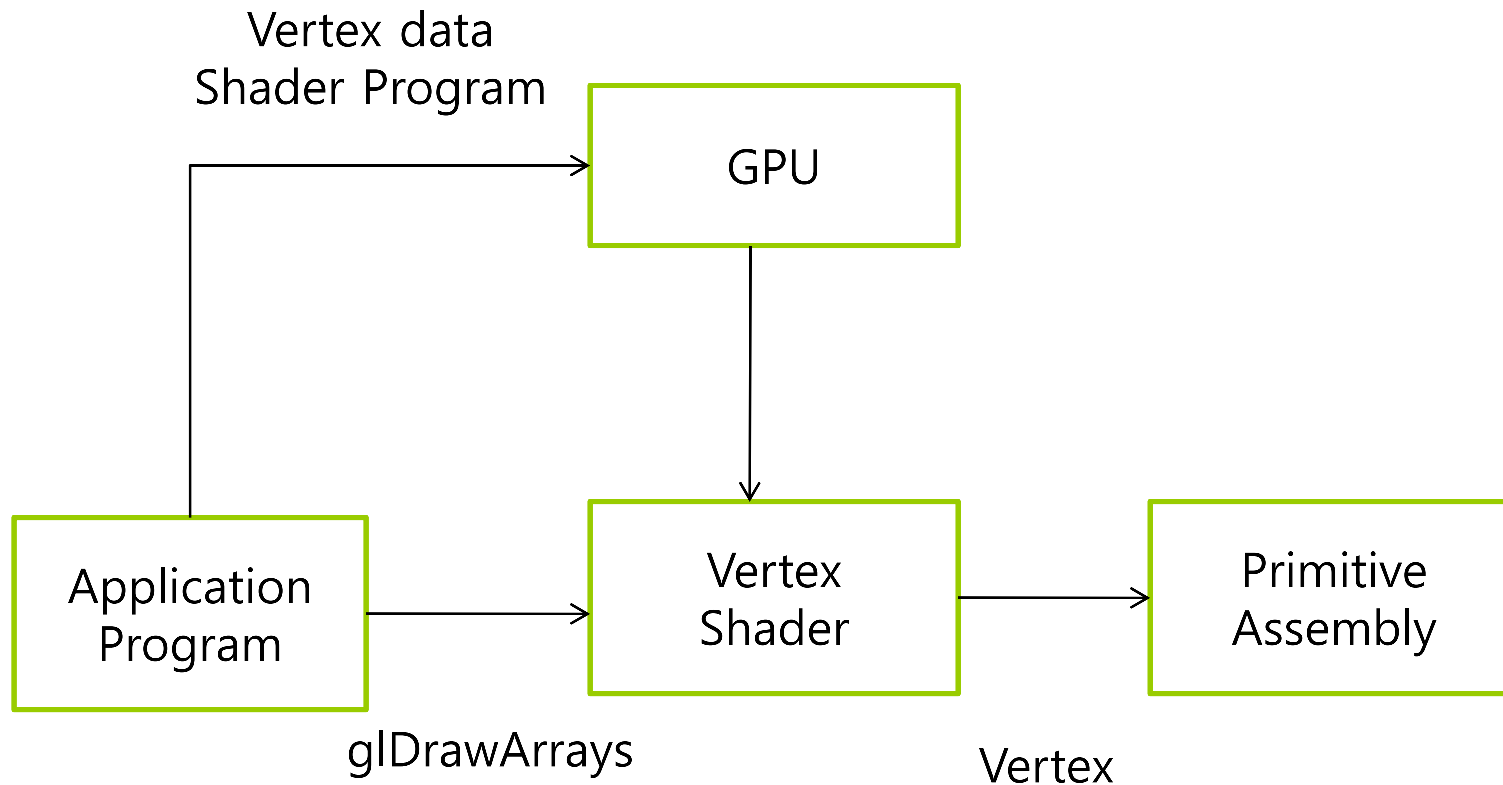


Bump mapping

# Simple Vertex Shader



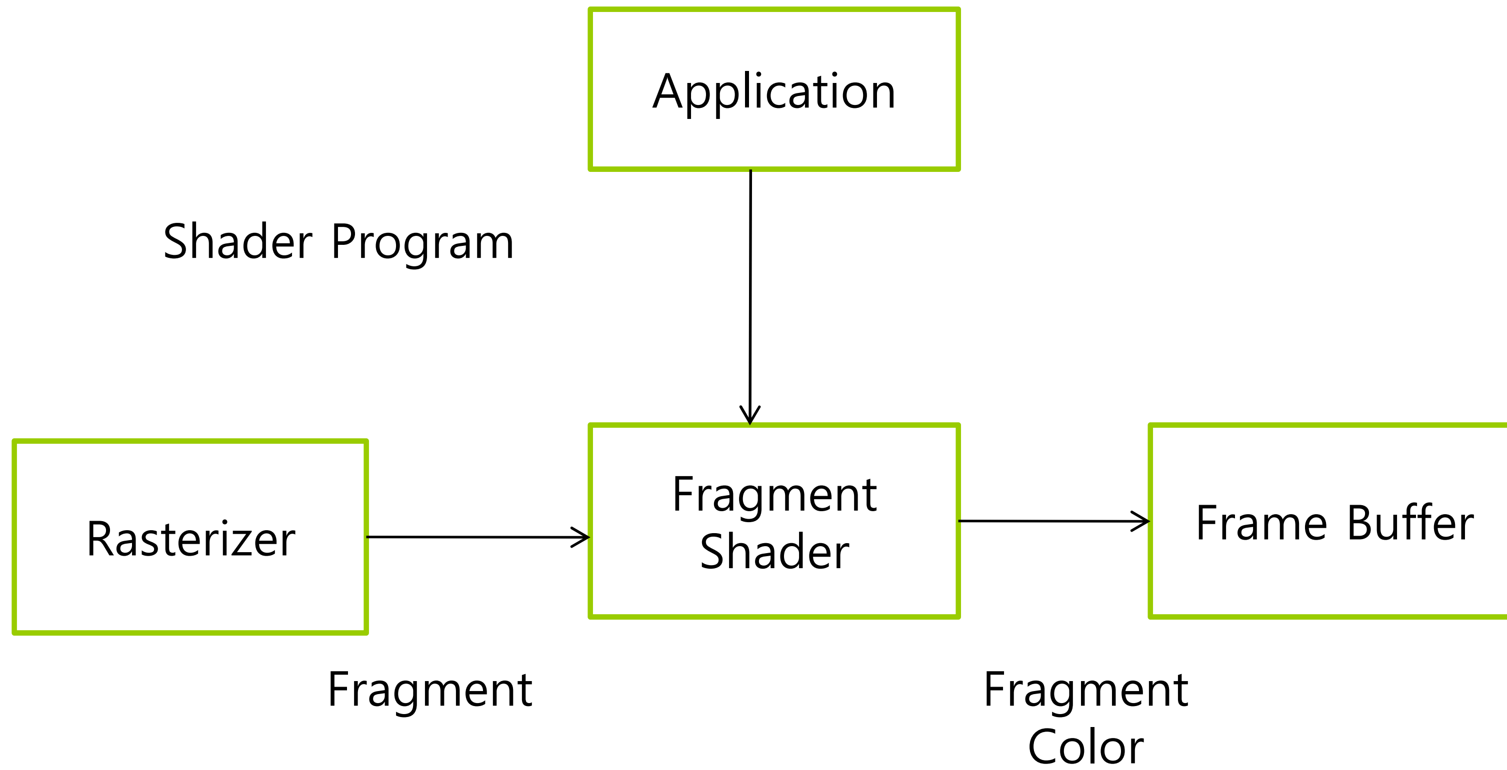
# Execution Model



# Simple Fragment Program

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

# Execution Model



# GLSL Data Types

- C types: int, float, bool
- Vectors
  - float vec2, vec3, vec4
  - int (ivec)와 boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Columns-major
  - m[row][column]
- Texture Sampler: a sampler type with texture access
  - sampler1D, sampler2D, sampler3D, samplerCube
  - sampler1DShadow, sampler2DShadow
- C++ style constructors
  - vec3 a = vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

# GLSL Pointers

- There is no concept of pointer in GLSL.
- C language structure is used to pass it to the function.
- Matrices and Vectors are basic types, and can be used as parameter inputs or return type outputs to GLSL functions.

```
mat3 func(mat3 a)
```

# GLSL Qualifiers

- Variable Qualifiers
  - `const` – constant
  - `attribute` – This is a global variable, can be changed for each **vertex**, and the value is changed from the **OpenGL program** to Vertex Shader. This qualifier is used only on Vertex Shader. It is read-only in shader.
    - Built-in vertex attribute: **gl_Position**
    - User-defined vertex attribute: **in vec3 velocity**
  - `uniform` – This is a global variable, can be changed for each **Primitive**, and the value is changed with the shader in the **OpenGL program**. This qualifier can be used on both Vertex Shader and Fragment Shader. It is a constant in shader.
  - `varying` – This is variable passed from Vertex Shader to Fragment Shader. Write is allowed in the Vertex Shader, but read-only in Fragment Shader.
    - In the latest version, Vertex Shader is used as "out", and Fragment Shader is used as "in".
    - User-defined varying variable: **out vec4 color;**

# Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;
void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

# Required Fragment Shader

```
in vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
// in latest version use form
// out vec4 fragcolor;
// fragcolor = color_out;
```

# GLSL Operators and Functions

- General C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

# GLSL Constructor

- Constructor
  - Variable initialization uses the C++ constructor
    - vec3 n = vec3(0.0, 1.0, 0.0);
  - Constructor can be used in expressions other than initialization
    - greenColor = myColor + vec3(0.0, 1.0, 0.0);
  - Assigning a scalar value to a vector assigns it to all elements of the vector.
    - ivec4 whiteColor = ivec4(255);
  - Can mix scalars, vectors, and matrices in the constructor, and are discarded if there are extra elements
    - vec4 v = vec4(x, vec2(y, z), w);
  - If you specify a single scalar value, it becomes a diagonal matrix (non-diagonal elements are padded with zeros)
    - mat2 m = mat2(1.0, 0.0, 0.0, 1.0);
    - mat2 m = mat2(1.0);
  - Typecasting is only possible through a constructor
    - float i = 4.7; int j = int(i);

# GLSL Swizzling and Selection

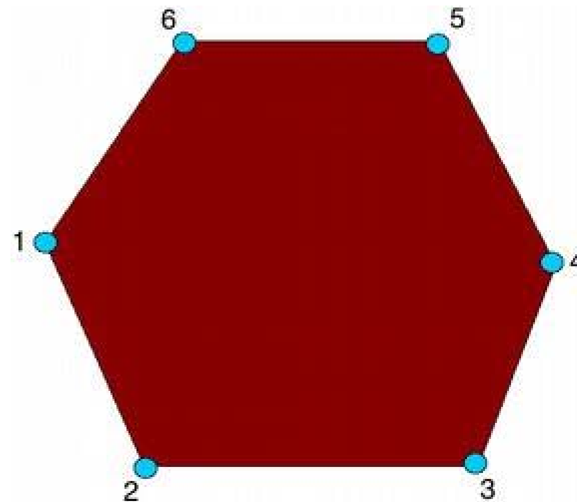
- Access vector or matrix elements using [] or (.) operator
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - vec3 a = vec3(0.0, 0.0, 1.0); **a[2], a.b, a.z, a.p** are all the same
  - mat3 m = mat3(1.0); float element21 = m[2][1]; // 0.0
  - mat3 m = mat3(1.0); vec3 column1 = m[0]; // (1, 0, 0)
- Can be repositioned and duplicated using element selector
  - vec3 myZYX = s.zyx;
- Only some elements of the vector can be modified using the element selector
  - vec4 a; a.yz = vec2(1.0, 2.0);

# GLSL Passing Values

- All types except arrays can be used as the return type of the function.
- All types including arrays and structures can be used as function arguments.
- **Since it is call only with “Call by value”,** you can specify whether the argument value within the function can be changed using the following qualifier.
  - **in** (default)
  - **const in**
  - **out**
  - **inout** (deprecated)

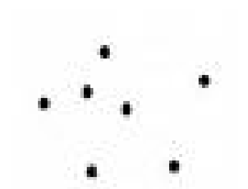
# OpenGL Geometry

- The most basic element for expressing each object
- In real-time graphics, linear primitives are mainly used, which is the simplest form of graphics expression.
  - Point, vertex
  - Line segments
  - Polygon
  - Polyhedron



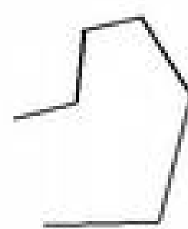
# OpenGL Geometry Primitives

GL_POINTS

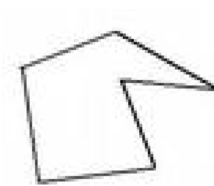


GL_LINES

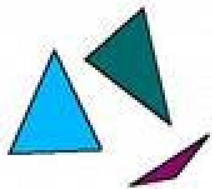
GL_LINE_STRIP



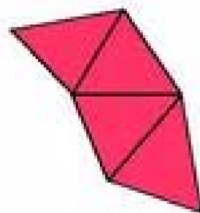
GL_LINE_LOOP



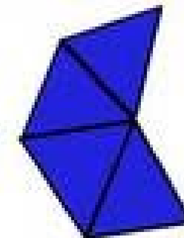
GL_TRIANGLES



GL_TRIANGLE_STRIP

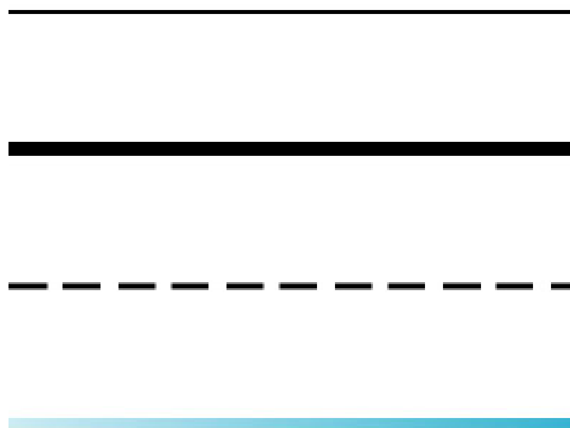


GL_TRIANGLE_FAN

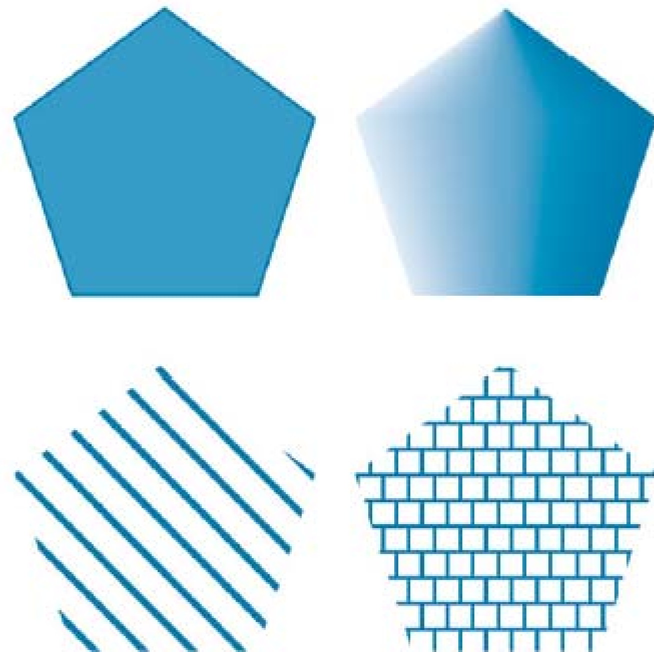


# OpenGL Attributes

- Each geometry primitive has properties. Properties control how basic elements can appear on the screen.
  - Color
  - Line thickness
  - Line styles
  - Polygon patterns



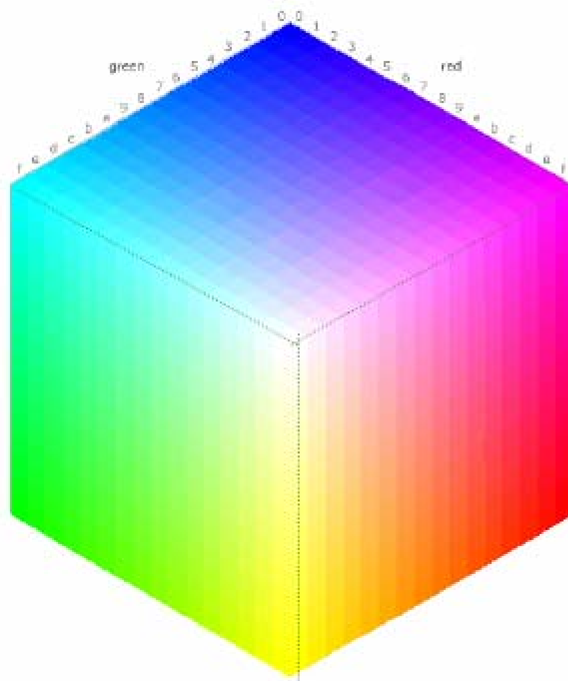
Line thickness and styles

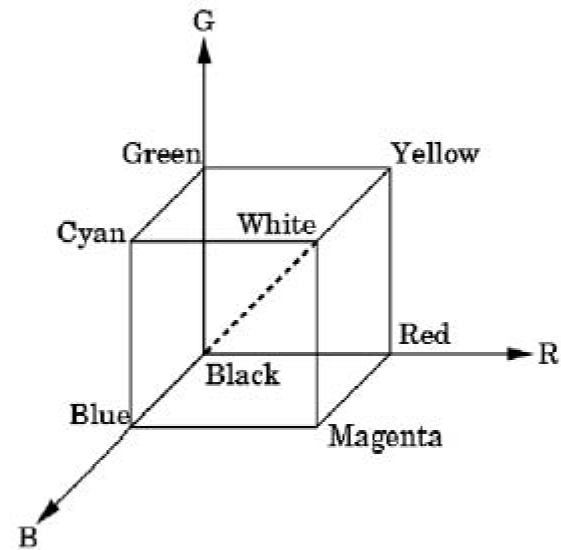


Polygon filling

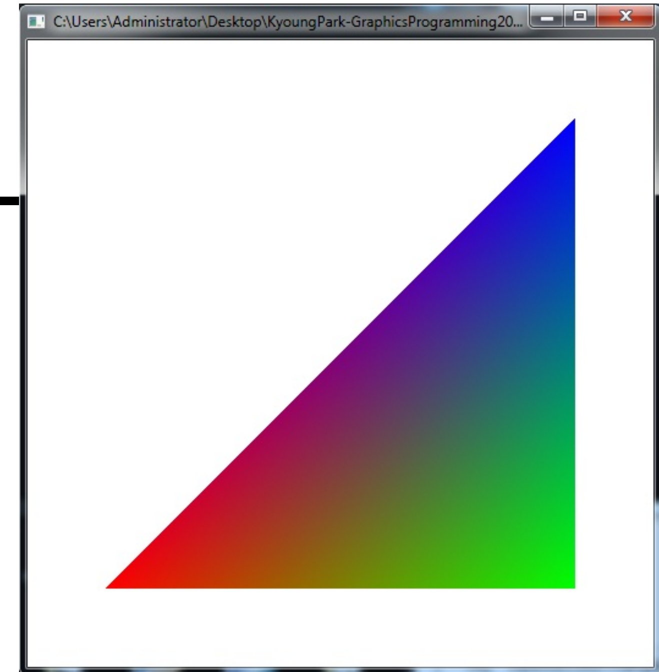
# OpenGL Attributes

- OpenGL Color Model
  - RGB (Red Green Blue) or RGBA(Red Green Blue Alpha)
  - RGB colors are separated and stored in the framebuffer.



# Color Triangle




```
const float vertexColor[] = { 1.0f, 0.0f, 0.0f, 1.0f, 0.0f,
1.0f, 0.0f, 1.0f, 0.0f,              0.0f, 1.0f, 1.0f };
const float vertexPositions[] = { -0.75f, -0.75f, 0.0f, 1.0f,
0.75f, -0.75f, 0.0f, 1.0f,           0.75f, 0.75f, 0.0f, 1.0f };
void SetData() {
    glGenVertexArrays(1, &vao);          // vao
    glBindVertexArray(vao);
    glGenBuffers(2, &vbo[0]);            // vbo
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);        // vertex position
    glBufferData(GL_ARRAY_BUFFER, 12*sizeof(GLfloat), vertexPositions, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);        // vertex color
    glBufferData(GL_ARRAY_BUFFER, 12*sizeof(GLfloat), vertexColor, GL_STATIC_DRAW);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);
    glBindVertexArray(0);
}
```