Computer Graphics I

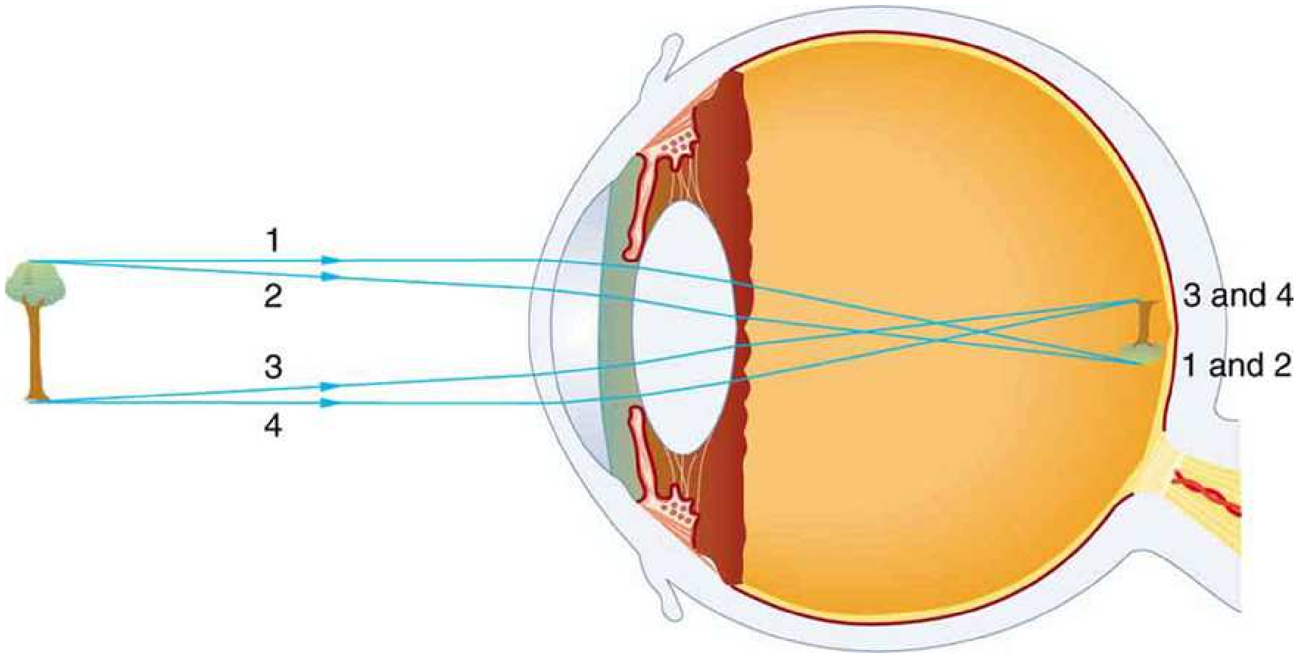
Lecture 2: The first graphics program

Xiaopei LIU

School of Information Science and Technology
ShanghaiTech University

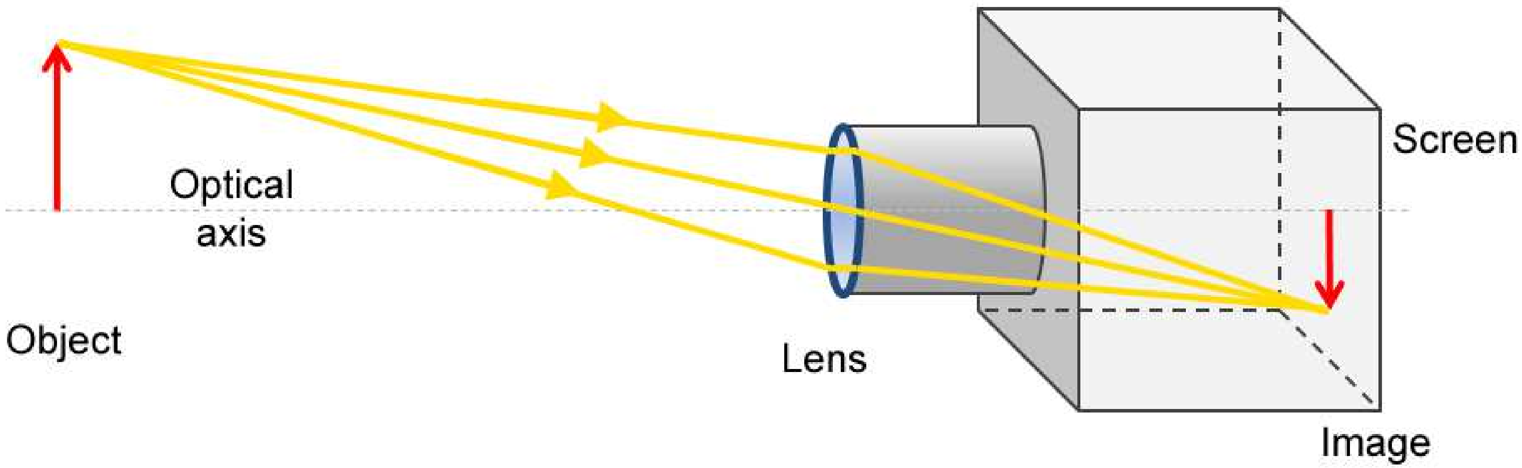
Producing images

- **How can we generate images?**
  - Image formation process in our eye



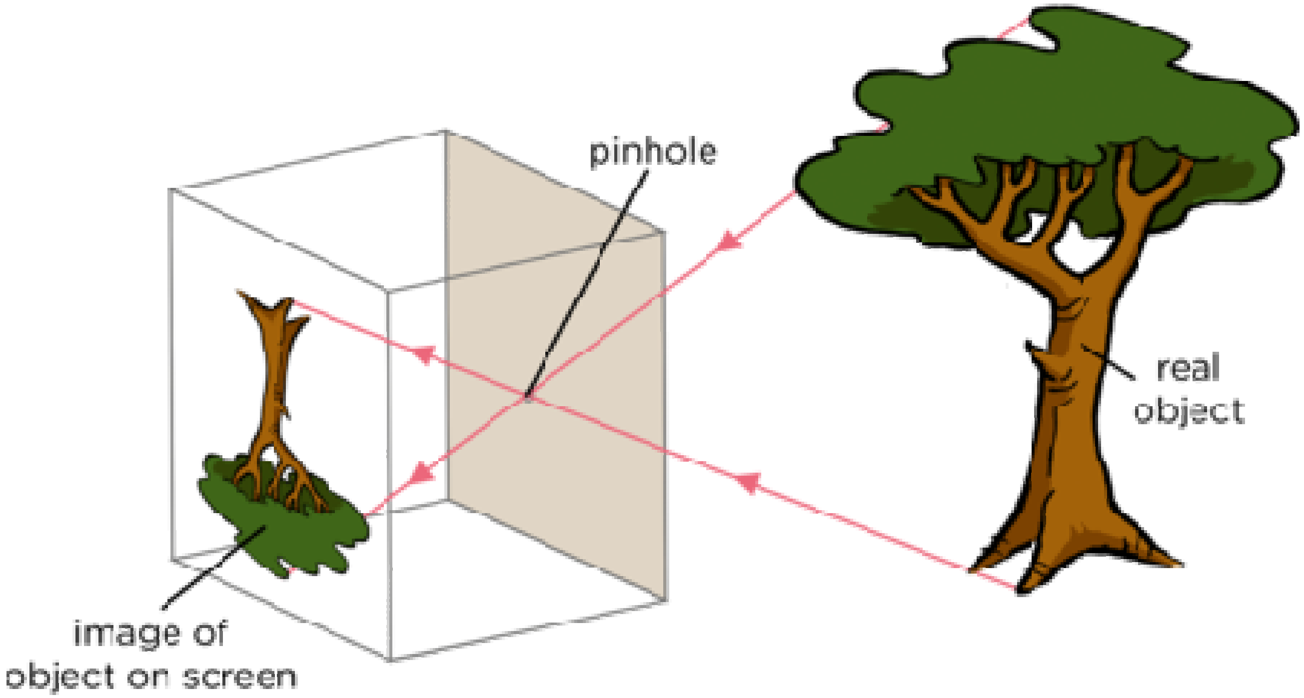
Camera system

- **Imaging in camera system through lens**



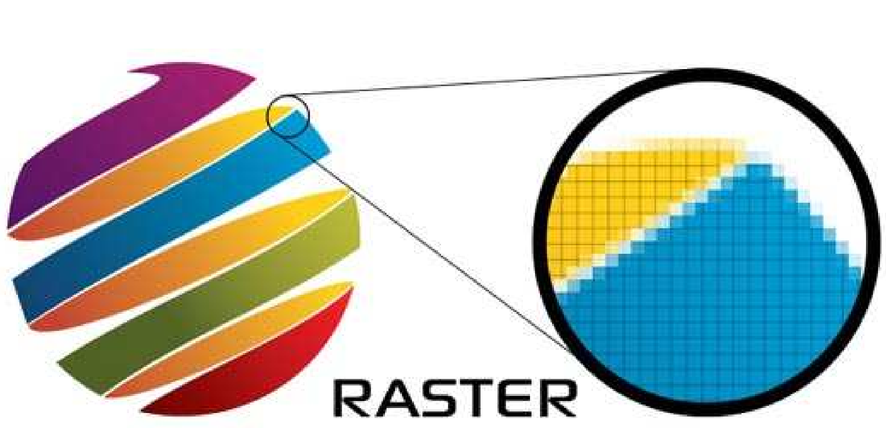
Virtual camera model

- **Camera model**
  - Pinhole camera

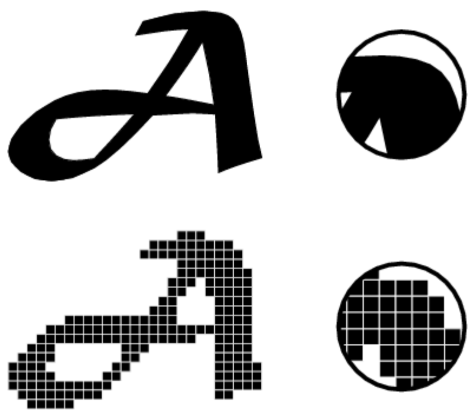


Digital image

- **A two-dimensional rectangle array (2D grid)**
  - Each grid element stores bits to represent color (bitmap)
  - Raster image (common) / vector image
  - Pixel/fragment: the element in the 2D array



raster image



vector v.s. raster image

Image resolution

- **Usually refer to grid resolution**
  - How many samples per grid dimension
  - Higher resolution represents more details

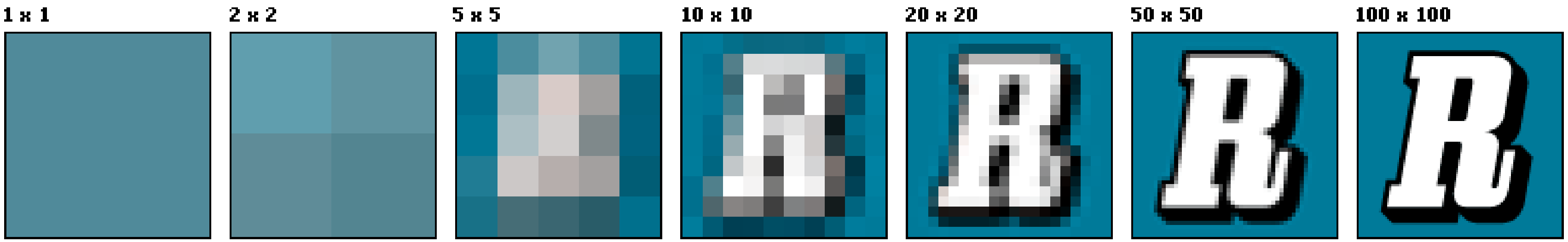
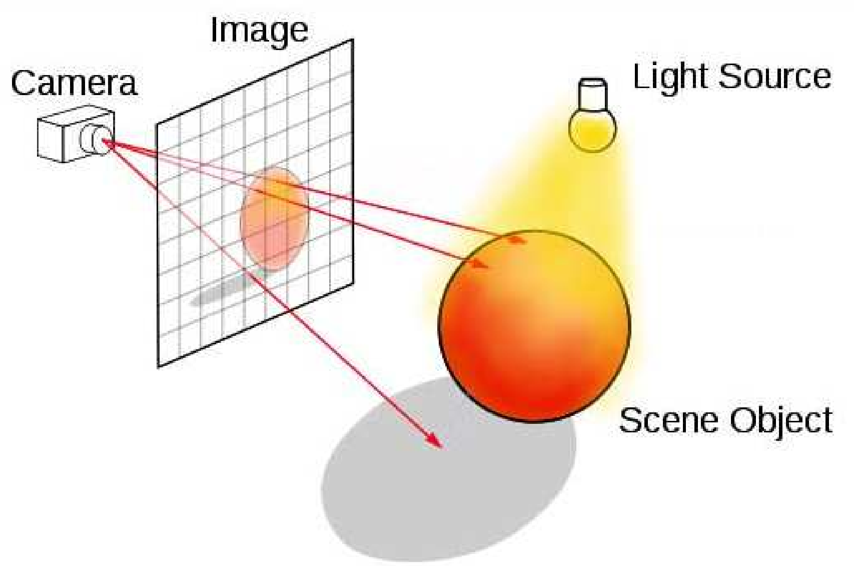
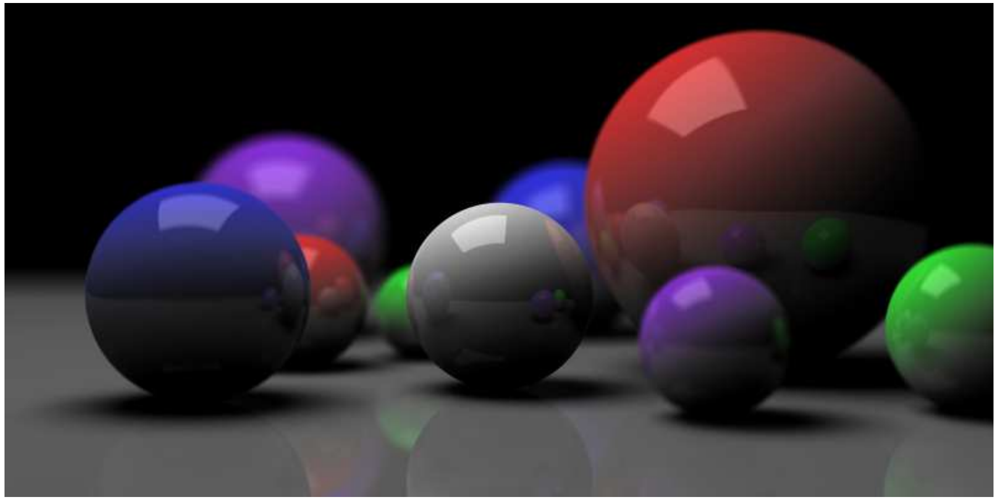


Image formation

- **Imaging with virtual camera**
  - Imaging plane in front of the camera (projection)





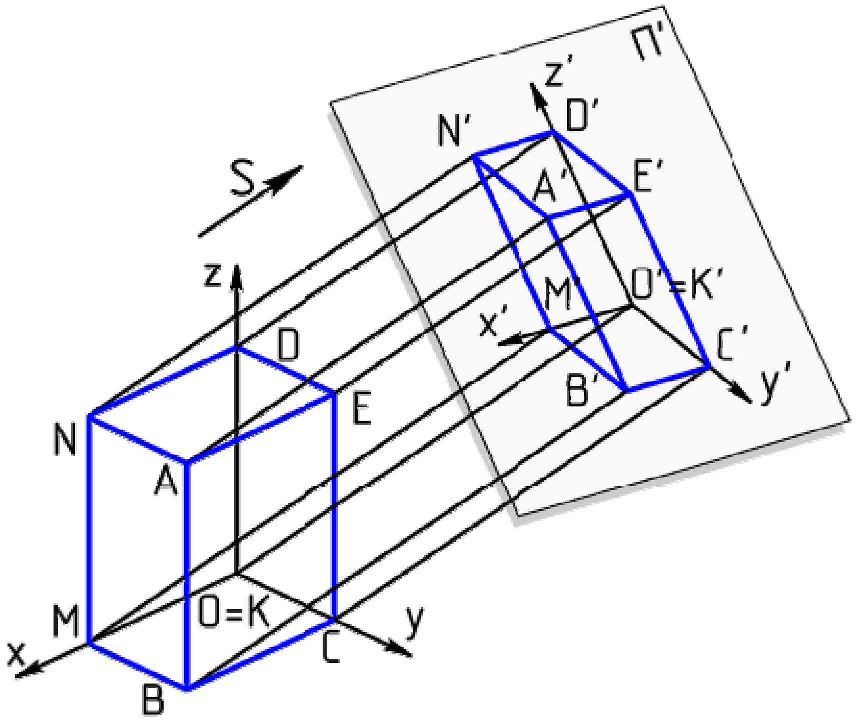
OpenGL

- **A cross-language, cross-platform application programming interface (API)**
  - Rendering 2D and 3D vector graphics
  - Typically used to interact with a graphics processing unit (GPU)
  - Started in 1991 by Silicon Graphics Incorporation (SGI)
  - https://en.wikipedia.org/wiki/OpenGL
  - https://www.opengl.org/
  - Documentation: https://www.khronos.org/registry/OpenGL/index_gl.php

Step 1: Projection

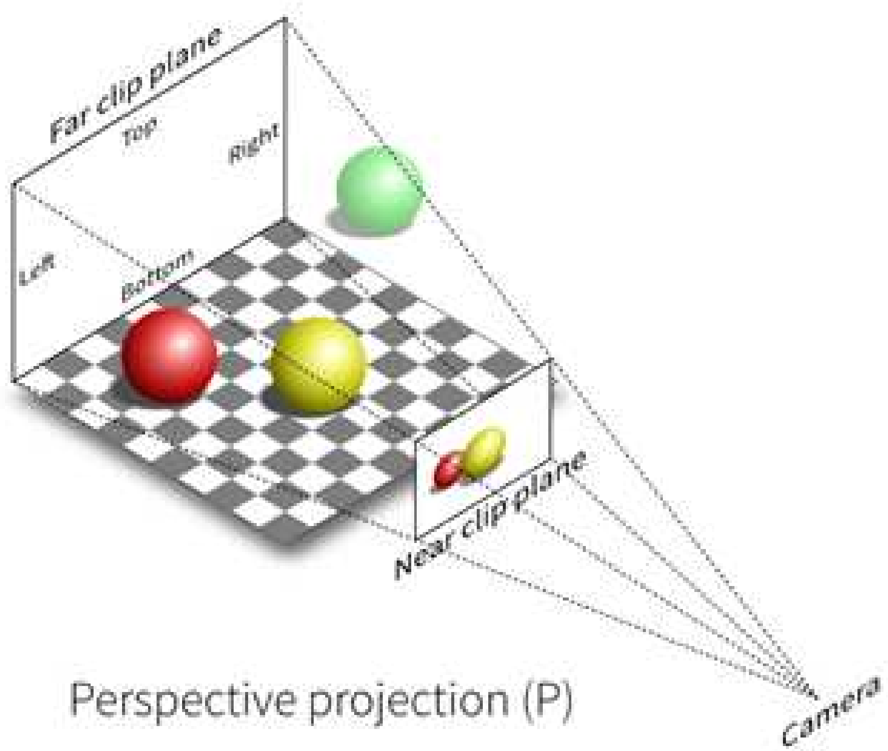
3D projection

- **Any method of mapping three-dimensional points onto a surface**
  - Linear or nonlinear
  - Most commonly, project onto a two-dimensional plane



3D projection

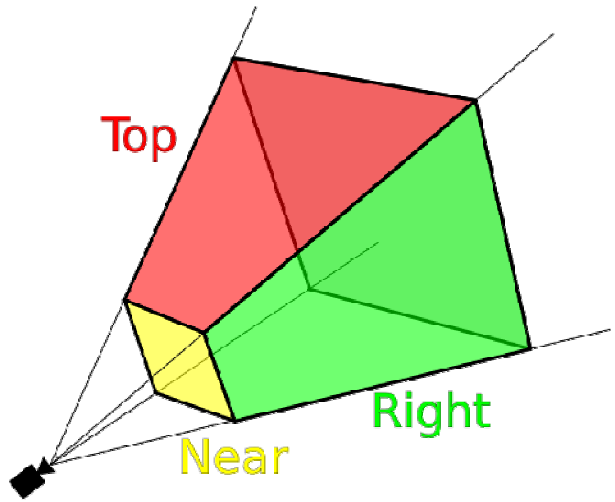
- **Perspective projection**
  – Optical rays converge at a point with finite distance to the projection plane



Perspective projection (P)

3D projection

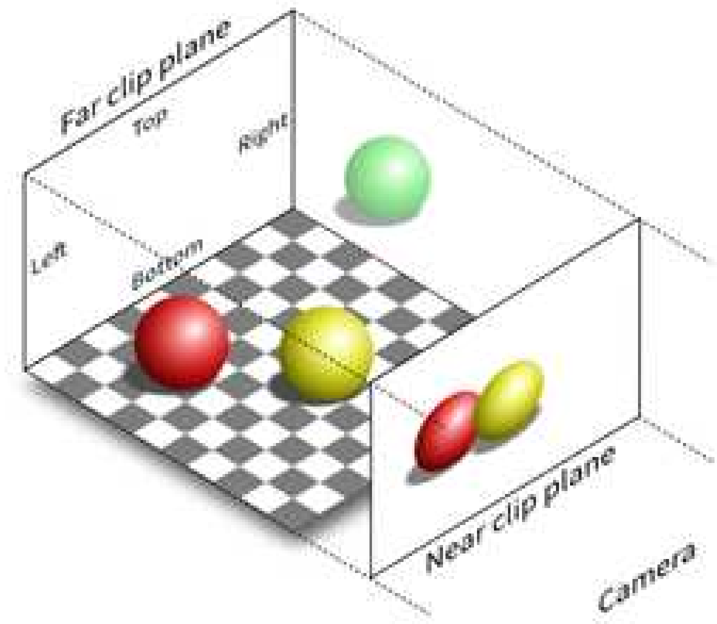
- **View frustum of perspective projection**
  - The region of space in the modeled world that may appear on the screen



- **View frustum culling**
  - The process of removing objects that lie completely outside the viewing frustum

3D projection

- **Orthographic projection**
  - Optical rays converge at a point with infinite distance to the projection plane



Orthographic projection (O)

Setting up 3D projection in OpenGL

- **Orthogonal projection**

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

glOrtho(left,right,bottom,top,zNear,zFar);

glMatrixMode(GL_MODELVIEW);
```

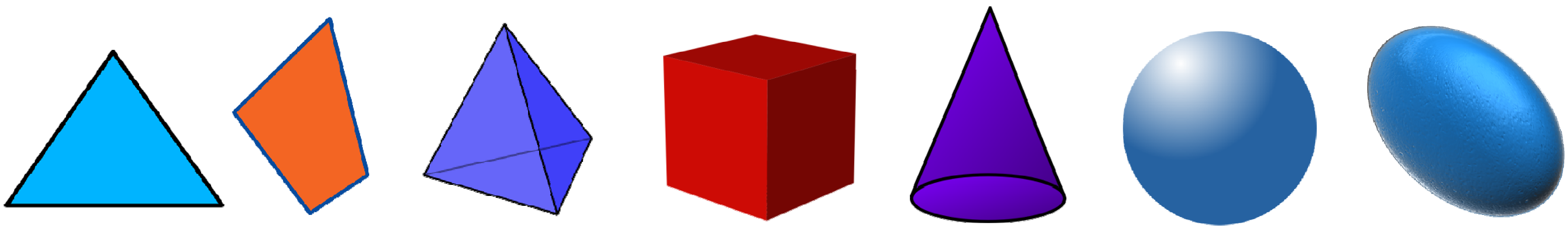
- **Perspective projection**

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluPerspective(fovy, aspect, zNear, zFar);

glMatrixMode(GL_MODELVIEW);
```

Step 2: Specify geometries

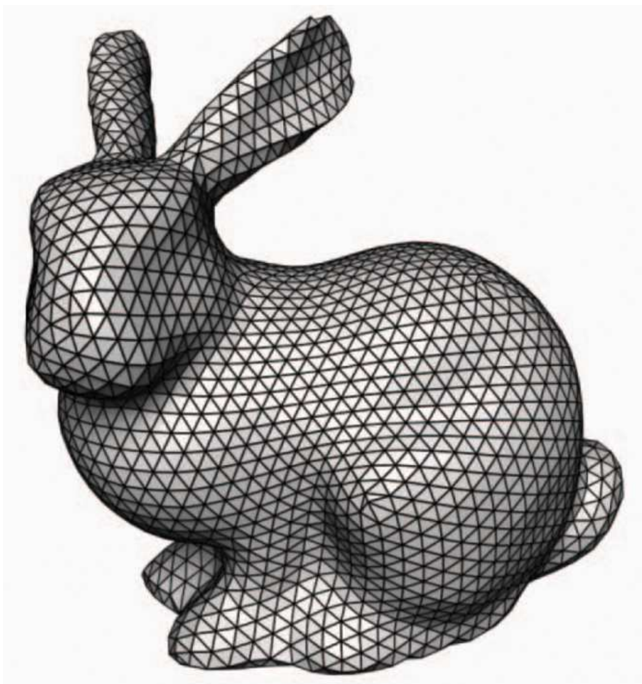
Geometry representation

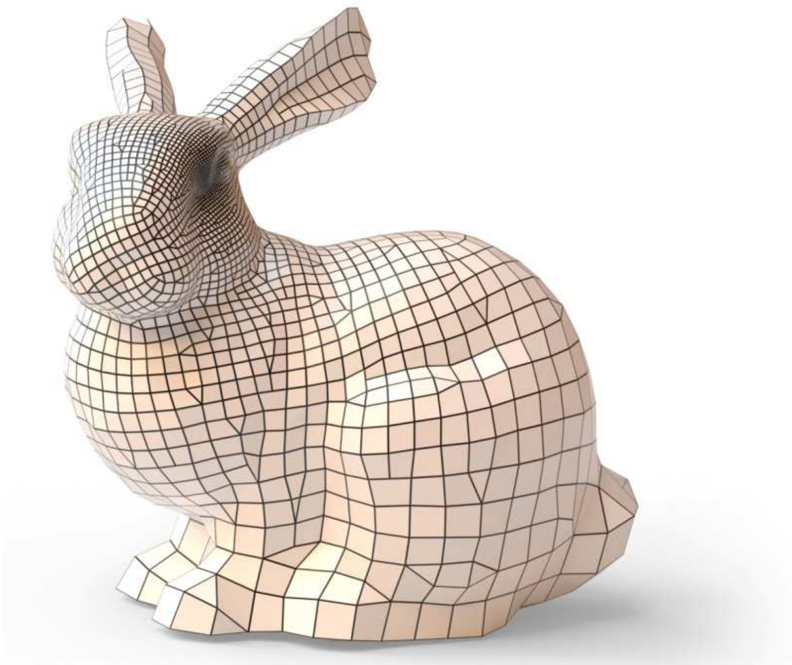
- **How can the shapes of objects be represented?**
  - Simple objects:
    - Triangle, quadrilateral, tetrahedron, cube, cone, sphere, ellipsoid, etc.



Geometry representation

- **Mesh**
  - Representation of shapes with a collection of geometrical primitives
    - Triangles
    - Quadrilaterals





Specify 2D primitives with OpenGL

- **Specify 2D triangles**

```
struct Position2D
{
    float x,y;
};

struct Triangle2D
{
   Position2D
         p1,p2,p3;
};

Triangle2D t1,t2,...;
```

```
glBegin(GL_TRIANGLES);

glVertex2f(t1.p1.x, t1.p1.y);
glVertex2f(t1.p2.x, t1.p2.y);
glVertex2f(t1.p3.x, t1.p3.y);

glVertex2f(t2.p1.x, t2.p1.y);
glVertex2f(t2.p2.x, t2.p2.y);
glVertex2f(t2.p3.x, t2.p3.y);

...

glEnd();
```

Specify 2D primitives with OpenGL

- **Specify 2D quadrilaterals**

```
struct Position2D
{
    float x,y;
};

struct Quad2D
{
   Position2D
        p1,p2,p3,p4;
};

Quad2D q1,q2,…;
```

```
glBegin(GL_QUADS);

glVertex2f(q1.p1.x, q1.p1.y);
glVertex2f(q1.p2.x, q1.p2.y);
glVertex2f(q1.p3.x, q1.p3.y);
glVertex2f(q1.p4.x, q1.p4.y);

glVertex2f(q2.p1.x, q2.p1.y);
glVertex2f(q2.p2.x, q2.p2.y);
glVertex2f(q2.p3.x, q2.p3.y);
glVertex2f(q2.p4.x, q2.p4.y);

…

glEnd();
```

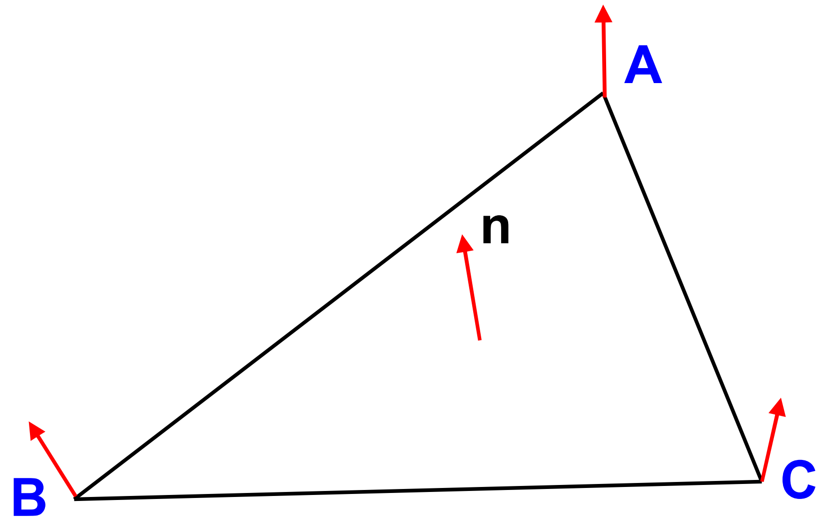
Specify 2D primitives with OpenGL

- **Specify 2D polygon**

```
glBegin(GL_POLYGON);

glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glVertex2f(x4, y4);
...
glVertex2f(xn, yn);

glEnd();
```

A 3D triangle

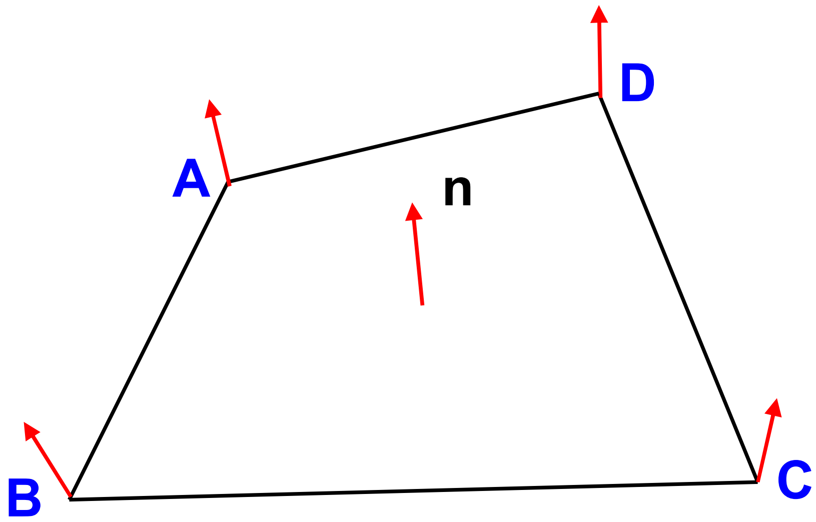
- **What constitute of a 3D triangle?**
  - Three vertices, three edges, one triangular face
  - One face normal, three vertex normals
  - Three vertex colors



n=AB x **AC**

Quadrilaterals

- **What constitute of a 3D quadrilateral?**
  - Four vertices, four edges, one face
    - Do not always ensure co-planar property
  - One face normal, four vertex normals
  - Four vertex colors



n=AC x **BD**

Specify 3D primitives with OpenGL

- **Specify 3D triangles**

```
struct Position3D
{
    float x,y,z;
};

struct Triangle3D
{
   Position3D
        p1,p2,p3;
};

Triangle2D t1,t2,…;
```

```
glBegin(GL_TRIANGLES);

glVertex3f(t1.p1.x, t1.p1.y, t1.p1.z);
glVertex3f(t1.p2.x, t1.p2.y, t1.p2.z);
glVertex3f(t1.p3.x, t1.p3.y, t1.p3.z);

glVertex3f(t2.p1.x, t2.p1.y, t2.p1.z);
glVertex3f(t2.p2.x, t2.p2.y, t2.p2.z);
glVertex3f(t2.p3.x, t2.p3.y, t2.p3.z);

…

glEnd();
```

Specify 3D primitives with OpenGL

- Specify **3D quadrilaterals**

```
struct Position3D
{
    float x,y,z;
};

struct Quad3D
{
   Position3D
         p1,p2,p3,p4;
};

Quad3D q1,q2,…;
```

```
glBegin(GL_QUADS);

glVertex3f(q1.p1.x, q1.p1.y, q1.p1.z);
glVertex3f(q1.p2.x, q1.p2.y, q1.p2.z);
glVertex3f(q1.p3.x, q1.p3.y, q1.p3.z);
glVertex3f(q1.p4.x, q1.p4.y, q1.p4.z);

glVertex3f(q2.p1.x, q2.p1.y, q2.p1.z);
glVertex3f(q2.p2.x, q2.p2.y, q2.p2.z);
glVertex3f(q2.p3.x, q2.p3.y, q2.p3.z);
glVertex3f(q2.p4.x, q2.p4.y, q2.p4.z);

…

glEnd();
```

Specify 3D primitives with OpenGL

- **Specify 3D triangles with face normal**

```
struct Position3D
{
    float x,y,z;
};

struct Triangle3D
{
   Position3D
         p1,p2,p3;
};

Triangle2D t1,t2,...;
```

```
glBegin(GL_TRIANGLES);

glNormal3f(n1.x,n1.y.n1.z);
glVertex3f(t1.p1.x, t1.p1.y, t1.p1.z);
glVertex3f(t1.p2.x, t1.p2.y, t1.p2.z);
glVertex3f(t1.p3.x, t1.p3.y, t1.p3.z);

glNormal3f(n2.x,n2.y.n2.z);
glVertex3f(t2.p1.x, t2.p1.y, t2.p1.z);
glVertex3f(t2.p2.x, t2.p2.y, t2.p2.z);
glVertex3f(t2.p3.x, t2.p3.y, t2.p3.z);

...

glEnd();
```

Specify 3D primitives with OpenGL

- **Specify 3D triangles with vertex normal**

```
glBegin(GL_TRIANGLES);

glNormal3f(t1.n1.x,t1.n1.y.t1.n1.z); glVertex3f(t1.p1.x, t1.p1.y, t1.p1.z);
glNormal3f(t1.n2.x,t1.n2.y.t1.n2.z); glVertex3f(t1.p2.x, t1.p2.y, t1.p2.z);
glNormal3f(t1.n3.x,t1.n3.y.t1.n3.z); glVertex3f(t1.p3.x, t1.p3.y, t1.p3.z);

glNormal3f(t2.n1.x,t2.n1.y.t2.n1.z); glVertex3f(t2.p1.x, t2.p1.y, t2.p1.z);
glNormal3f(t2.n2.x,t2.n2.y.t2.n2.z); glVertex3f(t2.p2.x, t2.p2.y, t2.p2.z);
glNormal3f(t2.n3.x,t2.n3.y.t2.n3.z); glVertex3f(t2.p3.x, t2.p3.y, t2.p3.z);

...

glEnd();
```

Vertex properties

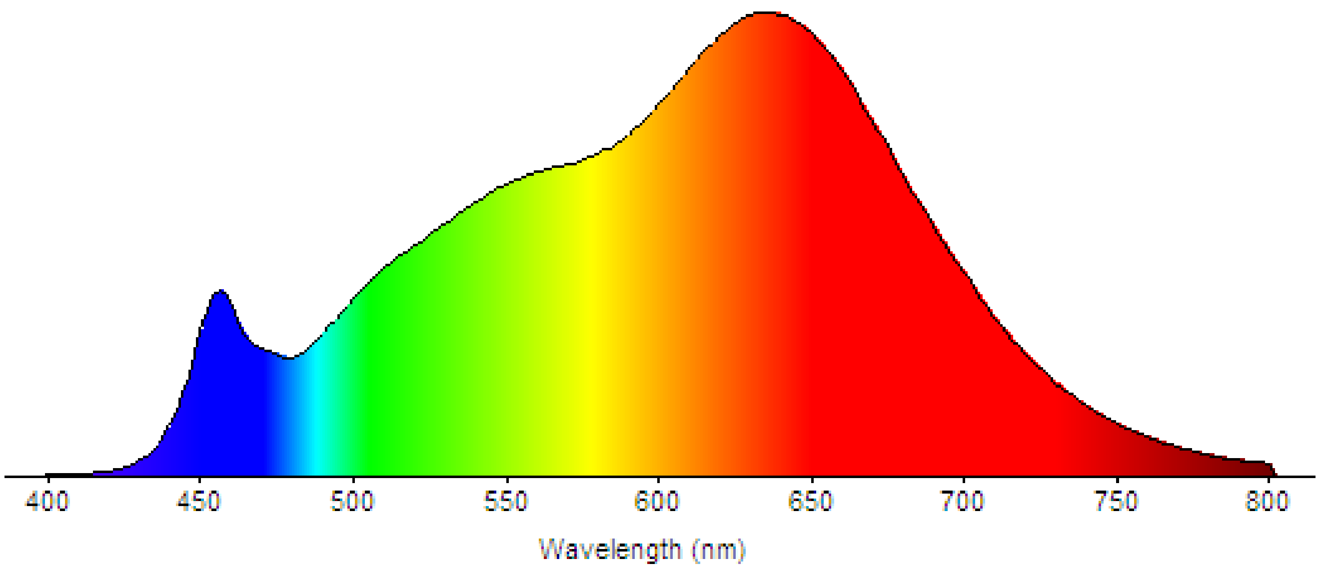
- **Each vertex contains the coordinate of its location**
  - (x,y) for 2D and (x,y,z) for 3D

- **It can also contain some vertex properties**
  - Vertex color (r,g,b)
  - Vertex normal (used for lighting) (nx,ny,nz)
  - Texture coordinate (for texture mapping)
  - Other user specified quantities

Step 3: Specify color

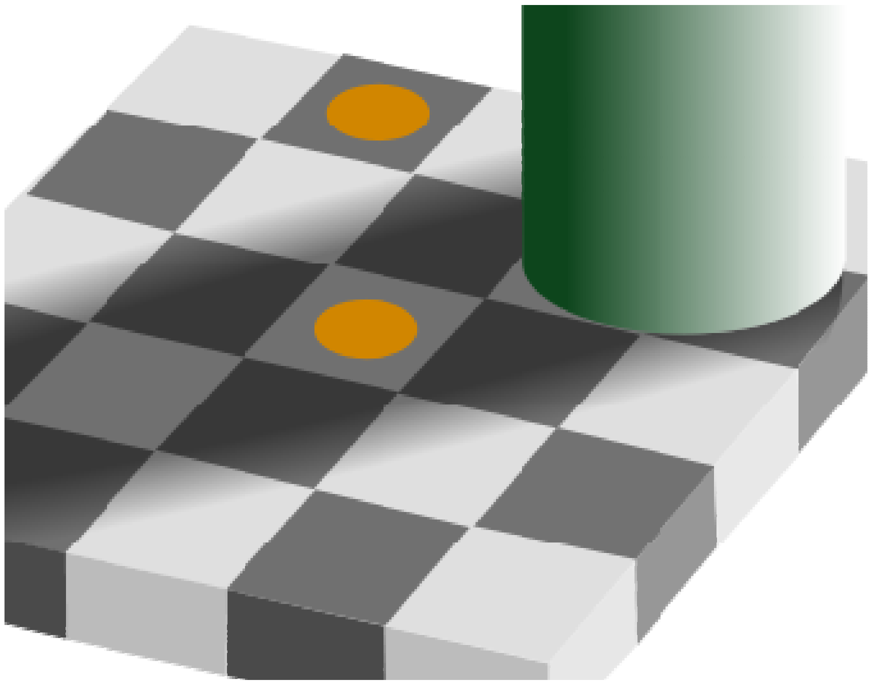
Color representation

- **Spectral power distribution of light**
  - A distribution function of wavelength
  - Describe the amount of light (power) at each (continuous) wavelength



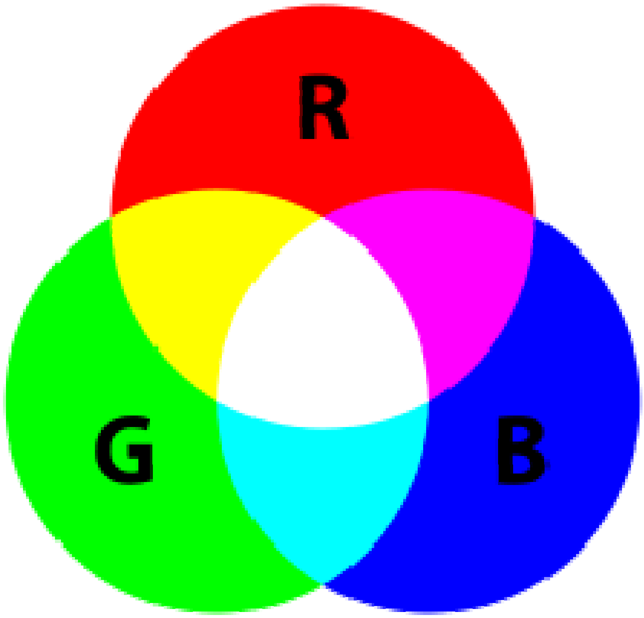
Color representation

- **What is color?**
  - Visual perception property for human eyes (subjective)
  - Is human perception of color consistent to physical representation (objective)?
    - Sometimes NOT!



Color representation

- **RGB color**
  - An additive color model
  - Red, green and blue light are added together in various ways to reproduce a broad array of colors
  - Set R,G,B intensities respectively
  - Unnormalized v.s. normalized
    - Unnormalized: [0,255] – 1 byte
    - Normalized: [0,1] – 1 float



Specify vertex color in OpenGL

- **A color can be associated with each vertex**
  - Specify color for points

```
glBegin(GL_POINTS);

glColor3f(r1,g1,b1); glVertex2f(x1,y1);
glColor3f(r2,g2,b2); glVertex2f(x2,y2);
...
glColor3f(rn,gn,bn); glVertex2f(xn,yn);

glEnd();
```

Specify vertex color in OpenGL

- **A color can be associated with each vertex**
  - Specify color for triangles

```
struct Position2D
{
    float x,y; //position
    float r,g,b; //color
};

struct Triangle2D
{
   Position2D
        p1,p2,p3;
};

Triangle2D t1,t2,...;
```

Specify vertex color in OpenGL

- **A color can be associated with each vertex**
  - Specify color for triangles

```
glBegin(GL_TRIANGLES);

glColor3f(t1.p1.r, t1.p1.g, t1.p1.b); glVertex2f(t1.p1.x, t1.p1.y);
glColor3f(t1.p2.r, t1.p2.g, t1.p2.b); glVertex2f(t1.p2.x, t1.p2.y);
glColor3f(t1.p3.r, t1.p3.g, t1.p3.b); glVertex2f(t1.p3.x, t1.p3.y);

glColor3f(t2.p1.r, t2.p1.g, t2.p1.b); glVertex2f(t2.p1.x, t2.p1.y);
glColor3f(t2.p2.r, t2.p2.g, t2.p2.b); glVertex2f(t2.p2.x, t2.p2.y);
glColor3f(t2.p3.r, t2.p3.g, t2.p3.b); glVertex2f(t2.p3.x, t2.p3.y);

...

glEnd();
```

Specify vertex color in OpenGL

- **A color can be associated with each vertex**
  - Specify color for quadrilaterals

```
struct Position2D
{
    float x,y; //position
    float r,g,b; //color
};

struct Quad2D
{
   Quad2D
        p1,p2,p3,p4;
};

Quad2D t1,t2,...;
```

Specify vertex color in OpenGL

- **A color can be associated with each vertex**
  - Specify color for quadrilaterals

```
glBegin(GL_QUADS);

glColor3f(q2.p1.r, q2.p1.g, q2.p1.b); glVertex2f(q2.p1.x, q2.p1.y);
glColor3f(q2.p2.r, q2.p2.g, q2.p2.b); glVertex2f(q2.p2.x, q2.p2.y);
glColor3f(q2.p3.r, q2.p3.g, q2.p3.b); glVertex2f(q2.p3.x, q2.p3.y);
glColor3f(q2.p4.r, q2.p4.g, q2.p4.b); glVertex2f(q2.p4.x, q2.p4.y);

glColor3f(q2.p1.r, q2.p1.g, q2.p1.b); glVertex2f(q2.p1.x, q2.p1.y);
glColor3f(q2.p2.r, q2.p2.g, q2.p2.b); glVertex2f(q2.p2.x, q2.p2.y);
glColor3f(q2.p3.r, q2.p3.g, q2.p3.b); glVertex2f(q2.p3.x, q2.p3.y);
glColor3f(q2.p4.r, q2.p4.g, q2.p4.b); glVertex2f(q2.p4.x, q2.p4.y);

...

glEnd();
```

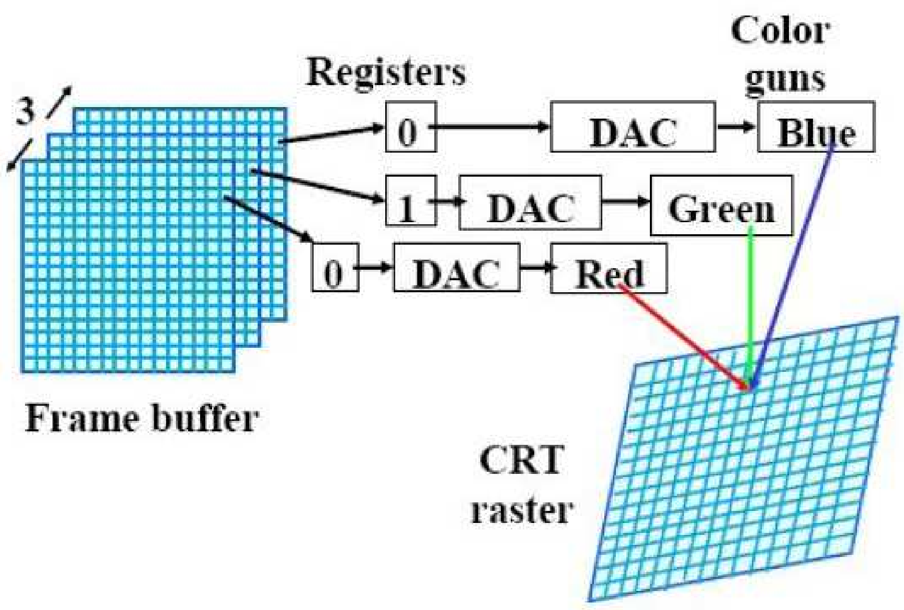
Step 4: Automatic Rasterization

Framebuffer

- **A portion of RAM containing a bitmap that drives a video display**
  - A memory buffer containing a complete frame of data
- **Screen buffer (video buffer)**
  - A part of computer memory used by a computer application
  - For the representation of the content to be shown on the computer display

Framebuffer

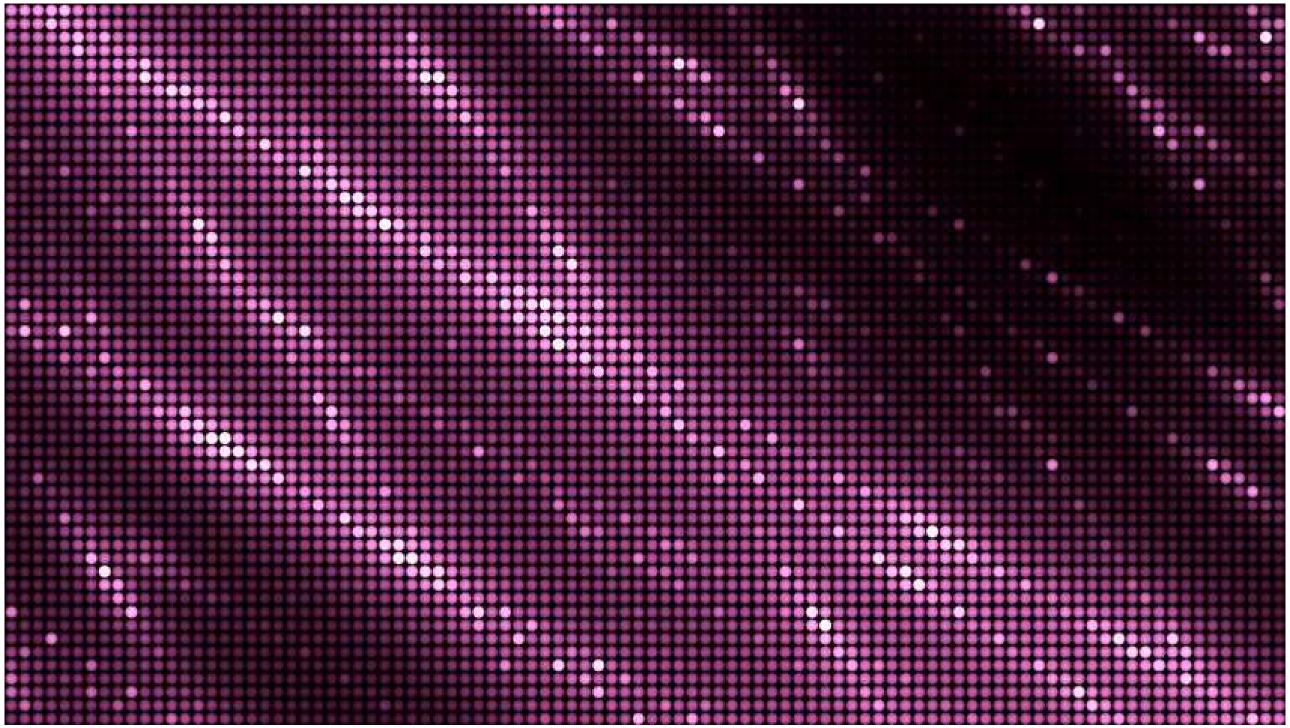
- **A framebuffer storing**
  - 2D array (discretized) containing color, depth, etc.
  - Can have multiple copies
  - Double buffer is commonly adopted



Simple Color buffer

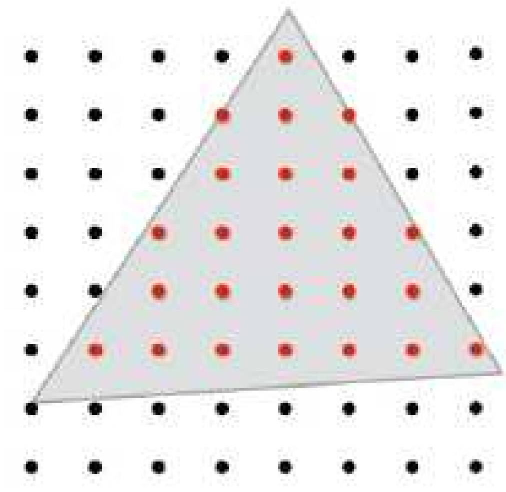
Digital screen

- **A digitized display device**
  - Pixels are physically displayed
  - In terms of 2D arrays, corresponding to an image

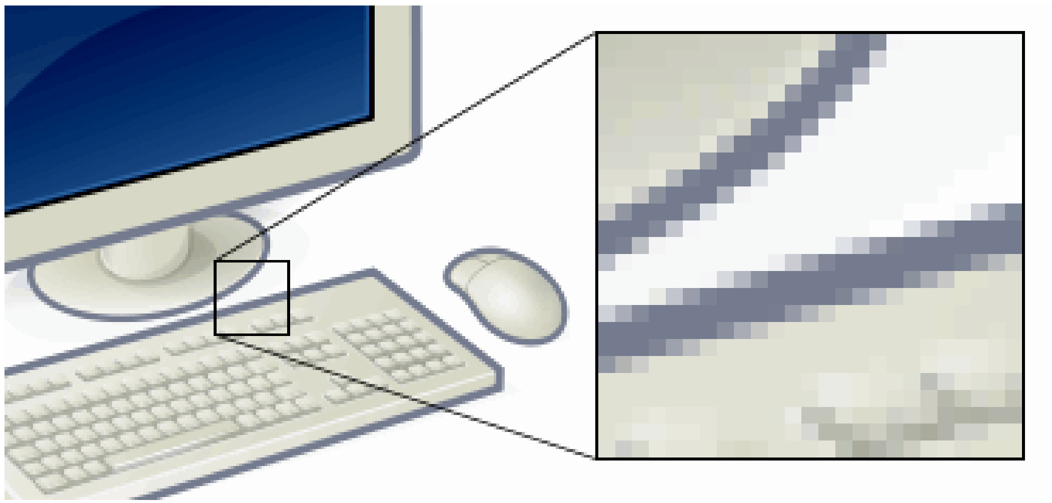


Digital screen

- **Rasterization**
  - The process of converting continuous signals to pixels
  - Filling pixel colors (and other values) in framebuffer



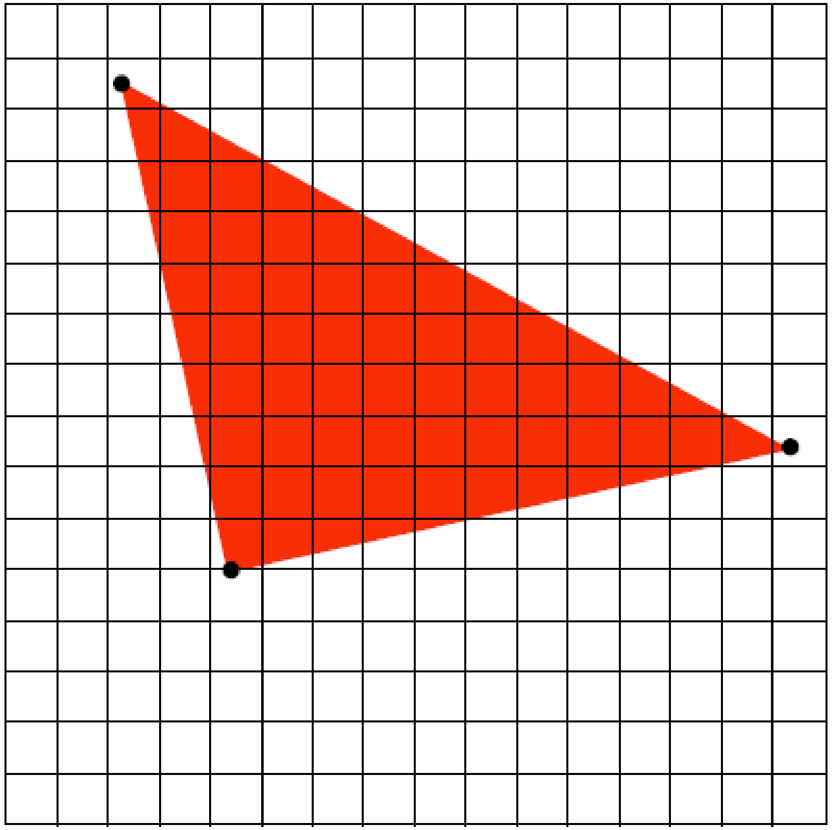
Sample triangle coverage



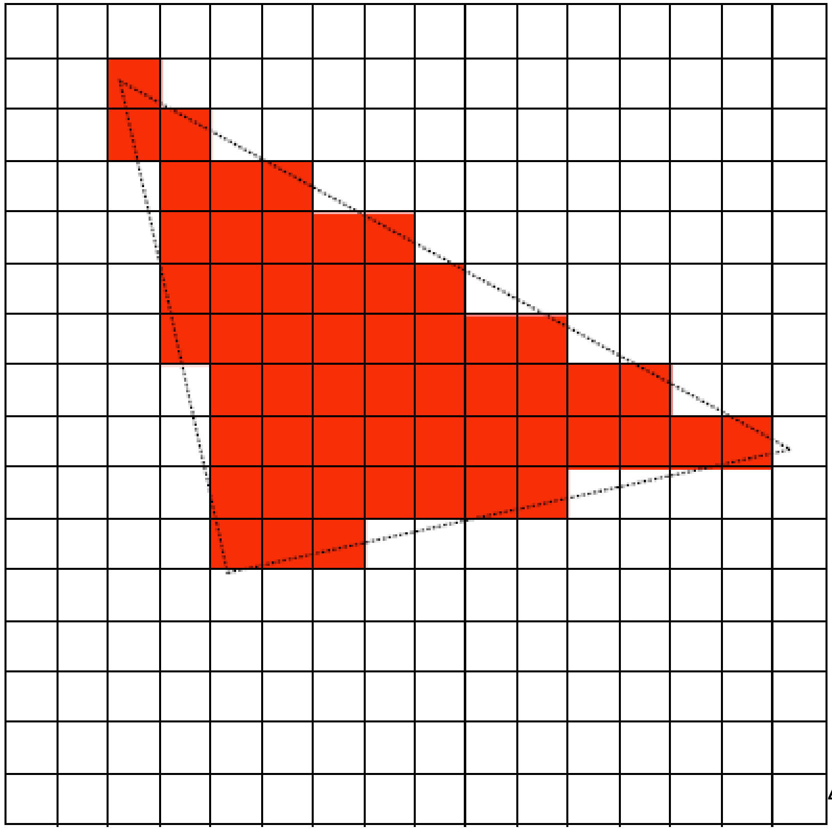
Rasterizing a triangle

- **What pixels do the triangle overlap?**

Input:
projected position of triangle vertices: P_0, P_1, P_2

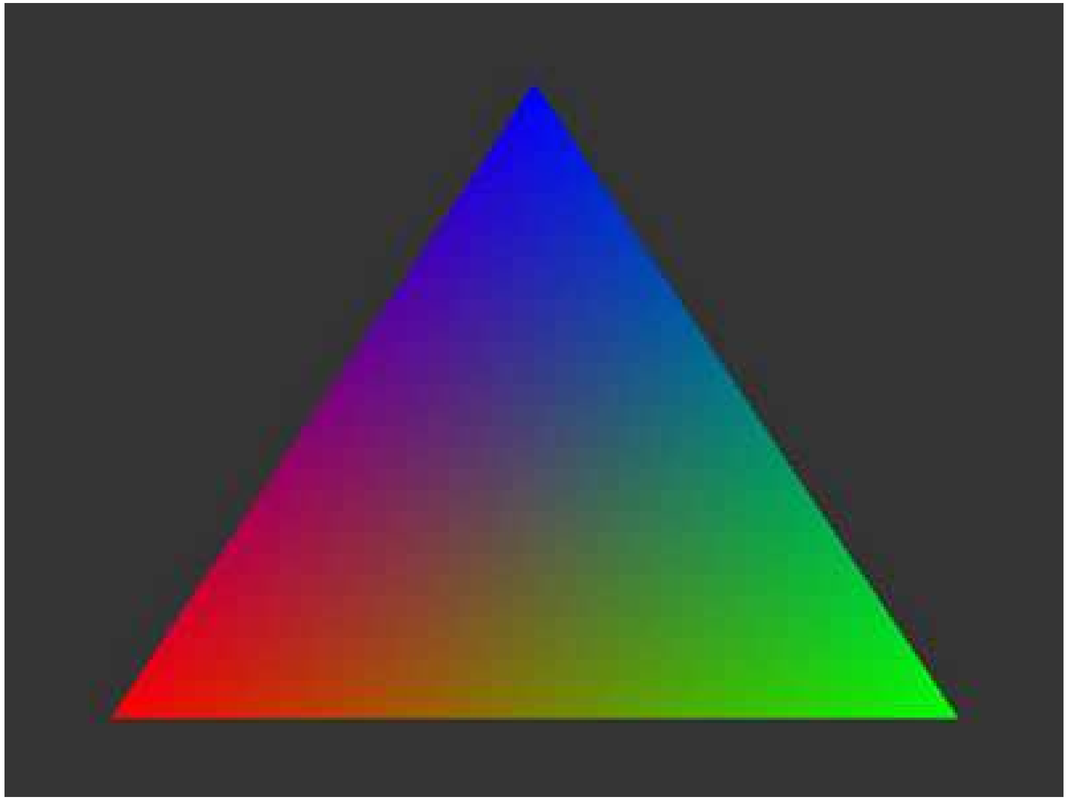


Output:
set of pixels "covered" by the triangle



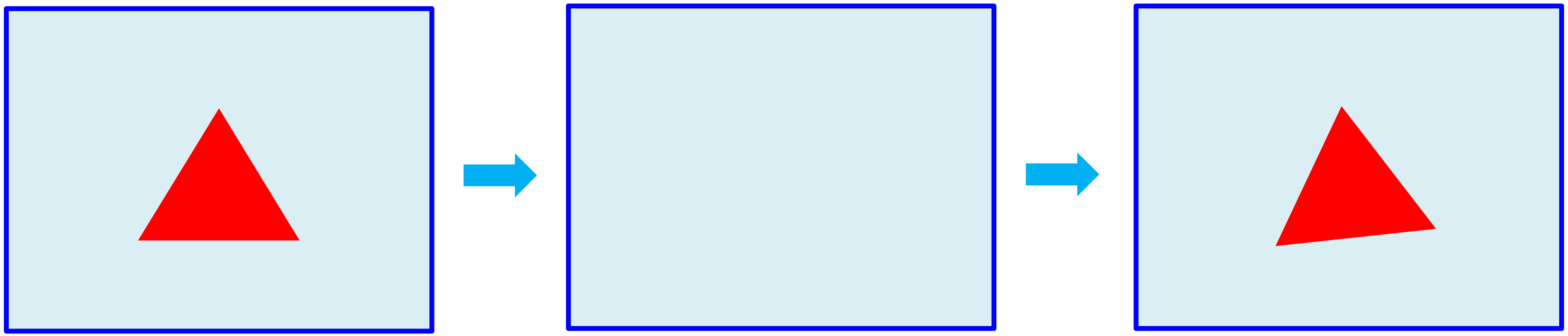
Filling pixel colors

- **Interpolate pixel colors in inner regions**
  - Linear interpolation based on vertex colors automatically



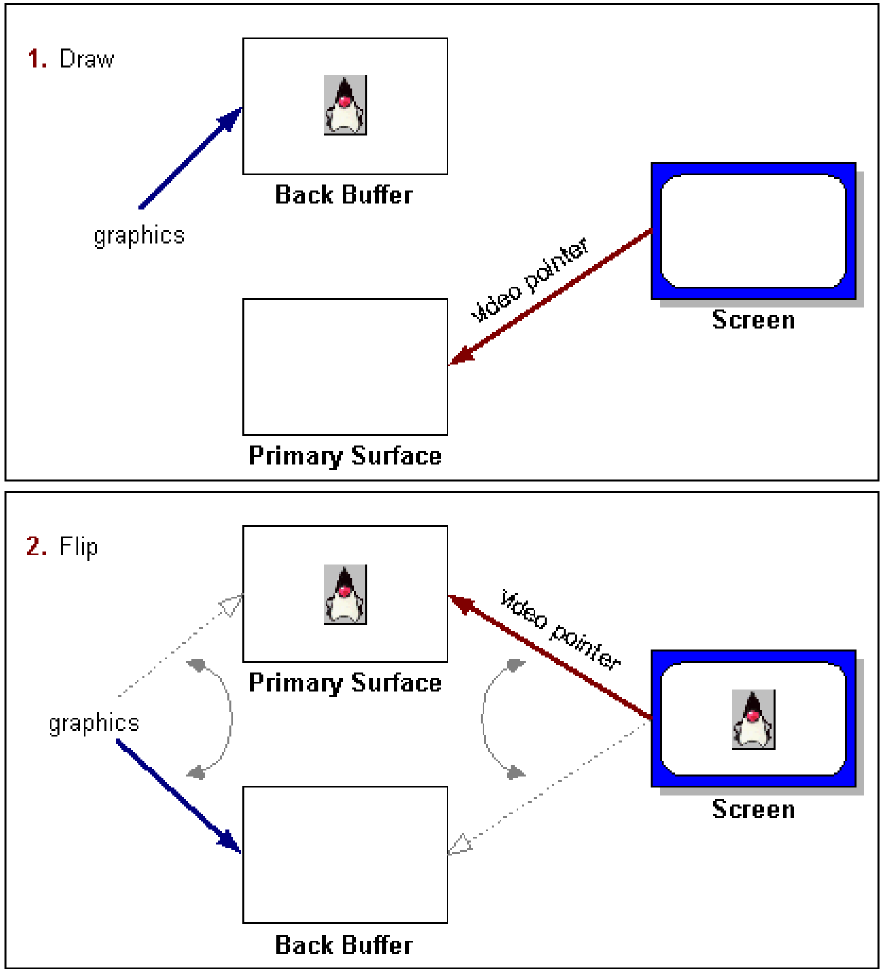
Filling pixel colors

- **Dynamic display**
  - The screen will be erased before displaying the next content
  - The sequential process results in flickering



Double buffering

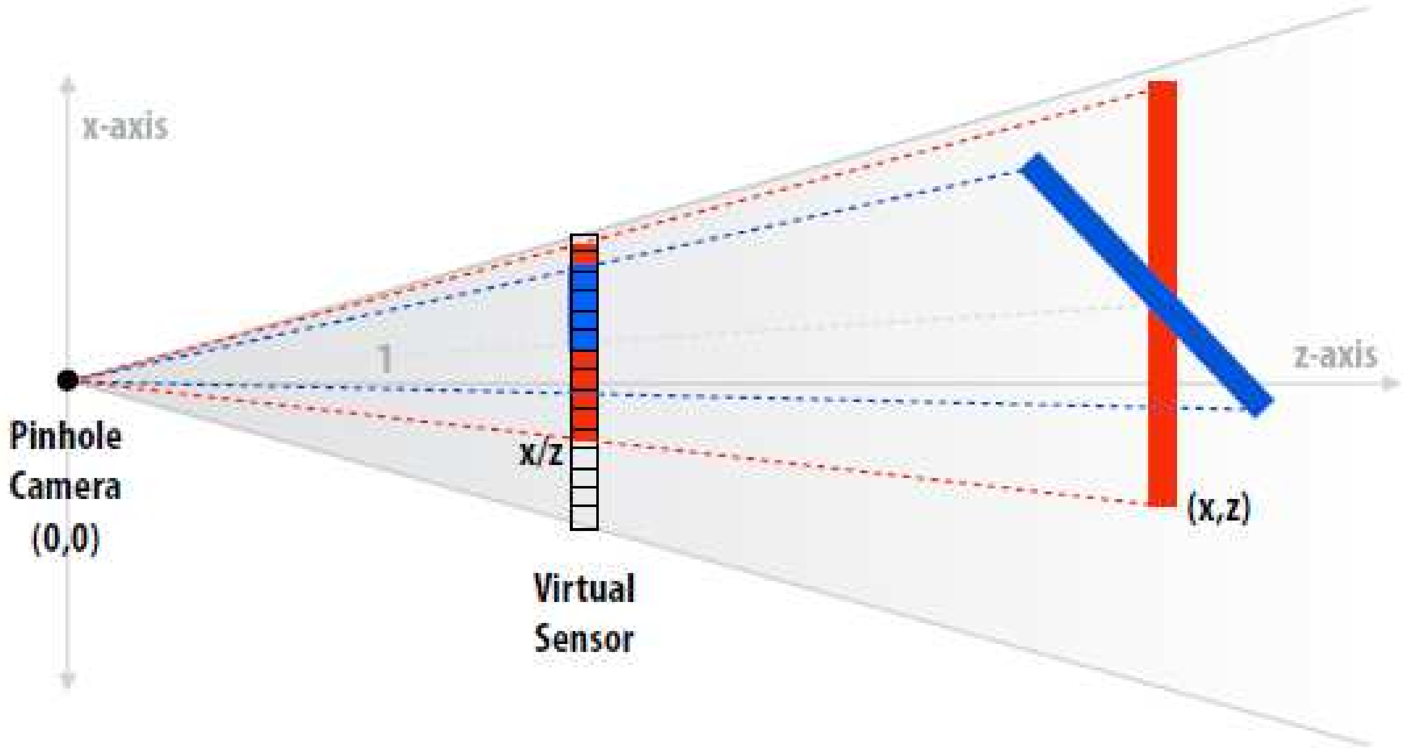
- **The way to eliminate flickering**



Step 4: Depth-Test

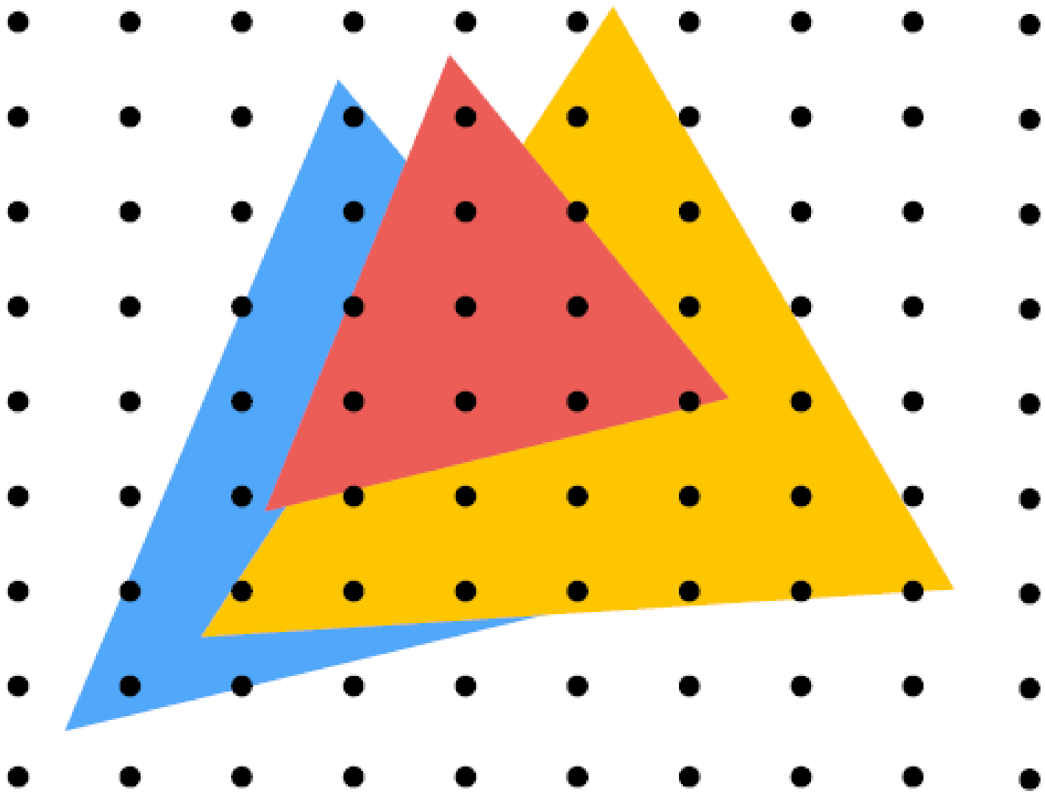
Concept of depth

- **The distance of a 3D point to the imaging plane**



Visibility

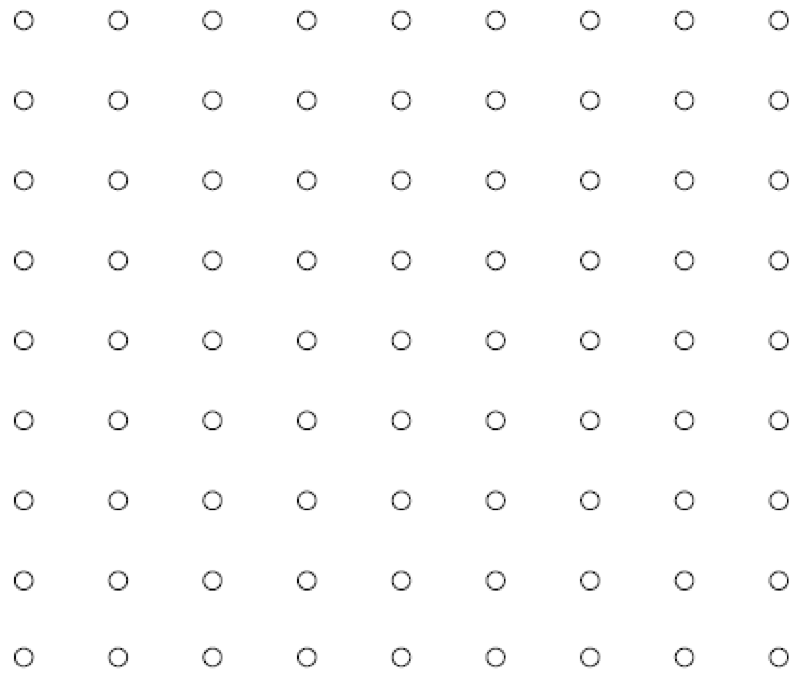
- **Visible surface determination**
  - The process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint



Depth buffer (Z-buffer)

- **For each coverage sample point**
  - Depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer

Initial state of depth buffer before rendering any triangles (all samples store farthest distance)



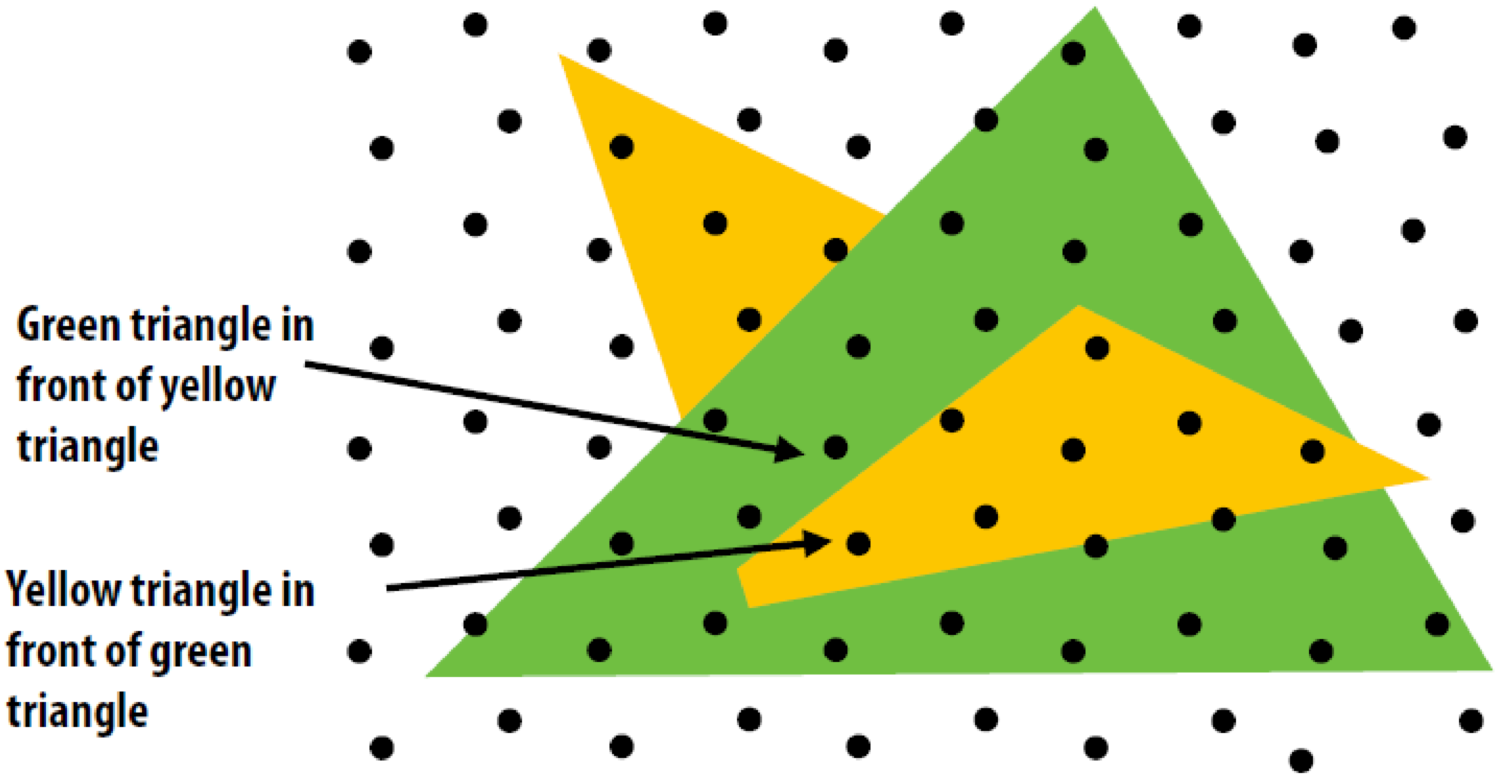
Grayscale value of sample point used to indicate distance

Black = small distance

White = large distance

Depth buffer (Z-buffer)

- **Does depth-buffer algorithm handle interpenetrating surfaces?**
  - Yes, of course!

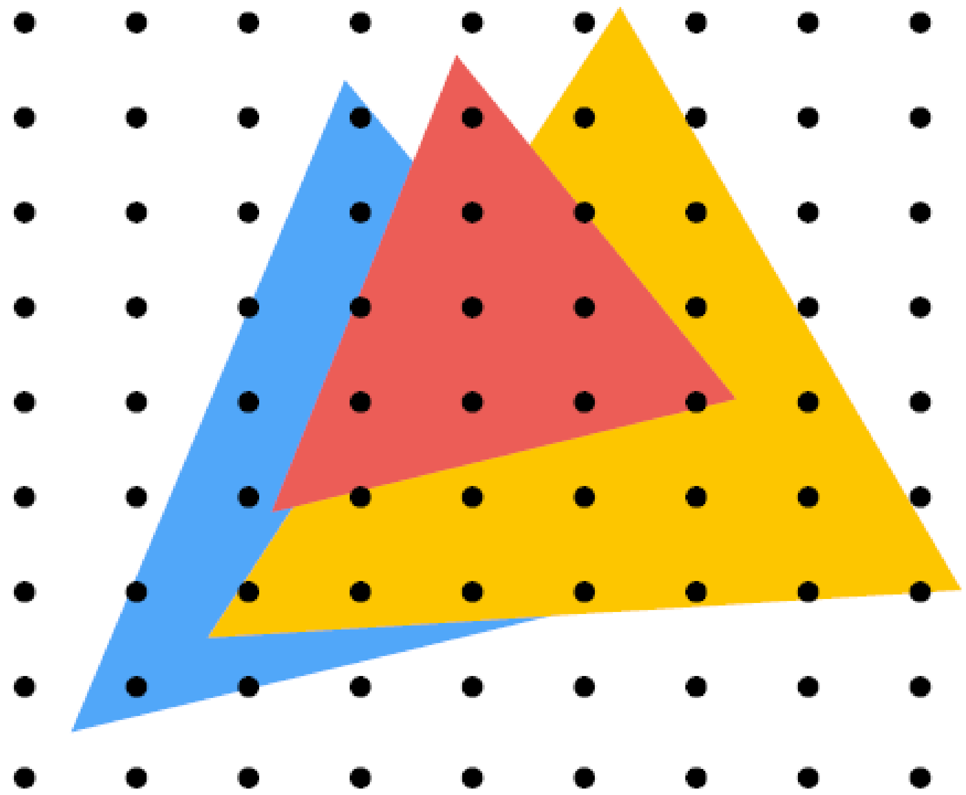


Depth buffer (Z-buffer)

- **An example of depth buffer**

Depth-test in pixel-fill

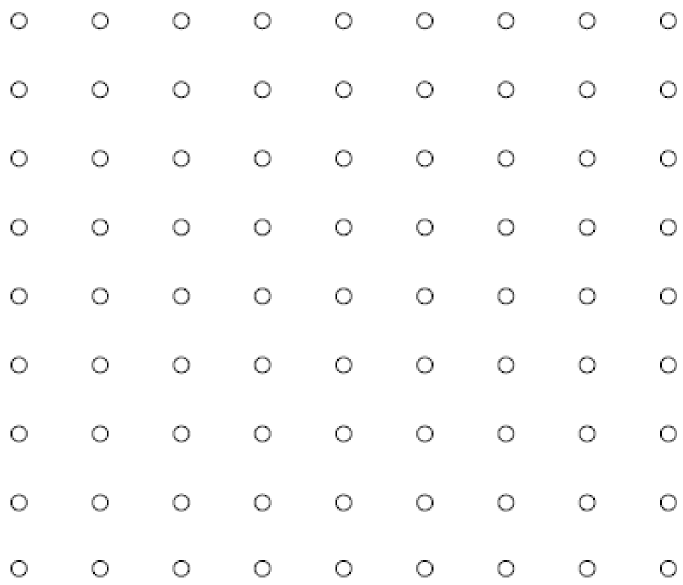
- **Example: rendering three opaque triangles**



Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5



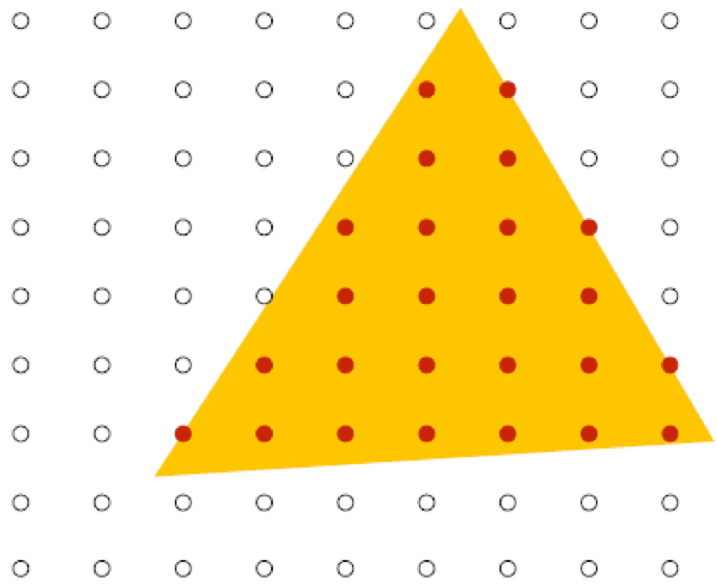
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test



Depth buffer contents

Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

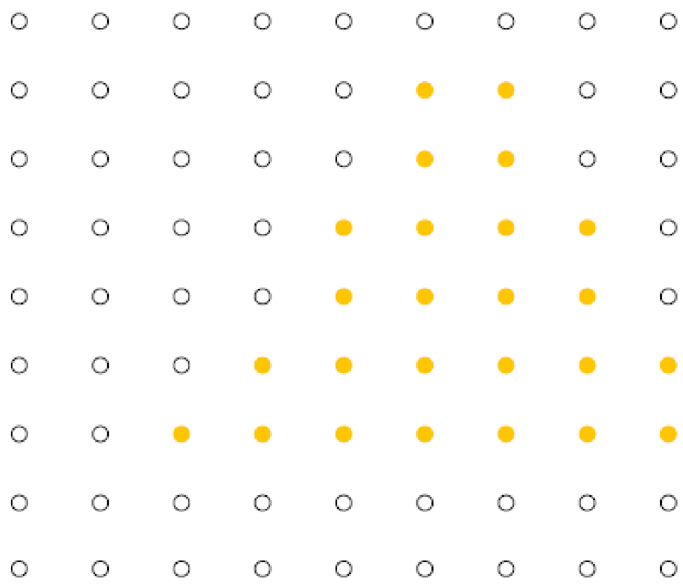
After processing yellow triangle:

Grayscale value of sample point used to indicate distance

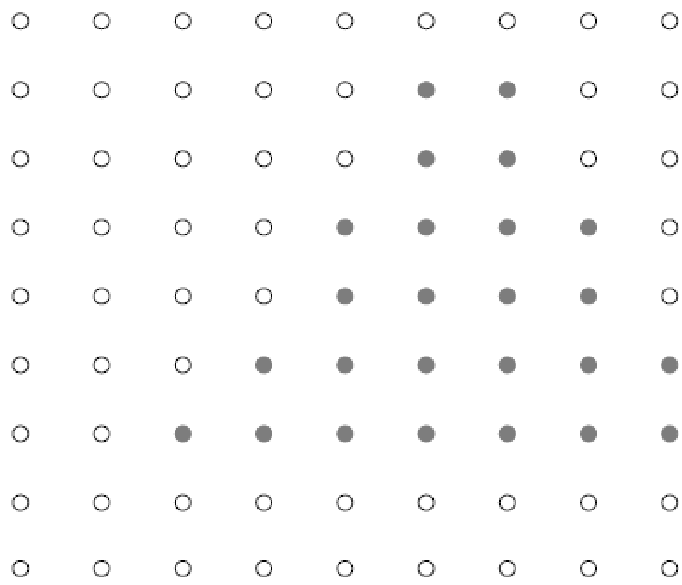
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

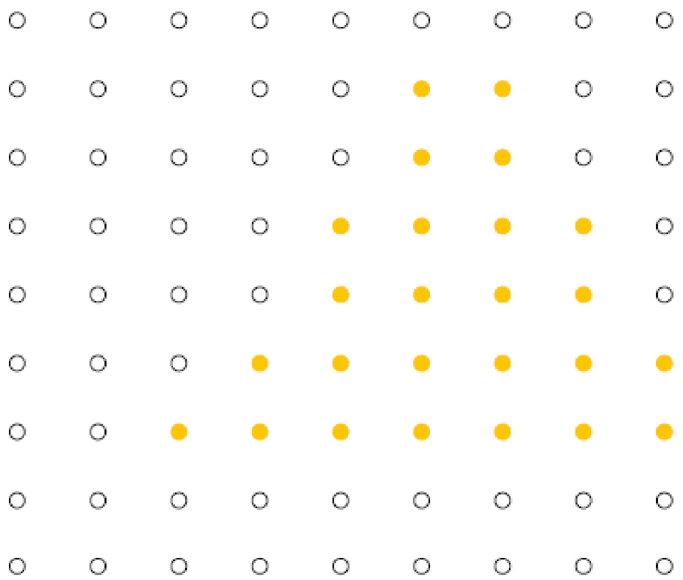
depth = 0.75

Grayscale value of sample point used to indicate distance

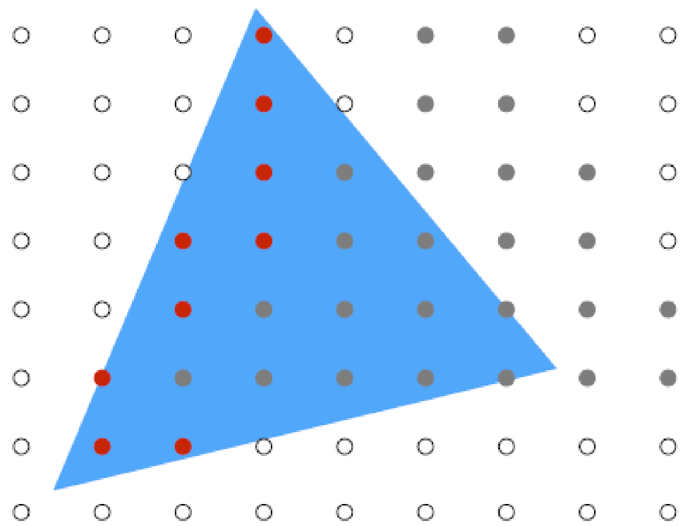
White = large distance

Black = small distance

Red = sample passed depth test



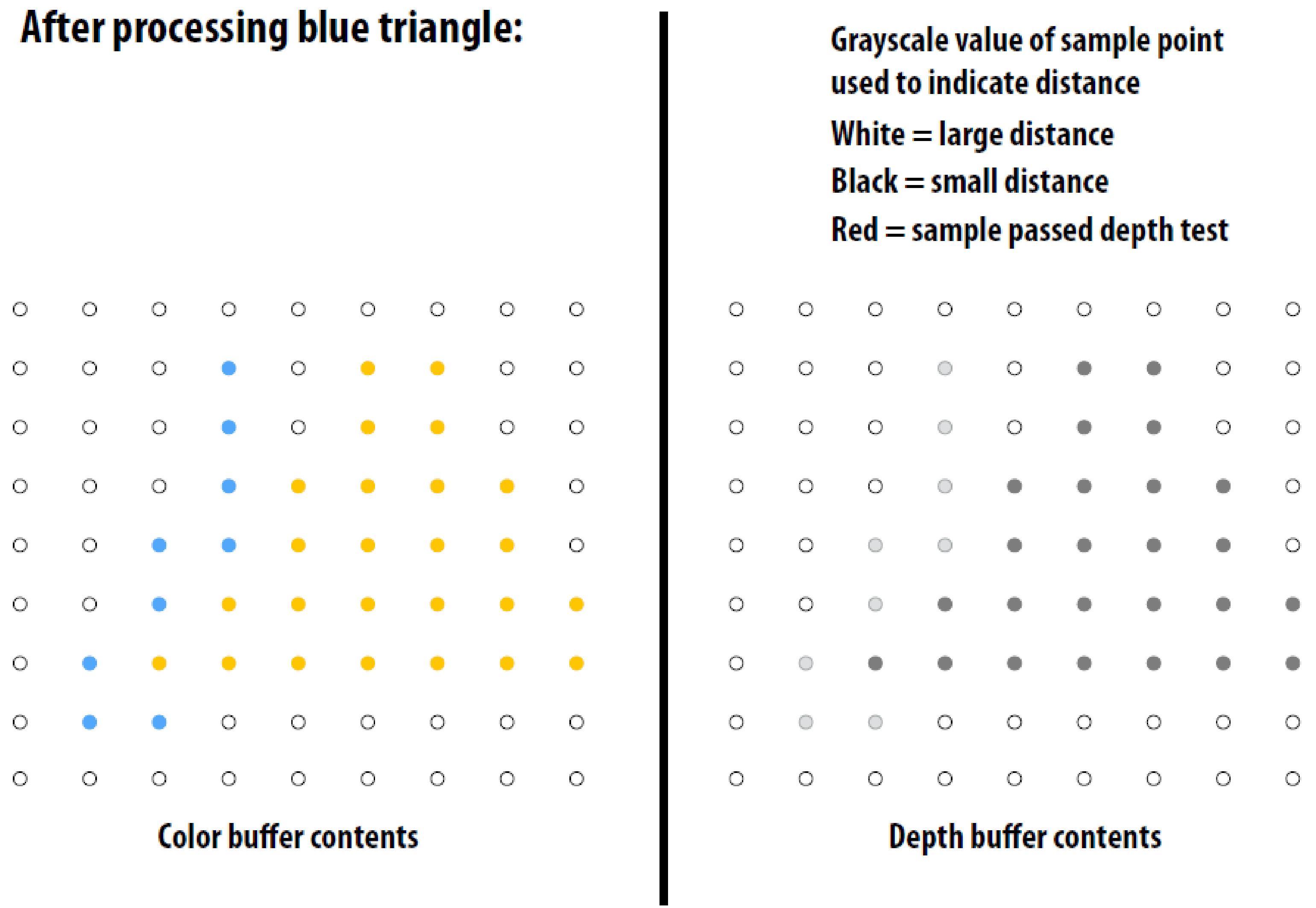
Color buffer contents



Depth buffer contents

Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

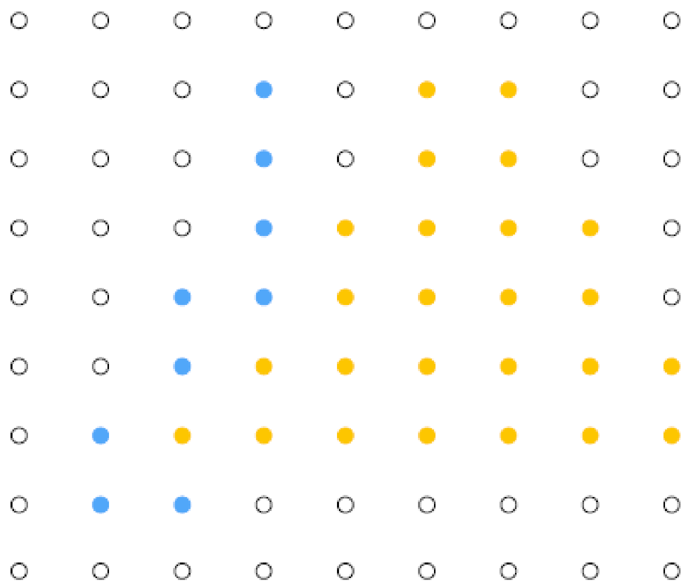


Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



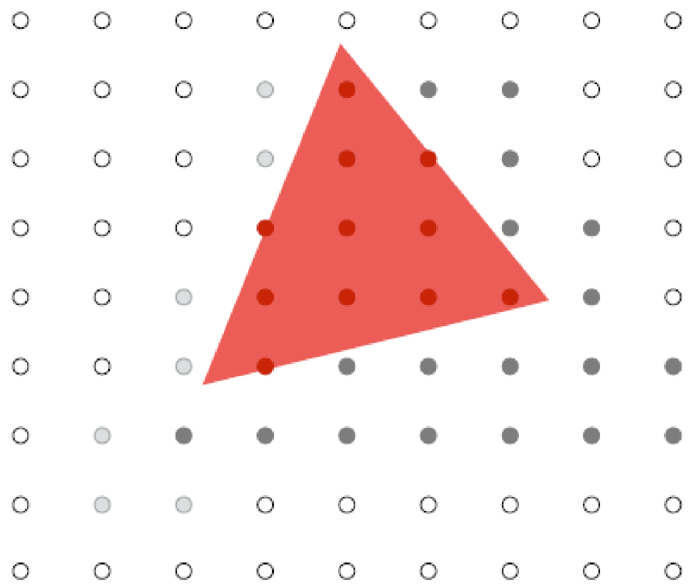
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

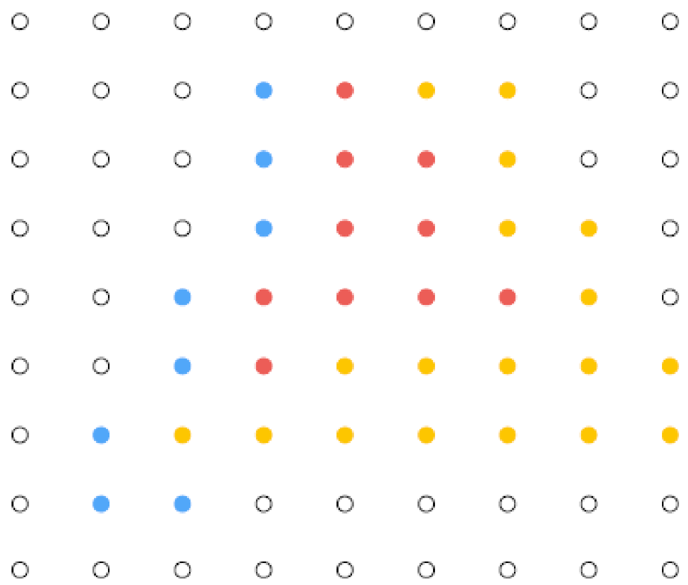


Depth buffer contents

Depth-test in pixel-fill

- Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



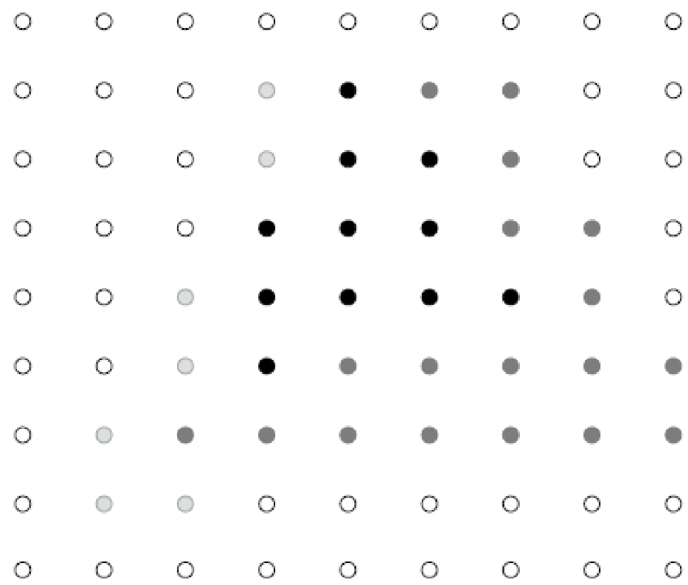
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test



Depth buffer contents

Enable depth-test in OpenGL

- **How to enable and disable depth-test in OpenGL?**

```
/* Make the window's context current */
glfwMakeContextCurrent(window);

glEnable(GL_DEPTH_TEST); //glDisable(GL_DEPTH_TEST);

/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Render here */
    glClear(GL_COLOR_BUFFER_BIT);
    /* Swap front and back buffers */

    //add your OpenGL rendering calls here

    glfwSwapBuffers(window);
    …
}
```

The first OpenGL program

The first OpenGL program

- **Using GLFW library** (http://www.glfw.org/)
  - Initialization and window creation, double buffer by default

```
GLFWwindow*  window;

/* Initialize the library */
if (!glfwInit()) return -1;

/* Create a windowed mode window and its OpenGL context */
 window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
if (!window) { glfwTerminate(); return -1; }

/* Make the window's context current */
glfwMakeContextCurrent(window);
```

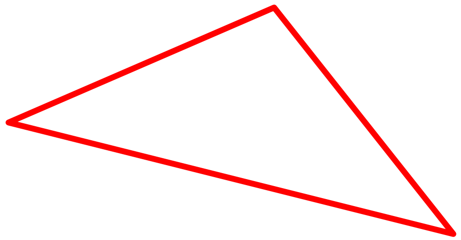
The first OpenGL program

- **Using GLFW library**
  - Render things

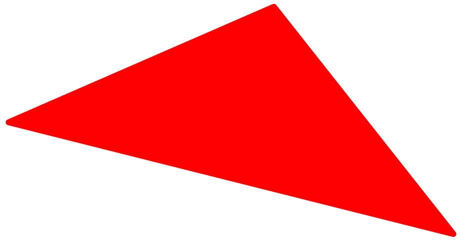
```
/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Clear color buffer */
    glClear(GL_COLOR_BUFFER_BIT);

    //add your OpenGL rendering calls here

    /* Swap front and back buffers */
    glfwSwapBuffers(window);
    /* Poll for and process events */
    glfwPollEvents();
}
```

Wired v.s. filled polygons

- **Wired triangle**
  - Only the edges are drawn



- **Filled triangle**
  - Not only the edges are drawn, but also the inner region is filled with color



- **Enable/disable polygon filling in OpenGL**

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINES);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

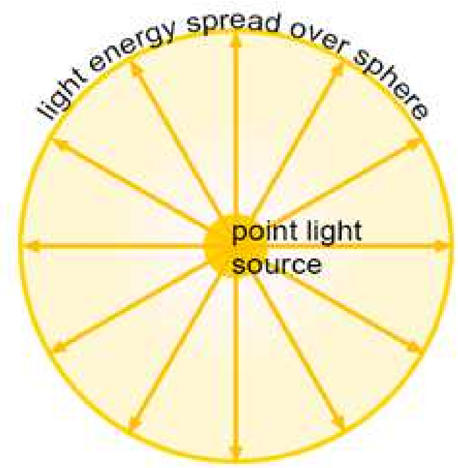
Step 5: Shading

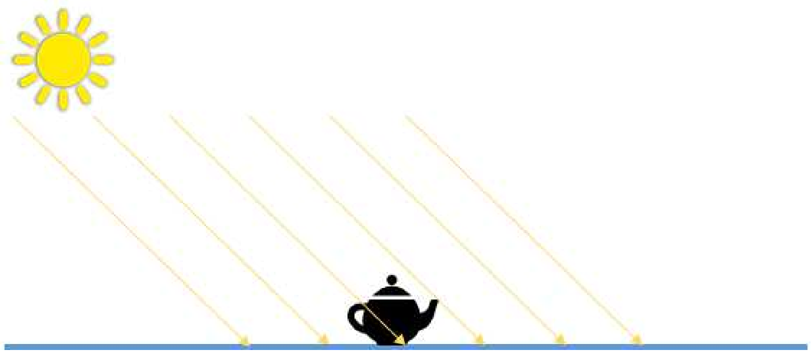
Shading

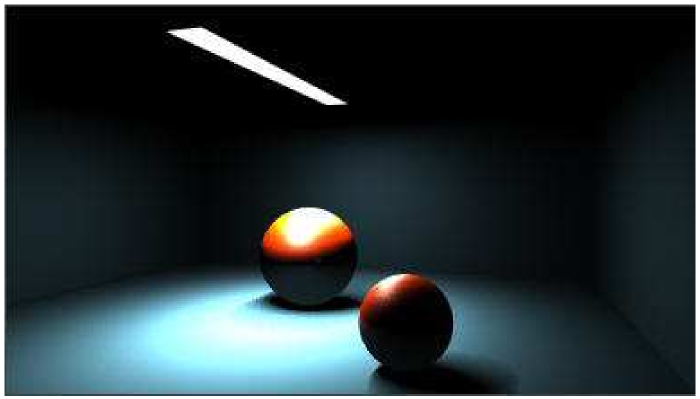
- **What is shading?**
  - Shading refers to the process of altering the color of an object/surface/polygon in the 3D scene
  - Based on the angle to lights and the distance from lights or appearance model to create a photorealistic effect

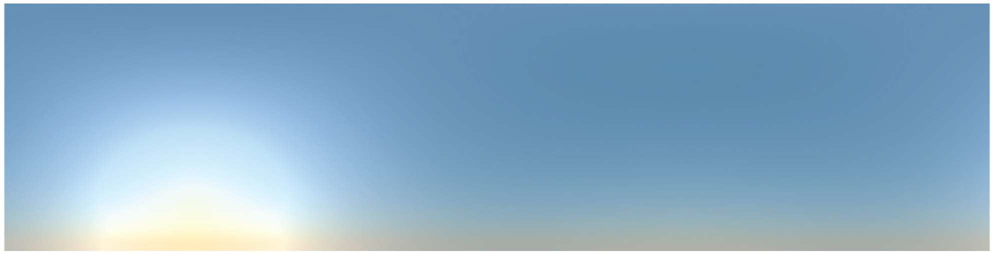
Light sources

- **The sources to illuminate light**
  - Point/parallel/spot light sources
  - Area light sources
  - Environmental light sources





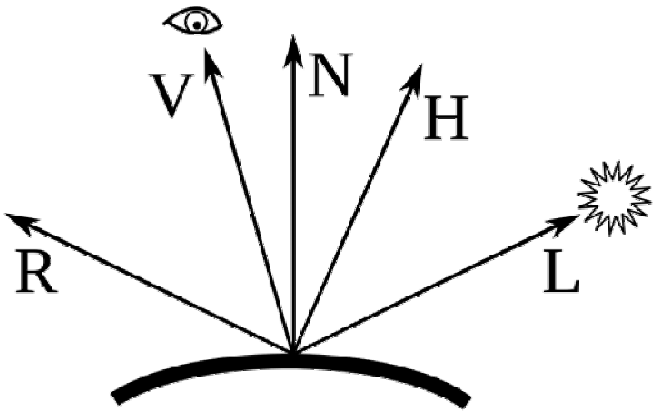


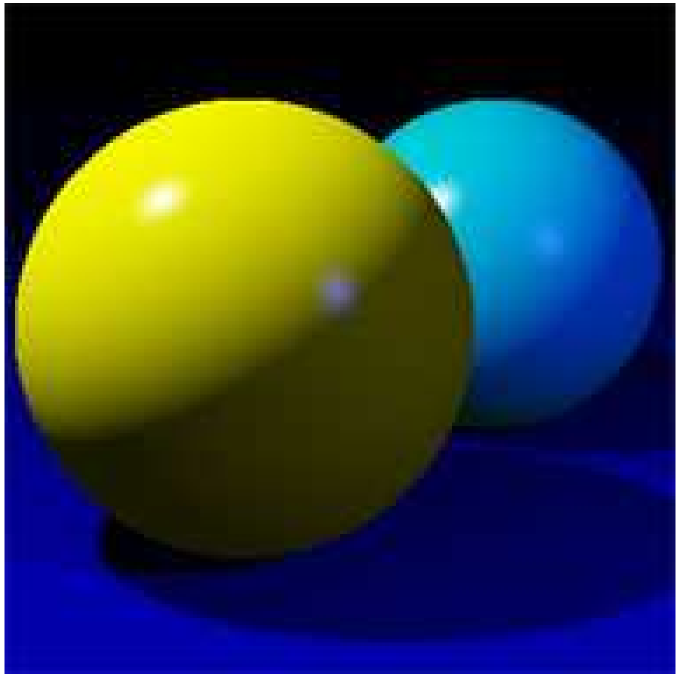


Environment light map

Lighting model

- **Determine how light is reflected**
  - Lambert diffuse reflection model
  - Phong reflection model



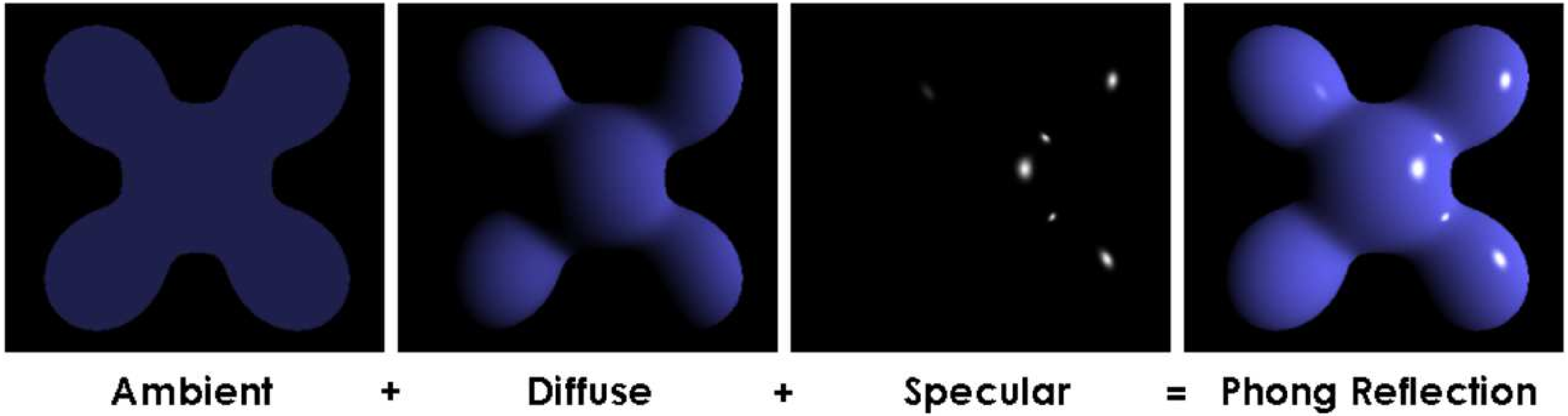


Lighting model

- **Phong reflection model**
  - Light source decomposition
    - Ambient light: constant environment lighting (view independent)
    - Diffuse light: light that is scattered uniformly to each direction (view independent)
    - Specular light: light that is scattered along specific directions (view dependent)
  - Material reflection decomposition
    - Ambient reflection: component that reflects only ambient light
    - Diffuse reflection: component that reflects only diffuse light
    - Specular reflection: component that reflects only specular light

Lighting model

- **Phong reflection model**



$$\hat{R}_m = 2(\hat{L}_m \cdot \hat{N})\hat{N} - \hat{L}_m$$

$$I_\text{p} = k_\text{a} i_\text{a} + \sum_{m \;\in\; \text{lights}} (k_\text{d} (\hat{L}_m \cdot \hat{N}) i_{m,\text{d}} + k_\text{s} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\text{s}})$$

Lighting in OpenGL

- **Enable lighting and set lighting components**

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Create light components
GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Lighting in OpenGL

- **Set material reflection components**

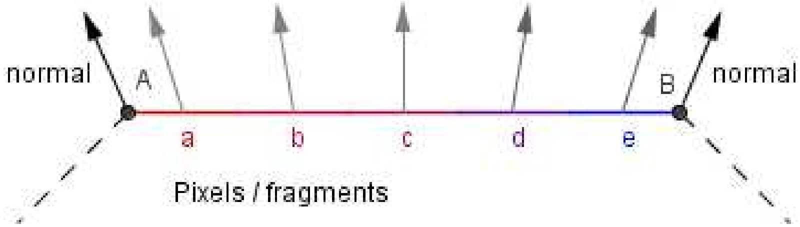
```
GLfloat ambient[] = { 1.0f, 1.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);

GLfloat diffuse[] = { 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);

GLfloat specular[] = { 1.0f, 1.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
```

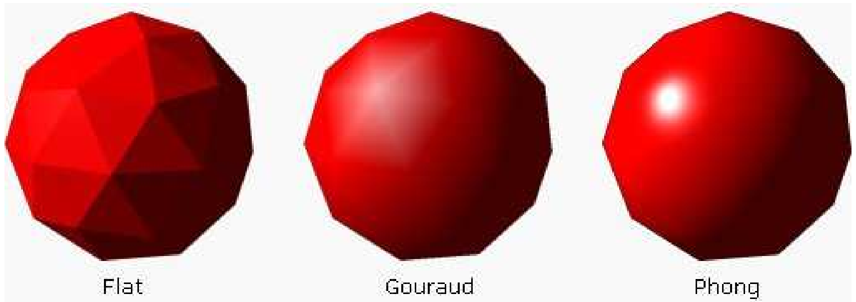
- **Enable smooth shading**

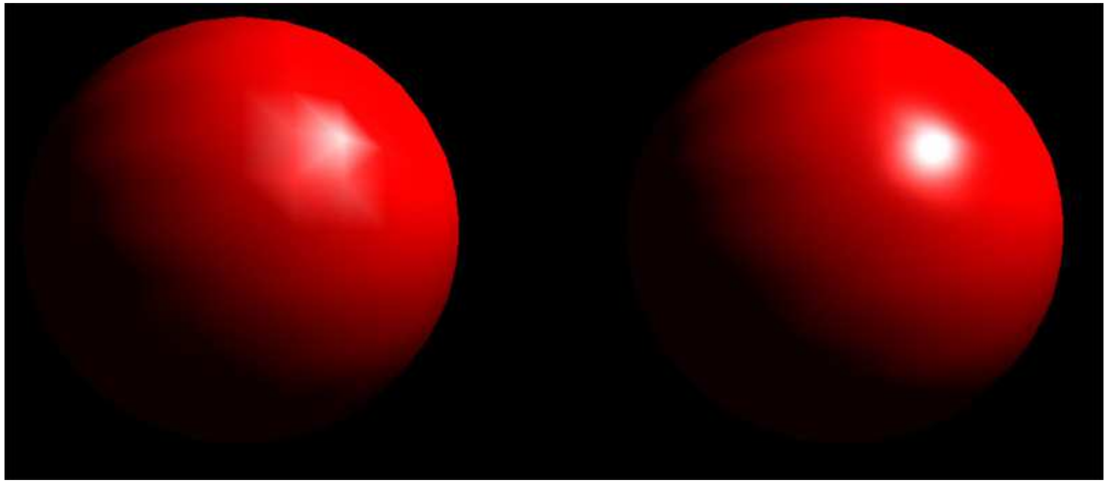
```
glShadeModel(GL_SMOOTH);        //glShadeModel(GL_FLAT);
```

Shading model

- **Gourand v.s. Phong shading**
  - Interpolation by color or by normal

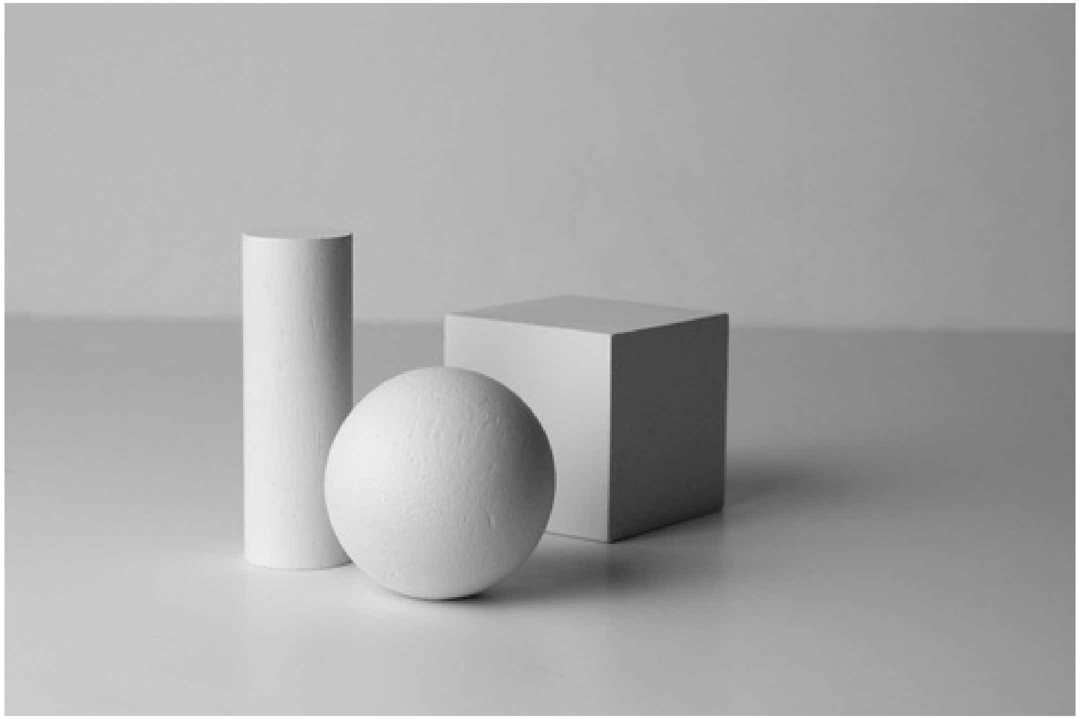






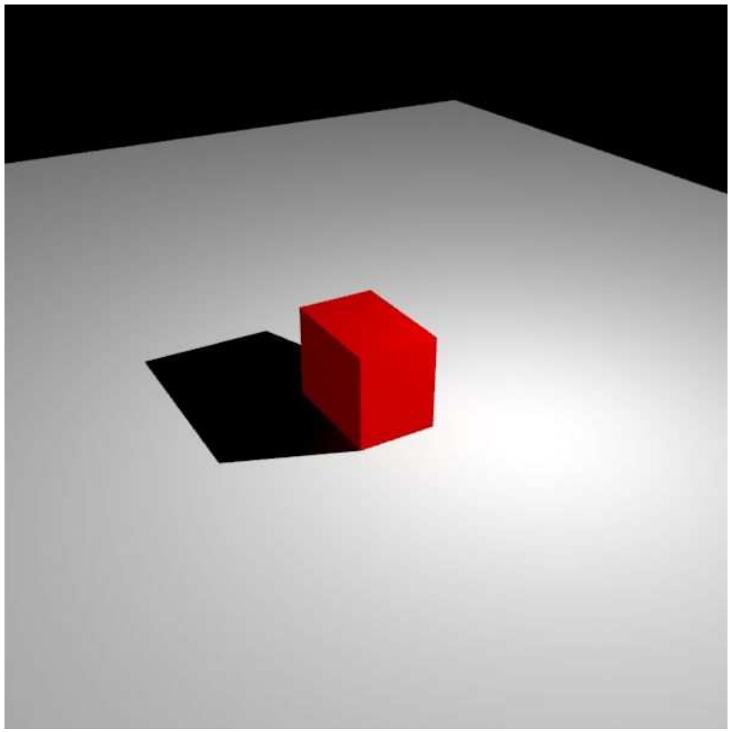
Shadow

- **What is a shadow**
  - A shadow is a dark area where light from a light source is blocked by an opaque object



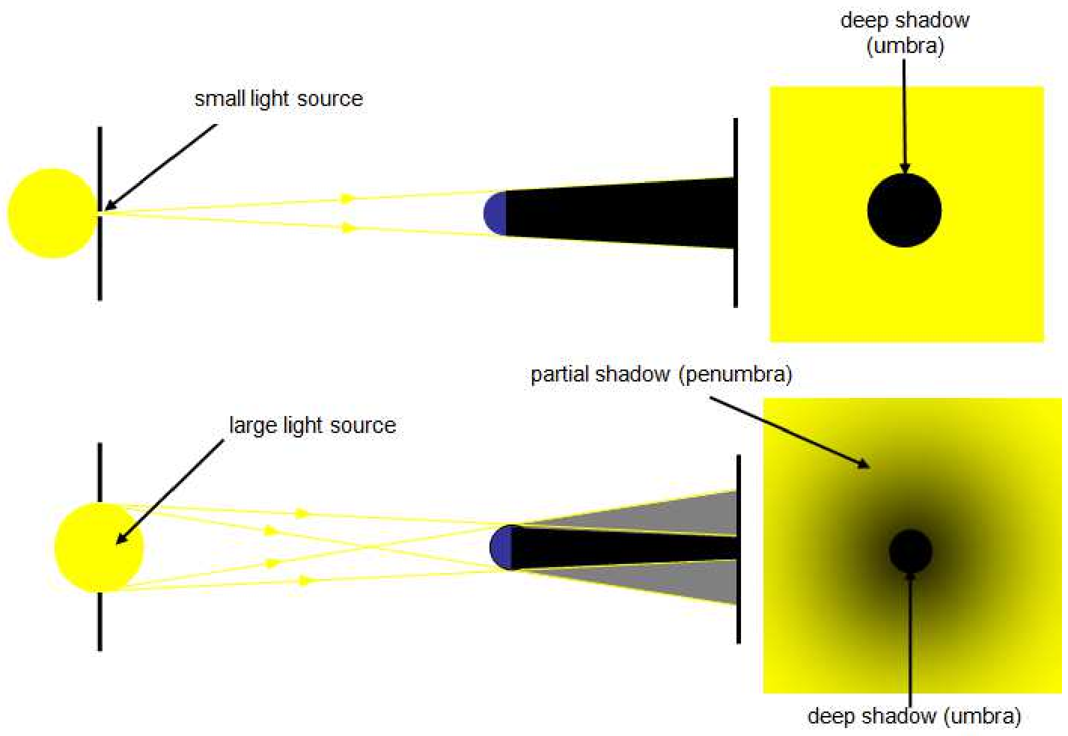
Shadow

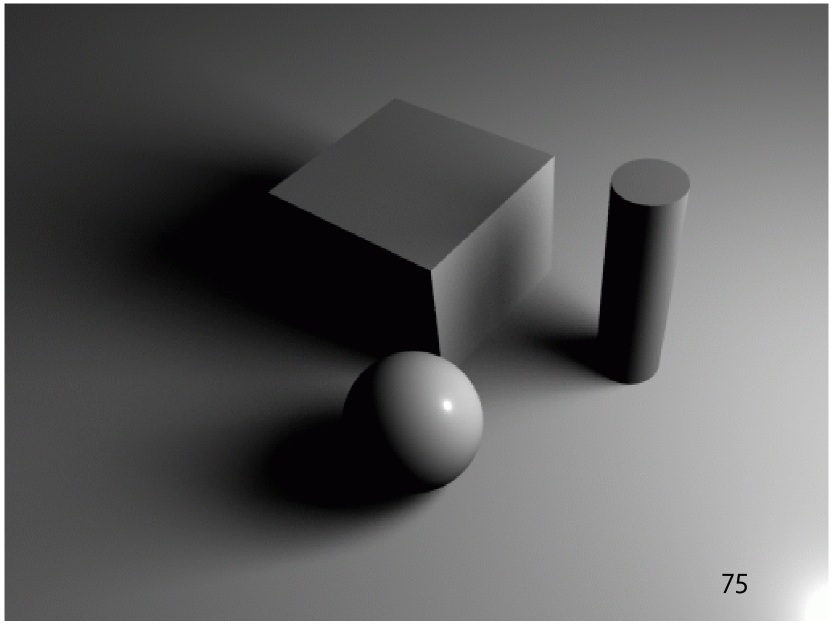
- **Hard shadow**
  - Shadow with sharp boundaries
  - Usually generated from a point/parallel light source
  - Shadow mapping (OpenGL)



Shadow

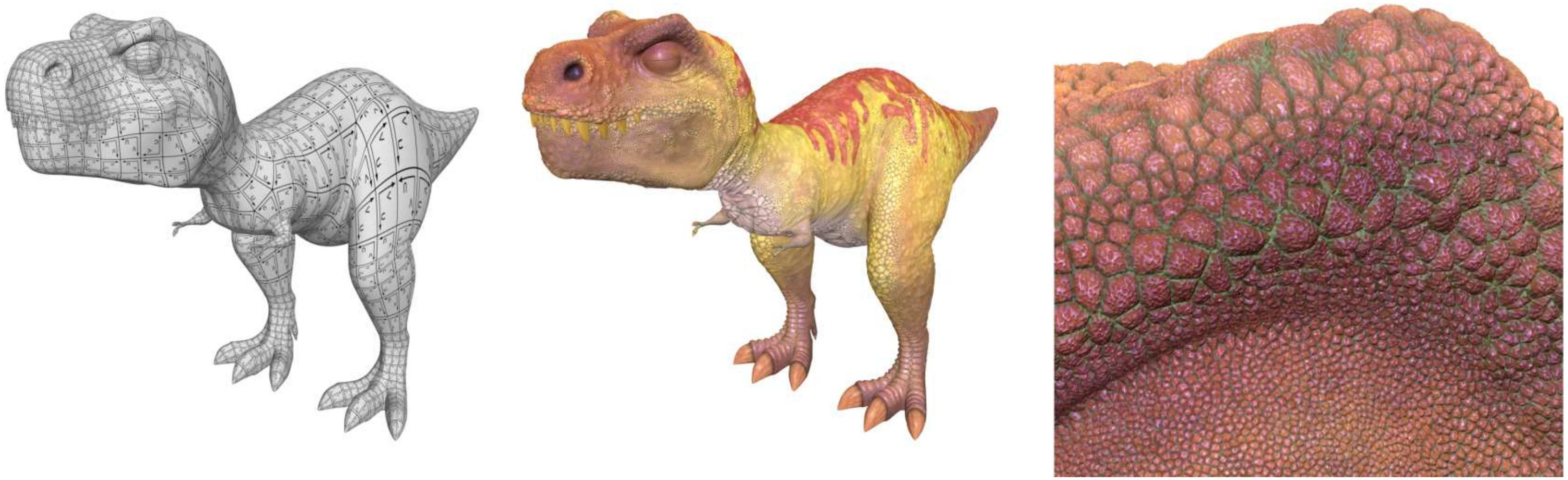
- **Soft shadow**
  - The boundary is smooth
  - Usually from area or environmental light sources
- **Shadow regions**





Texturing

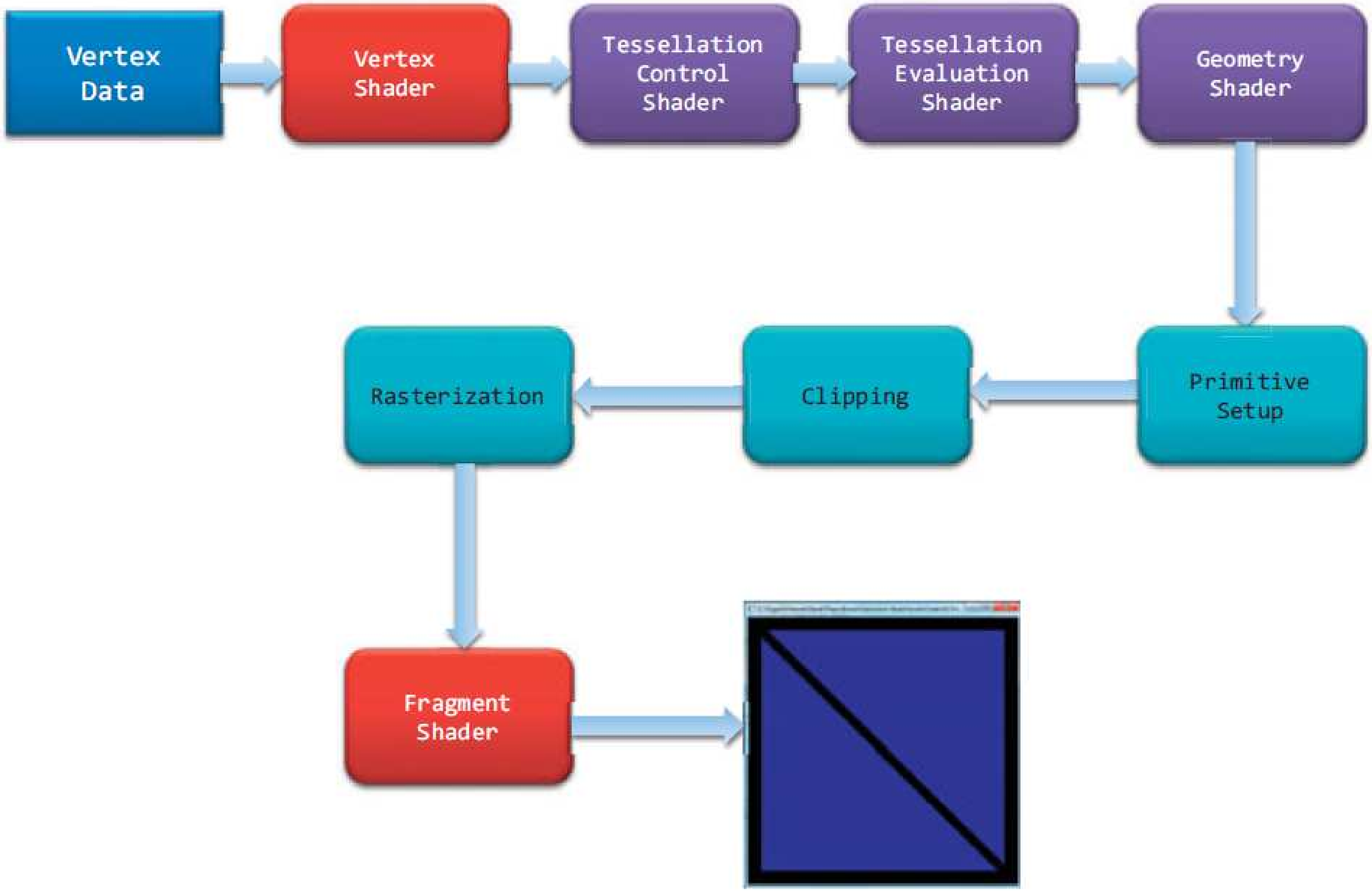
- **Texture map**
  - A texture map is an image applied (mapped) to the surface of a shape (mesh)
- **Texture mapping**
  - A mapping from image space to projection space



OpenGL Shader

OpenGL pipeline

- **OpenGL graphics pipeline**



Shader

- **Without-shader age**
  - Rendering pipeline is fixed
  - Only tune a few parameters
- **With-shader age**
  - Vertex shader
    - Processes each vertex separately
  - Tessellation shader
    - Generates additional geometry
  - Geometry shader
    - Modify entire geometric primitives
  - Fragment shader
    - A fragment's color and depth values are computed

Vertex shader

- **Processing of individual vertices before projection**
  - Receive a single vertex with attributes from the vertex stream
  - Generate a single vertex with modified attributes to the output vertex stream, in parallel

```
void
main()
{
    // set the normal for the fragment shader and
    // the vector from the vertex to the camera
    fragmentNormal = gl_Normal;
    cameraVector = cameraPosition - gl_Vertex.xyz;

    // set the vectors from the vertex to each light
    for(int i = 0; i < NUM_LIGHTS; ++i)
        lightVector[i] = lightPosition[i] - gl_Vertex.xyz;

    // output the transformed vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment (pixel) shader

- **Processing a fragment generated by rasterization**
  - After the rasterization process, with vertex attributes automatically interpolated
  - Take a single fragment as input and produce a single fragment as output, in parallel

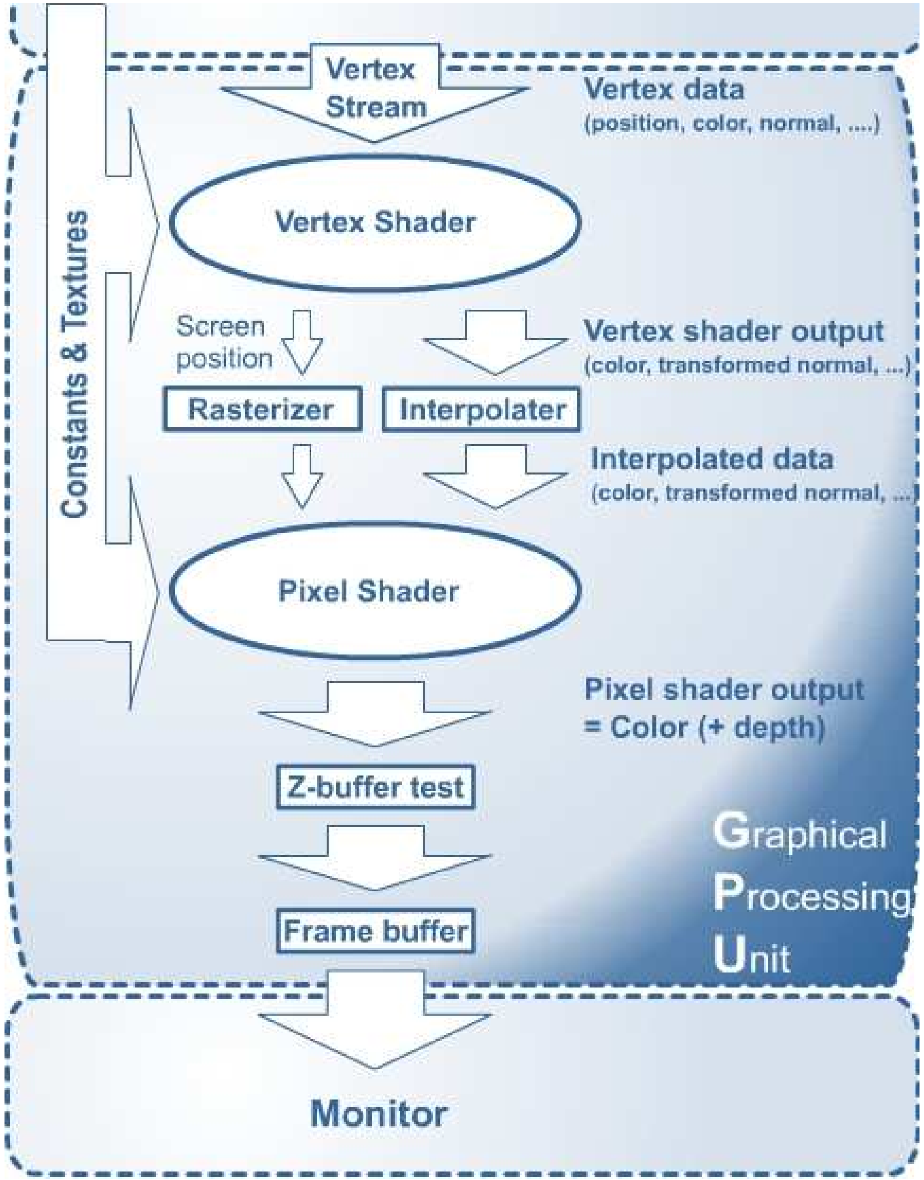
```
void
main()
{
    // initialize diffuse/specular lighting
    vec3 diffuse = vec3(0.0, 0.0, 0.0);
    vec3 specular = vec3(0.0, 0.0, 0.0);

    // normalize the fragment normal and camera direction
    vec3 normal = normalize(fragmentNormal);
    vec3 cameraDir = normalize(cameraVector);

    // loop through each light
    for(int i = 0; i < NUM_LIGHTS; ++i) {
        // calculate distance between 0.0 and 1.0
        float dist = min(dot(lightVector[i], lightVector[i]), MAX_DIST_SQUARED) / MAX_DIST_SQUARED;
        float distFactor = 1.0 - dist;

        // diffuse
        vec3 lightDir = normalize(lightVector[i]);
        float diffuseDot = dot(normal, lightDir);
        diffuse += lightColor[i] * clamp(diffuseDot, 0.0, 1.0) * distFactor;

        ...
    ...
```

Shader compilation

- **Compile at run-time (dynamic compilation)**
  - Reading a file or shader source strings and compile

```
static GLuint
shaderCompileFromFile(GLenum type, const char *filePath)
{
    char *source;
    GLuint shader;
    GLint length, result;

    /* get shader source */
    source = shaderLoadSource(filePath);
    if(!source)
        return 0;

    /* create shader object, set the source, and compile */
    shader = glCreateShader(type);
    length = strlen(source);
    glShaderSource(shader, 1, (const char **)&source, &length);
    glCompileShader(shader);
    free(source);

    /* make sure the compilation was successful */
    glGetShaderiv(shader, GL_COMPILE_STATUS, &result);
    if(result == GL_FALSE) {
        char *log;

        /* get the shader info log */
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
        log = malloc(length);
        glGetShaderInfoLog(shader, length, &result, log);

        /* print an error message and the info log */
        fprintf(stderr, "shaderCompileFromFile(): Unable to compile %s: %s\n", filePath, log);
        free(log);

        glDeleteShader(shader);
        return 0;
    }

    return shader;
}
```

Relation between vertex & fragment shader



Next Lecture :

Coordinate spaces, projection & rasterization