Midterm Review

Introduction to Computer Graphics

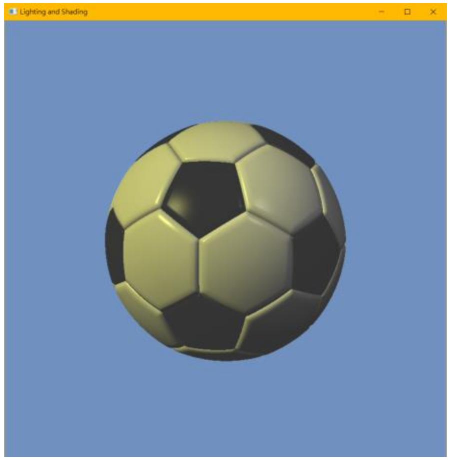
Yu-Ting Wu

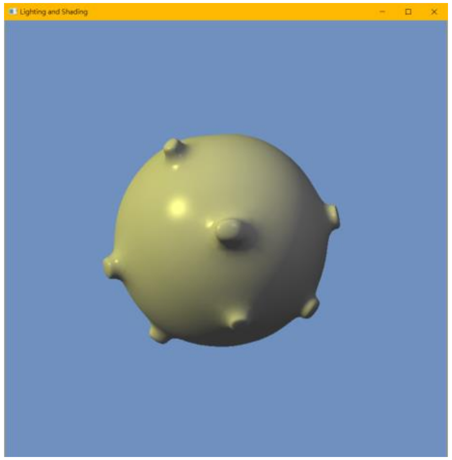
Announcement

- We **DO NOT** have a midterm exam for ICG
- There is **NO** class on Oct. 31 (moved to the **18th** week)
- The announcement of Homework#2 will be postponed to **Nov. 7**

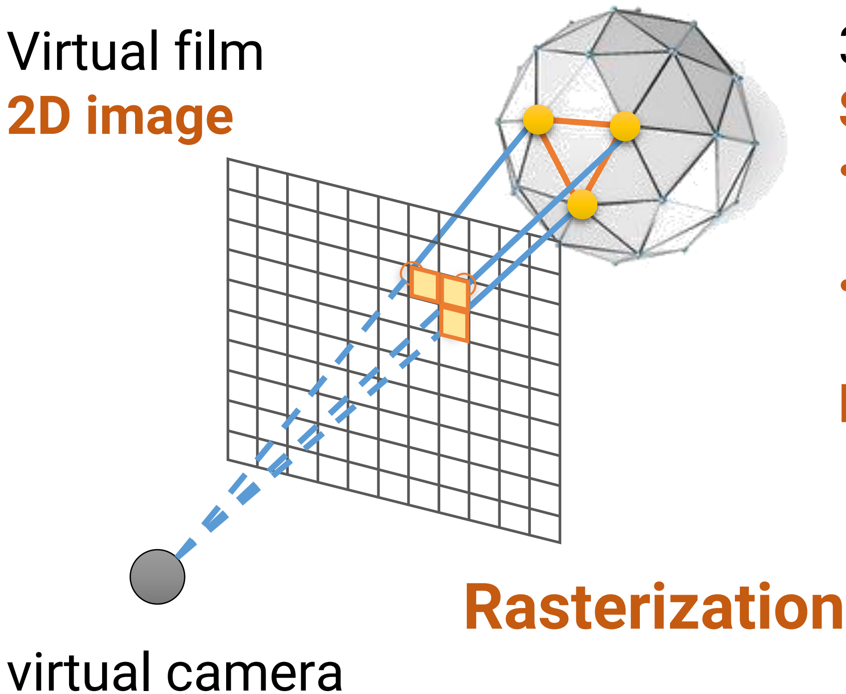
HW2 Spoiler

- Implement lighting with shaders





The Rendering Process



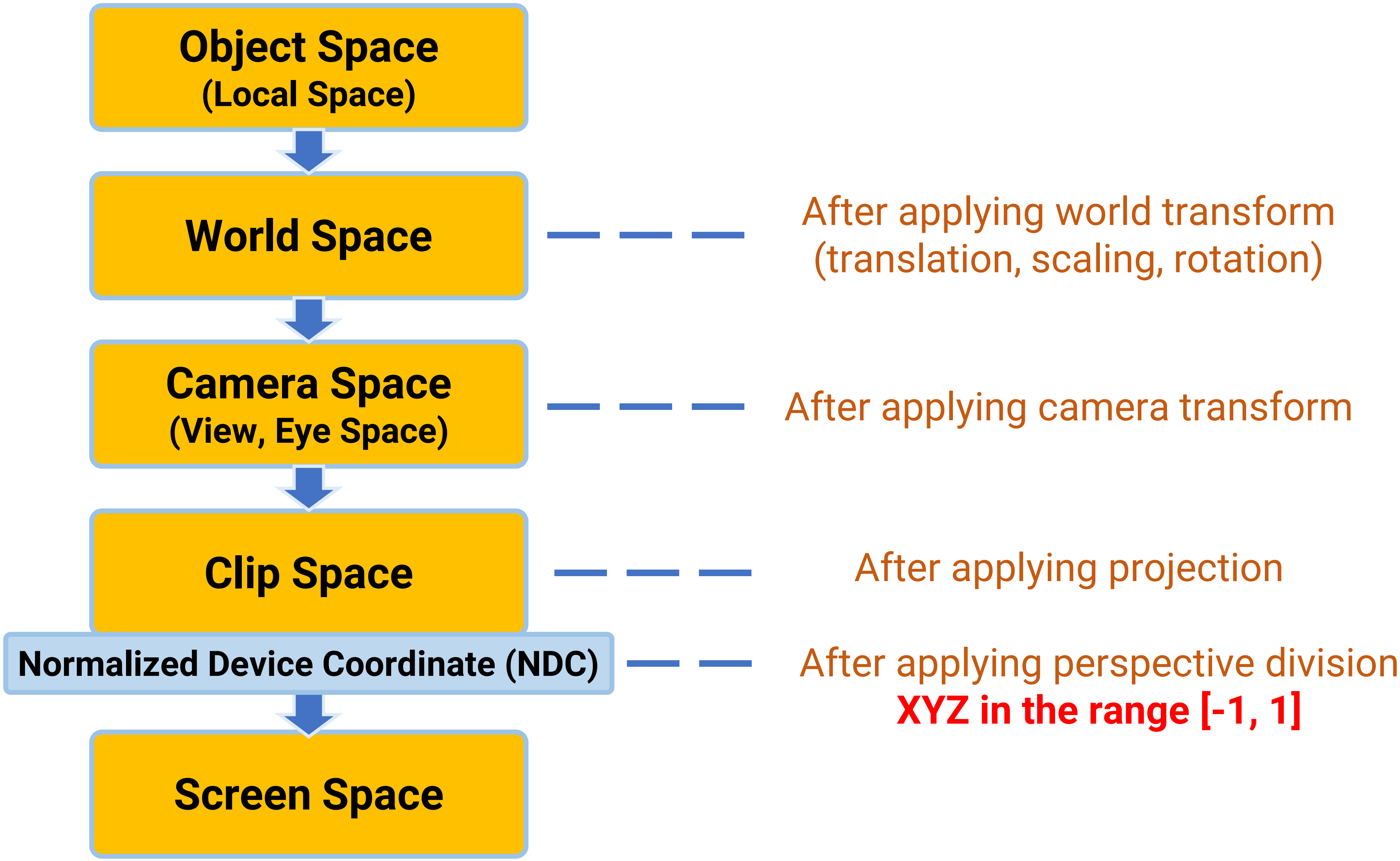
3D virtual scene

Scene objects

- 3D geometry data (triangle mesh)
- Material properties (*Phong* model)

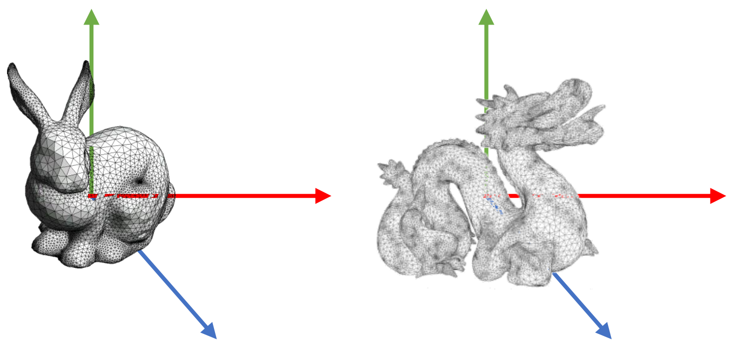
Lights

Transformation



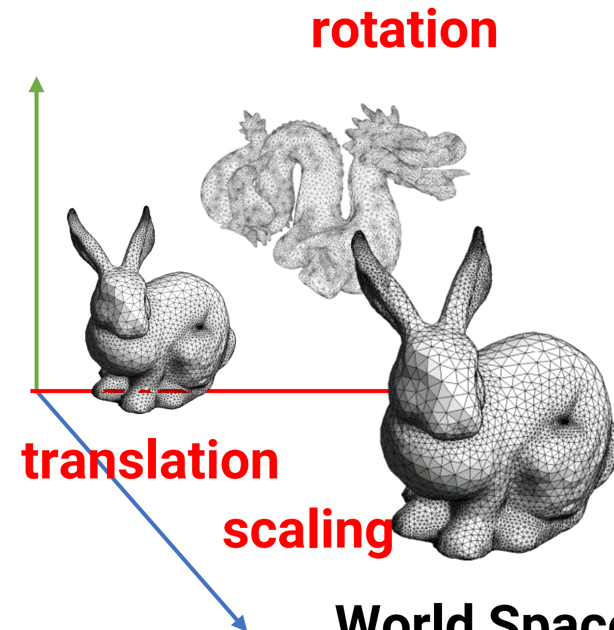
Object Space to World Space

Why? reuse models and save memory



Object Space

World Transformation



World Space

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translation

`glm::translate`

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scaling

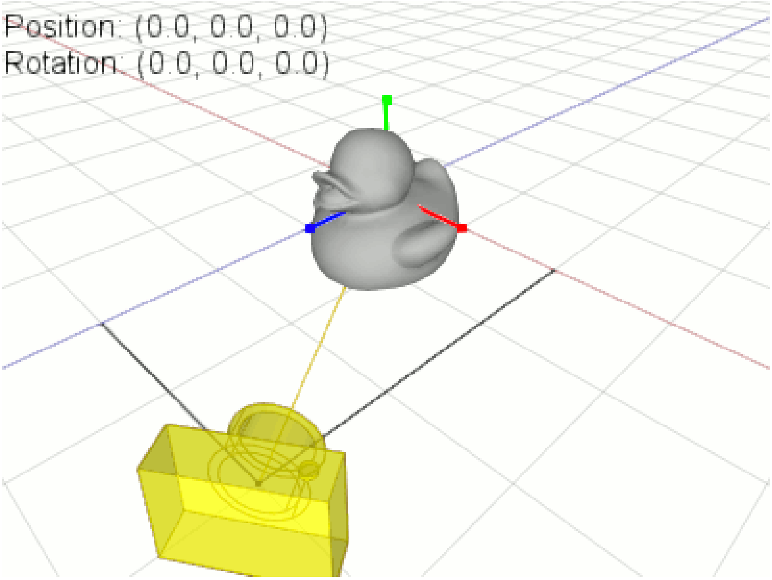
`glm::scale`

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation (Y)

`glm::rotate`

World Space to Camera Space

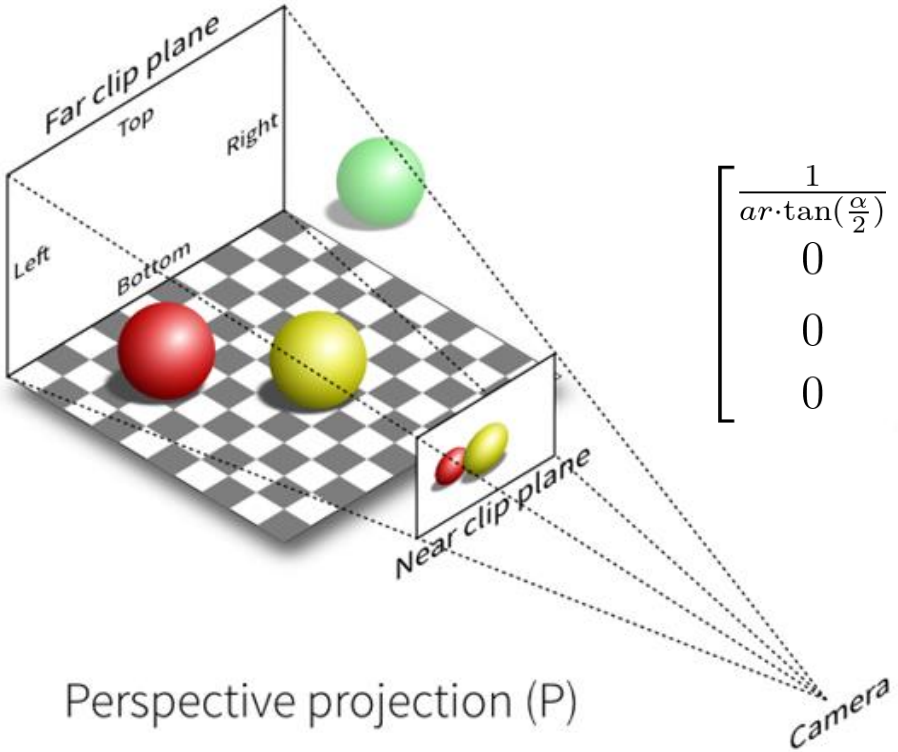


Why? for simpler math!

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
glm::lookAt(camPos, target, up);
```

Camera Space to Clip Space (NDC)



Perspective projection (P)

$$\begin{bmatrix} \frac{1}{ar \cdot \tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-nearZ - farZ}{nearZ - farZ} & \frac{2 \cdot farZ \cdot nearZ}{nearZ - farZ} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
glm::perspective(fovy, aspectRatio, nearZ, farZ);
```

CPU v.s. GPU Transformation

- CPU

prepare vertex data → apply vertex transform → create vertex buffer and upload vertex data to GPU → **GPU**

loop through all vertices

```
void ApplyTransformCPU(std::vector<glm::vec3>& vertexPositions, const glm::mat4x4& mvpMatrix)
{
    for (unsigned int i = 0 ; i < vertexPositions.size(); ++i) {
        glm::vec4 p = mvpMatrix * glm::vec4(vertexPositions[i], 1.0f);
        if (p.w != 0.0f) {
            float inv = 1.0f / p.w;
            vertexPositions[i].x = p.x * inv;
            vertexPositions[i].y = p.y * inv;
            vertexPositions[i].z = p.z * inv;
        }
    }
}
```

CPU v.s. GPU Transformation (cont.)

- GPU

prepare vertex data → create vertex buffer and upload vertex data to GPU → **GPU** → apply vertex transform

```
locMVP = glGetUniformLocation(shaderProgId, "MVP");
glUniformMatrix4fv(locMVP, 1, GL_FALSE, glm::value_ptr(MVP));
```

CPU

1

Vertex Shader

GPU

2

```
#version 330 core
layout (location = 0) in vec3 Position;
uniform mat4 MVP;
void main() {
    gl_Position = MVP * vec4(Position, 1.0);
}
```

No loop because the vertex shader is executed for each vertex **in parallel** by nature

Implementation

- In the **CPU** application, we
  - **Load the scene data (from files)**
    - Create vertex and index buffers
    - Provide material properties
    - Setup lights
  - **Load and create shaders**
  - **Setup the rendering state (via OpenGL APIs)**
    - Background color, polygon mode … etc.
  - **Set variable values to the GPU shaders**
    - Transformation matrices, material data, light data … etc.
  - **Call "Draw" functions to render objects (via OpenGL APIs)**
    - Vertex buffer format, primitive type, # of indices

Implementation (cont.)

- On the **GPU**, we
  - Execute the **Vertex Shader** for each vertex that belongs to a triangle
    - Vertex transformation
    - Vertex lighting (optional)
    - Interpolate vertex attributes (pass to fragment shader)

OpenGL performs **rasterization** by **hardware**

  - Execute the **Fragment Shader** for each fragment generated by the rasterization for each triangle
    - Fragment shading (lighting, texturing … etc.)

Vertex Buffer

- Store vertex data (attributes)
  - Position
  - Normal
  - Texture coordinate
  - Others

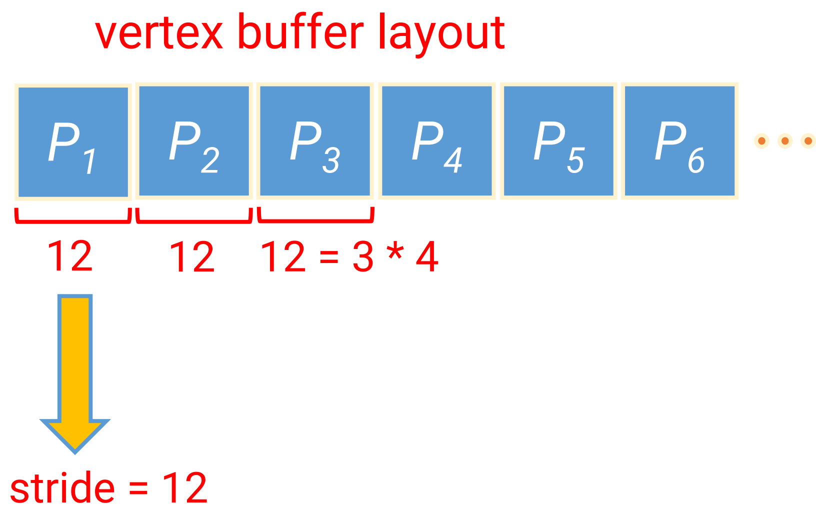
in **Object Space** if transformation is performed by the Vertex Shader on GPU (otherwise, in **Clip Space**)

- Upload to GPU for rendering

Vertex Buffer Layout

- Depend on the vertex attributes you provide
- Example: only position data

```
// VertexP Declarations.
struct VertexP
{
    VertexP() {
        position = glm::vec3(0.0f, 0.0f, 0.0f);
    }
    VertexP(glm::vec3 p) {
        position = p;
    }
    glm::vec3 position;
};
```



- During rendering

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexP), 0);
```

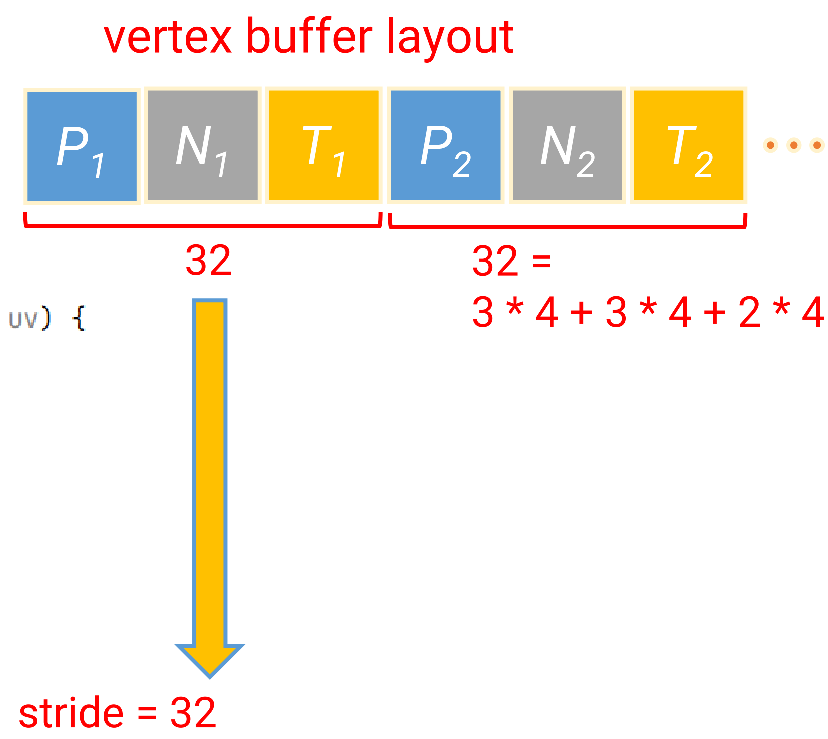
Note in ***"Implementation: Simple Drawing"***, we set the stride to 0 because OpenGL allows doing so if there is only 1 attribute and the data is tightly packed

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

Vertex Buffer Layout (cont.)

- Depend on the vertex attributes you provide
- Example: with position/normal/texcoord data

```
// VertexPTN Declarations.
struct VertexPTN
{
    VertexPTN() {
        position = glm::vec3(0.0f, 0.0f, 0.0f);
        normal = glm::vec3(0.0f, 1.0f, 0.0f);
        texcoord = glm::vec2(0.0f, 0.0f);
    }
    VertexPTN(glm::vec3 p, glm::vec3 n, glm::vec2 uv) {
        position = p;
        normal = n;
        texcoord = uv;
    }
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec2 texcoord;
};
```

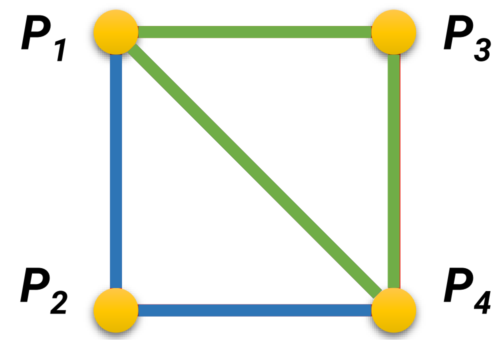


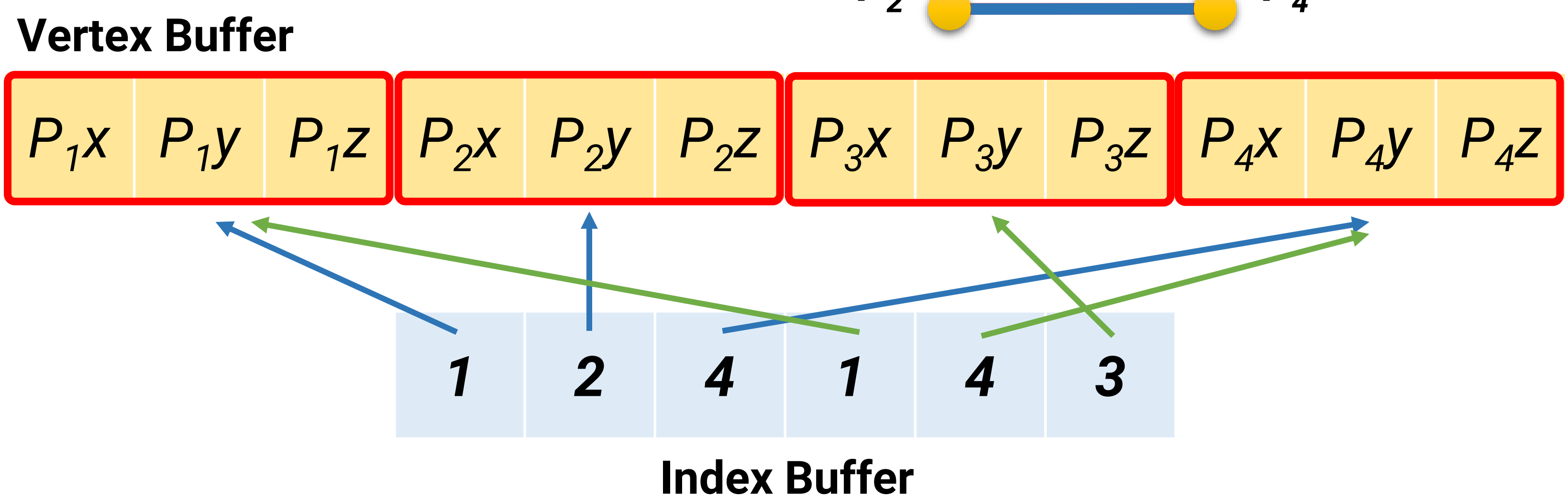
- During rendering:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPTN), 0);
```

Index Buffer

- Reduce the size of a vertex buffer by reusing vertex data
- When forming a triangle, we specify the indices of vertices in the vertex buffer



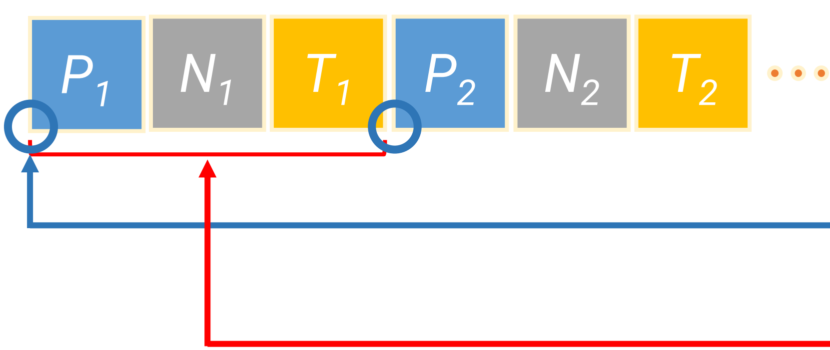


Rendering with Vertex/Index Buffer

- Render with only the position attribute

```
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPTN), 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);
glDrawElements(GL_TRIANGLES, GetNumIndices(), GL_UNSIGNED_INT, 0);
glDisableVertexAttribArray(0);
```

vertex buffer layout



the byte offset of the first element of the attribute

stride = 32

Rendering with Vertex/Index Buffer (cont.)

- Render with only the position and normal attributes

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPTN), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexPTN), (const GLvoid*)12);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);
glDrawElements(GL_TRIANGLES, GetNumIndices(), GL_UNSIGNED_INT, 0);
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
```

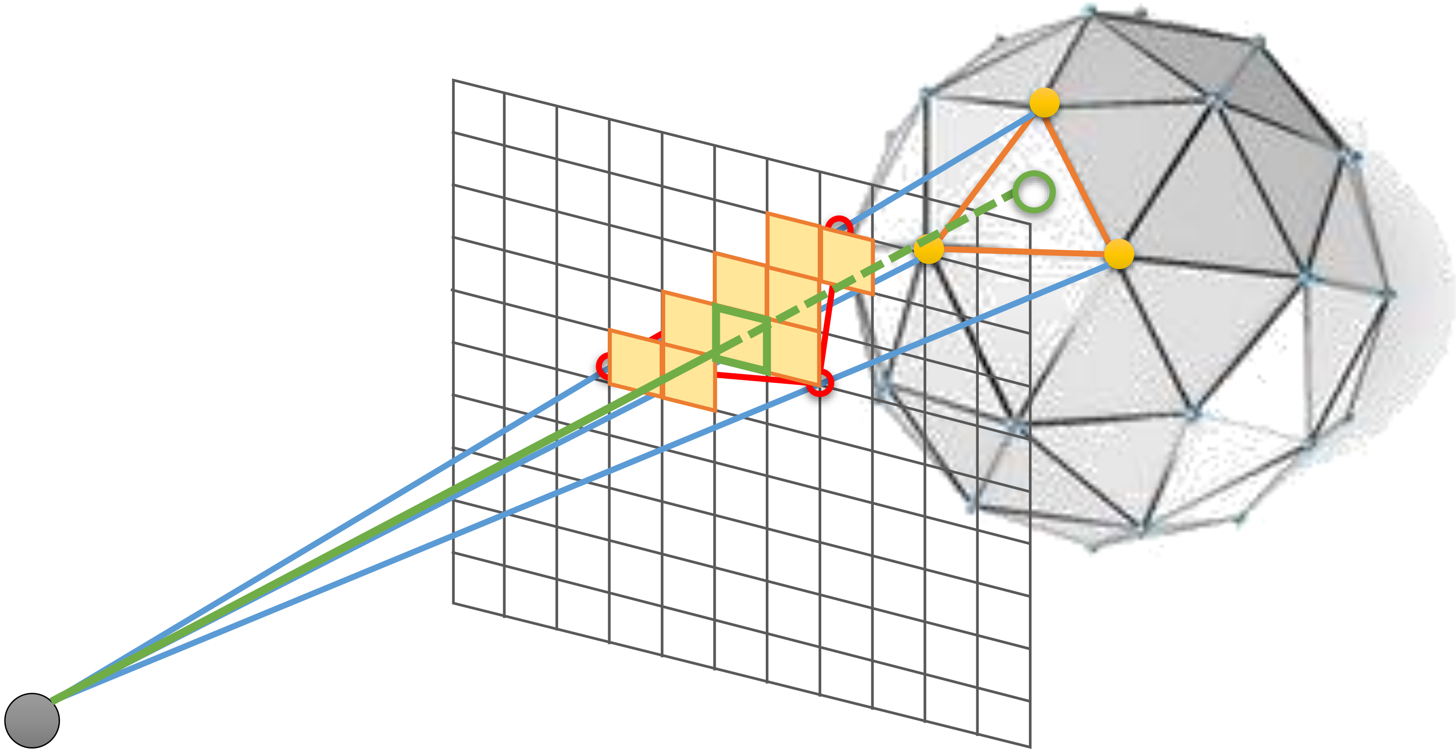
vertex buffer layout

the byte offset of the first element of the attribute

stride = 32

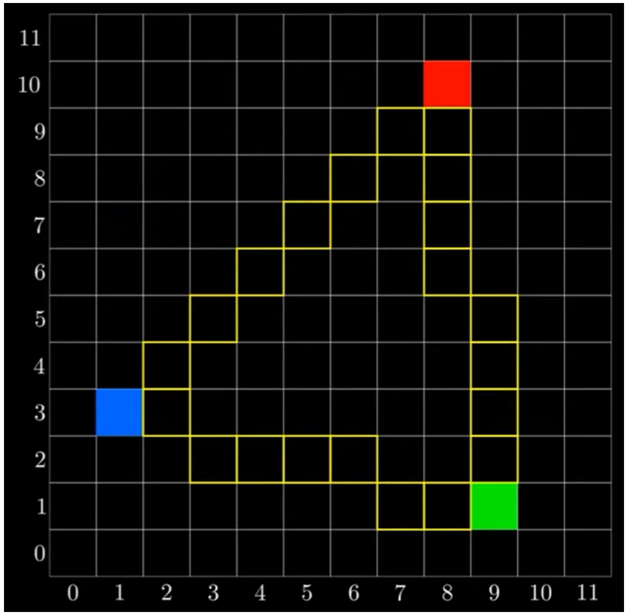
Rasterization

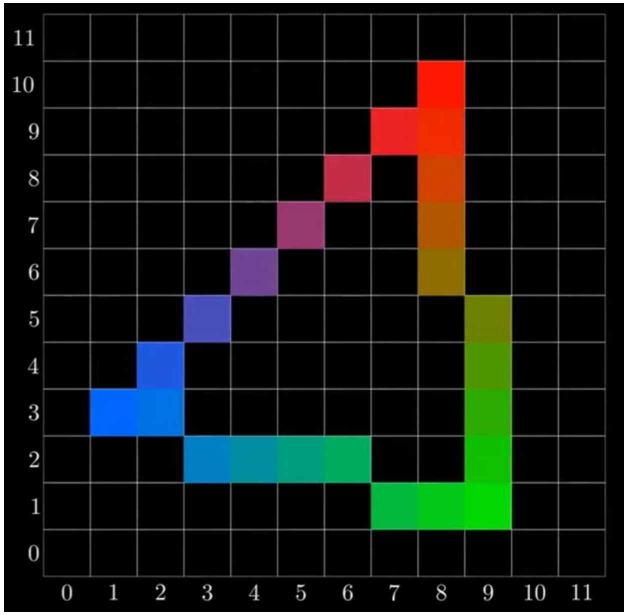
- Generate **fragments** for each triangle
- Interpolate vertex attributes at each fragment



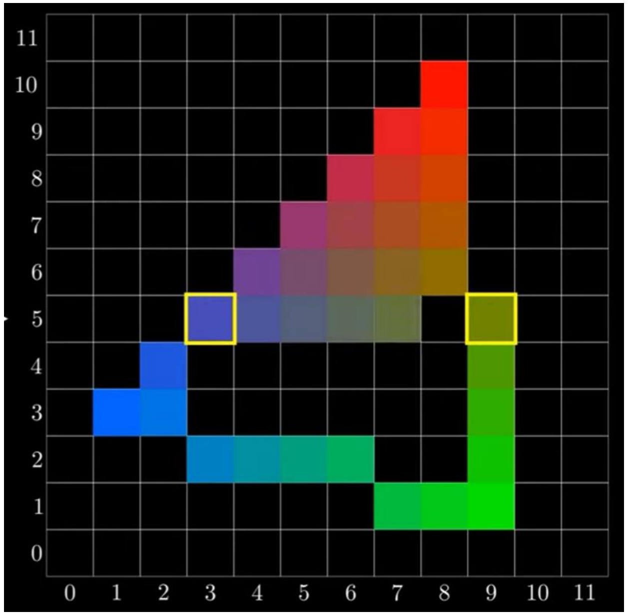
Vertex Attribute Interpolation

- Color





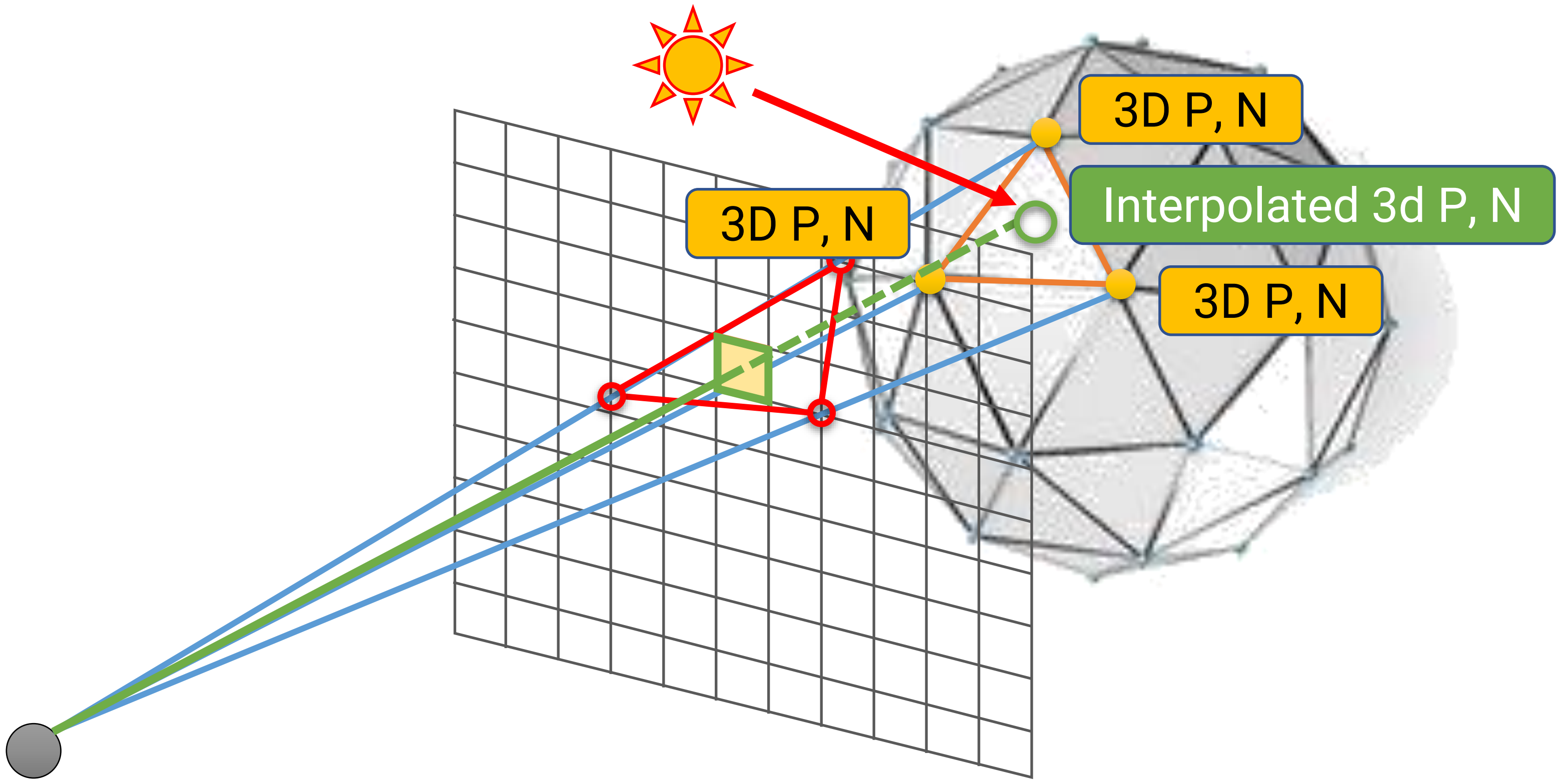
Attributes interpolation of edge pixels using vertices



Attributes interpolation of inner pixels using edge points

Vertex Attribute Interpolation (cont.)

- If we want to compute lighting at each fragment (in the fragment shader), we need per-fragment geometry attributes such as 3D position and normal



Vertex Attribute Interpolation (cont.)

- Example: interpolate **world-space vertex position** and **world-space vertex normal**

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 Position;
layout (location = 1) in vec3 Normal;

// Transformation matrix.
uniform mat4 worldMatrix;
uniform mat4 normalMatrix;
uniform mat4 MVP;

// Data pass to fragment shader.
out vec3 iPosWorld;
out vec3 iNormalWorld;

void main()
{
    gl_Position = MVP * vec4(Position, 1.0);

    // Pass vertex attributes.
    vec4 positionTmp = worldMatrix * vec4(Position, 1.0);
    iPosWorld = positionTmp.xyz / positionTmp.w;

    iNormalWorld = (normalMatrix * vec4(Normal, 0.0)).xyz;
}
```

Tell OpenGL you want to interpolate these attributes

world matrix for transforming normal

Fragment Shader

```
#version 330 core

// Data from vertex shader.
in vec3 iPosWorld;
in vec3 iNormalWorld;

out vec4 FragColor;

void main()
{
    vec3 N = normalize(iNormalWorld);
    vec3 visColor = 0.5 * N + 0.5;
    FragColor = vec4(N, 1.0);
}
```

Ensure the interpolated normal has a unit length

Map the range of normal from [-1, 1] to [0, 1] for visualization

Vertex Attribute Interpolation (cont.)

- Remember the homogeneous coordinate for a 3D point *(x, y, z)* is *(x, y ,z,* ***1****)*
  - Why? To enable the combination of a **translation** matrix with other transformation matrices

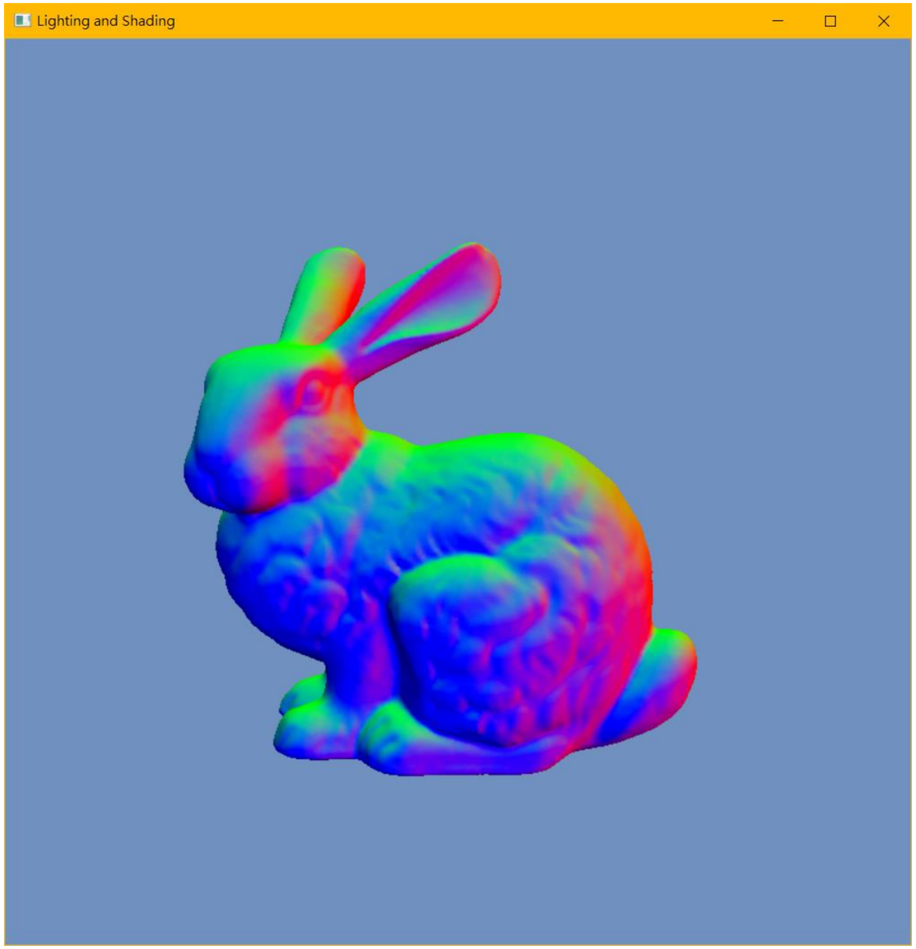
$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \\ z' &= z + t_z \end{aligned}$$

- When transforming a vector, we represent a 3D direction *(dx, dy, dz)* by *(dx, dy, dz,* ***0****)* because we do not want a translation for "direction"
  - Otherwise, the direction (0.578, 0.578, 0.578) will become (3.578, 4.578, 5.578) after a translation of (3, 4, 5)

Vertex Attribute Interpolation (cont.)

visualize world-space position as color



visualize world-space normal as color

Any Questions?