EECE5698
Networked XR Systems

Some of the slide credits: Anthony Rowe

Lecture Outline for Today

- Rendering Basics
- Edge Rendering

Rendering Basics

- Wiki definition: “Rendering or image synthesis is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program. ”

Rendering Basics

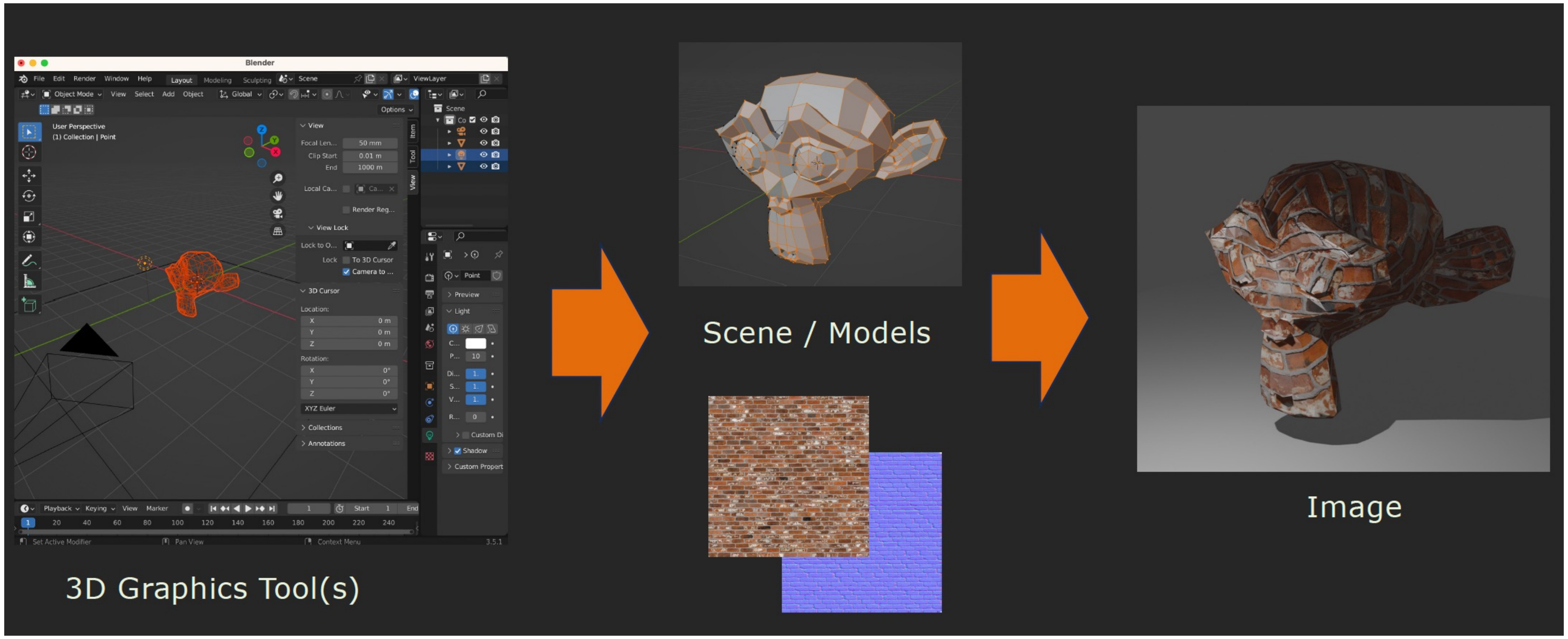
- Rendering is crucial in various fields such as video games, simulations, movie production, and virtual reality, providing the final appearance of models and scenes with textures, colors, and lighting.
- Key Components:
    - **Models:** The geometric data representing 3D objects.
    - **Textures:** The surface details that give materials their appearance.
    - **Lighting:** The simulation of light to create shadows, highlights, and color variations.

Rendering Basics

- **Real-time Rendering:**
  - Used in video games and interactive graphics where images must be generated at a rapid pace, typically 30 to 60 frames per second.
  - Prioritizes speed over image quality, employing various optimizations to achieve smooth performance.
- **Offline (Pre-rendered) Rendering:**
  - Utilized in situations where image quality is paramount, such as in feature films and high-quality animations.
  - Takes more time to produce a single frame but achieves higher levels of detail and lighting accuracy.
- **Ray Tracing vs. Rasterization:**
  - **Ray Tracing:** Simulates the physical behavior of light to produce more realistic images, calculating reflections, refractions, and shadows.
  - **Rasterization:** Converts 3D models into 2D images quickly, often used in real-time rendering, but less capable of complex light interactions compared to ray tracing.

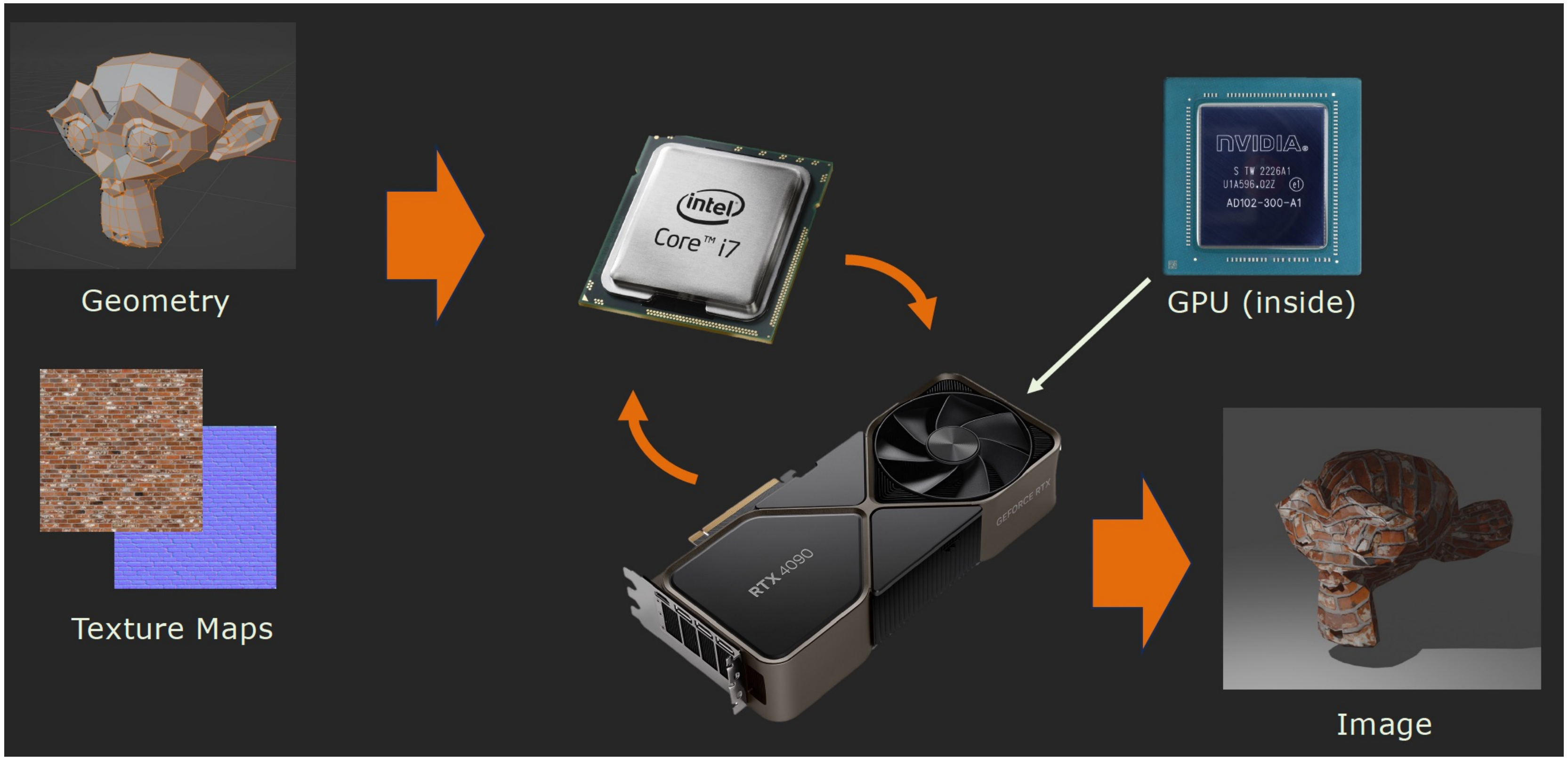
Rendering Basics

High level



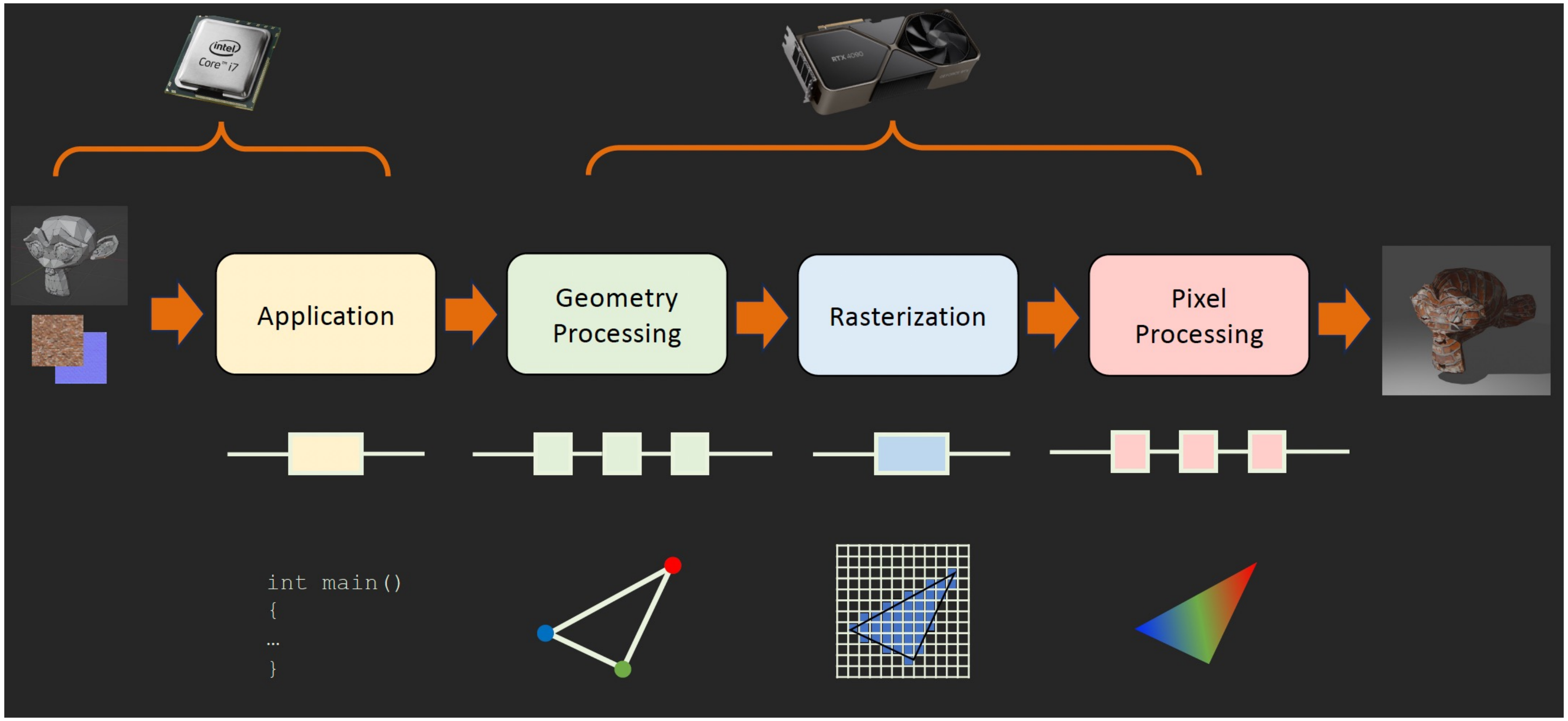
Rendering Basics

Hardware view



Rendering Basics

- Rendering pipeline



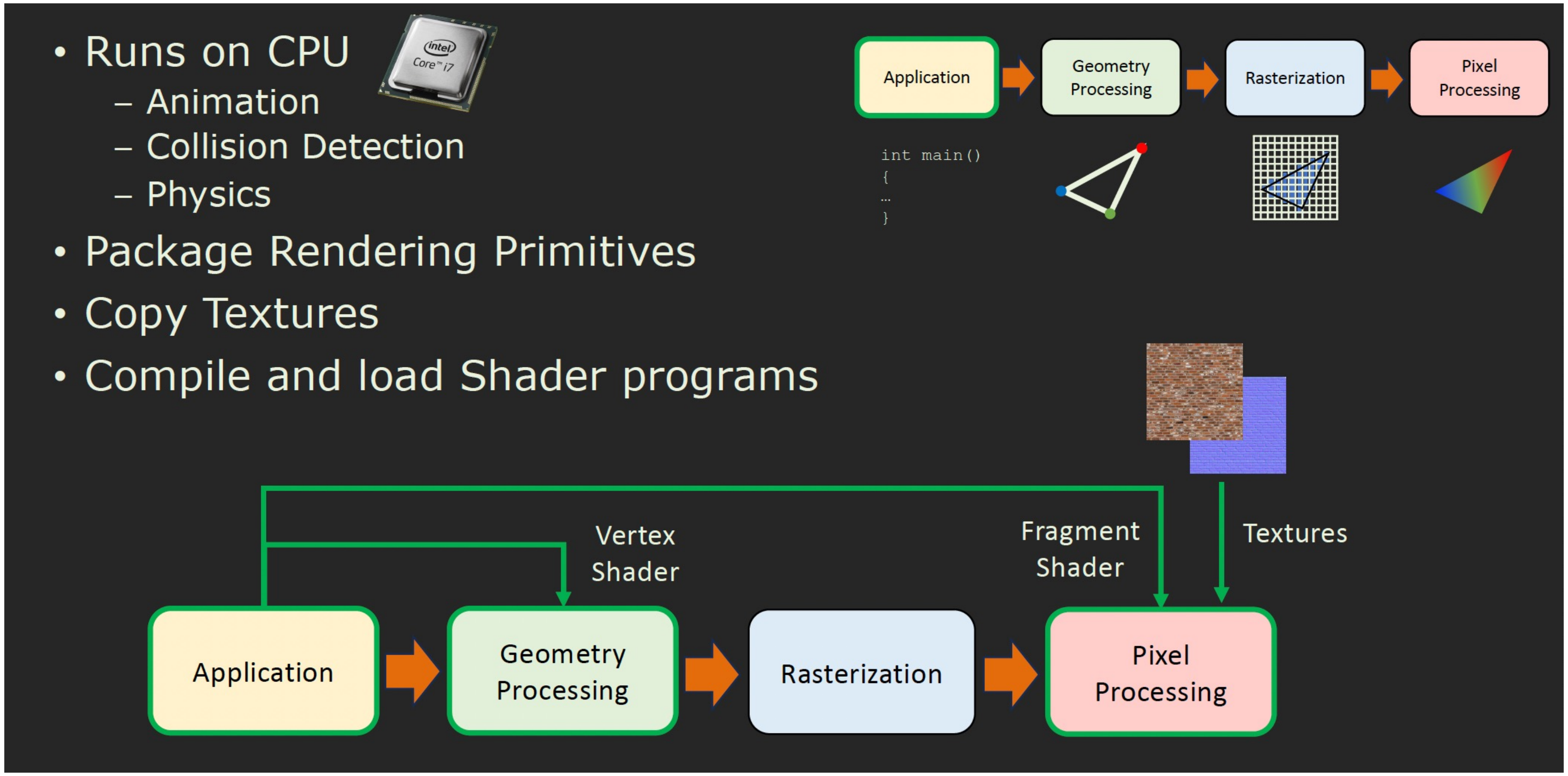
```
int main()
{
…
}
```

Rendering Basics – Key Steps

1. **Model Loading:** Importing 3D models into the rendering engine.
2. **Scene Setup:** Positioning models, lights, and cameras within the scene.
3. **Geometry Processing:** Transforming 3D coordinates to 2D screen space.
4. **Rasterization:** Converting 3D models into pixels on a 2D surface.
5. **Shading:** Applying textures, colors, and lighting effects.
6. **Output:** Rendering the final image for display or storage.

Rendering Basics

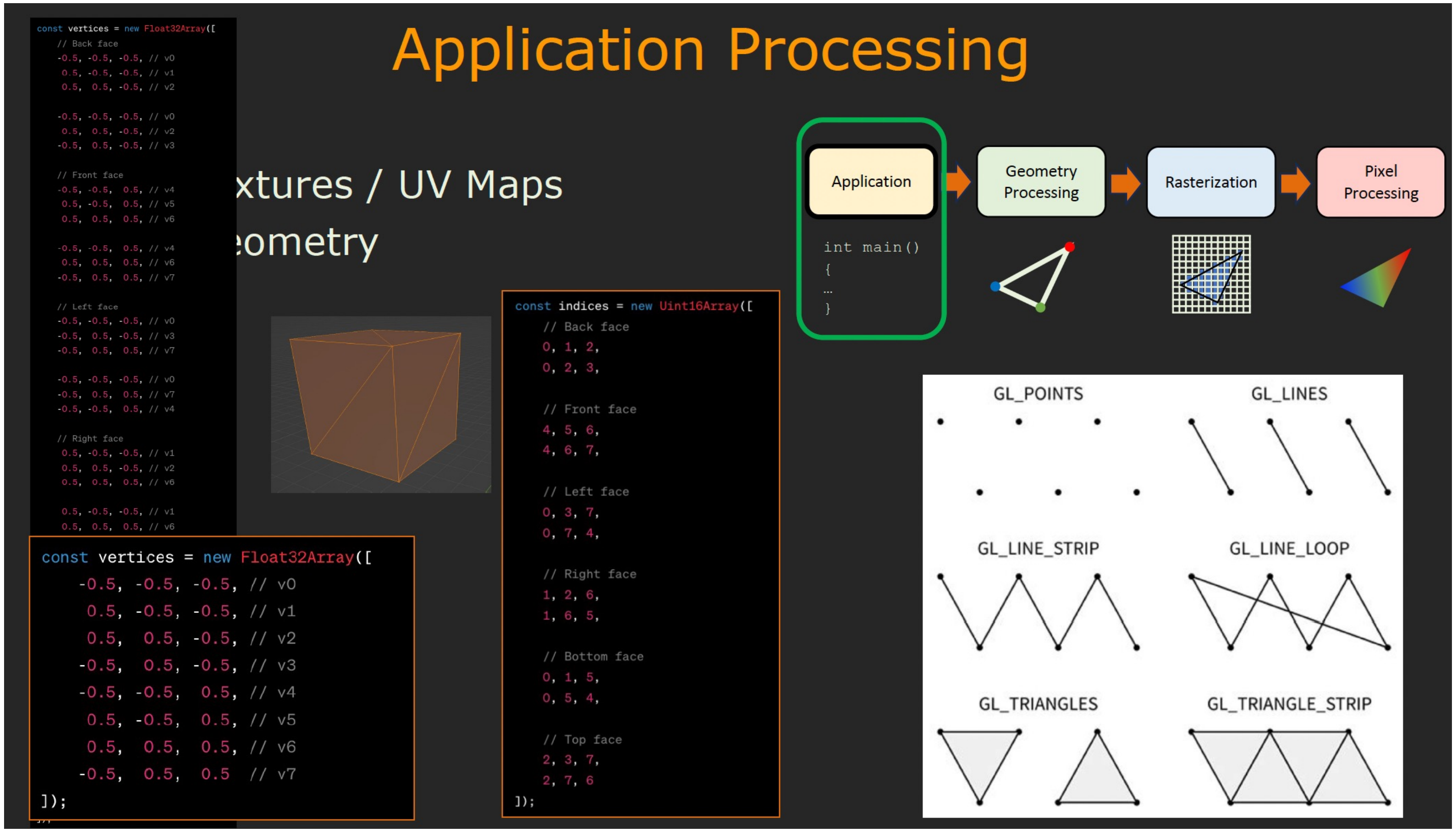
- Application processing



- Runs on CPU
  - Animation
  - Collision Detection
  - Physics
- Package Rendering Primitives
- Copy Textures
- Compile and load Shader programs

```
int main()
{
…
}
```

Rendering Basics



```
const vertices = new Float32Array([
    // Back face
    -0.5, -0.5, -0.5, // v0
     0.5, -0.5, -0.5, // v1
     0.5,  0.5, -0.5, // v2

    -0.5, -0.5, -0.5, // v0
     0.5,  0.5, -0.5, // v2
    -0.5,  0.5, -0.5, // v3

    // Front face
    -0.5, -0.5,  0.5, // v4
     0.5, -0.5,  0.5, // v5
     0.5,  0.5,  0.5, // v6

    -0.5, -0.5,  0.5, // v4
     0.5,  0.5,  0.5, // v6
    -0.5,  0.5,  0.5, // v7

    // Left face
    -0.5, -0.5, -0.5, // v0
    -0.5,  0.5, -0.5, // v3
    -0.5,  0.5,  0.5, // v7

    -0.5, -0.5, -0.5, // v0
    -0.5,  0.5,  0.5, // v7
    -0.5, -0.5,  0.5, // v4

    // Right face
     0.5, -0.5, -0.5, // v1
     0.5,  0.5, -0.5, // v2
     0.5,  0.5,  0.5, // v6

     0.5, -0.5, -0.5, // v1
     0.5,  0.5,  0.5, // v6
```

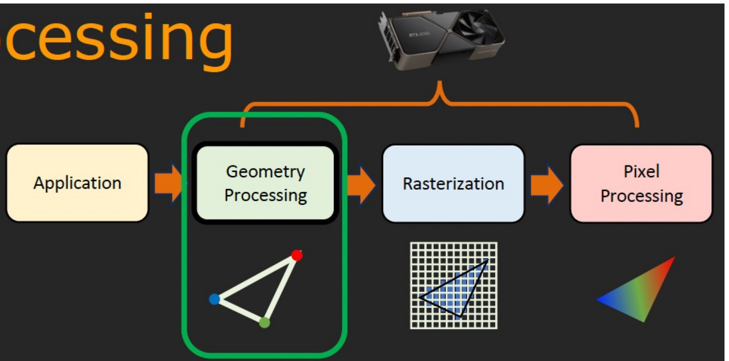
```
const vertices = new Float32Array([
    -0.5, -0.5, -0.5, // v0
     0.5, -0.5, -0.5, // v1
     0.5,  0.5, -0.5, // v2
    -0.5,  0.5, -0.5, // v3
    -0.5, -0.5,  0.5, // v4
     0.5, -0.5,  0.5, // v5
     0.5,  0.5,  0.5, // v6
    -0.5,  0.5,  0.5  // v7
]);
```

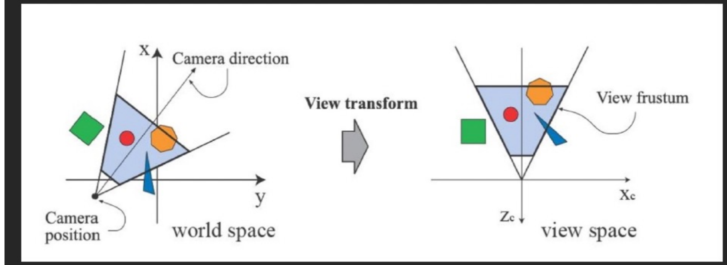
```
const indices = new Uint16Array([
    // Back face
    0, 1, 2,
    0, 2, 3,

    // Front face
    4, 5, 6,
    4, 6, 7,

    // Left face
    0, 3, 7,
    0, 7, 4,

    // Right face
    1, 2, 6,
    1, 6, 5,

    // Bottom face
    0, 1, 5,
    0, 5, 4,

    // Top face
    2, 3, 7,
    2, 7, 6
]);
```

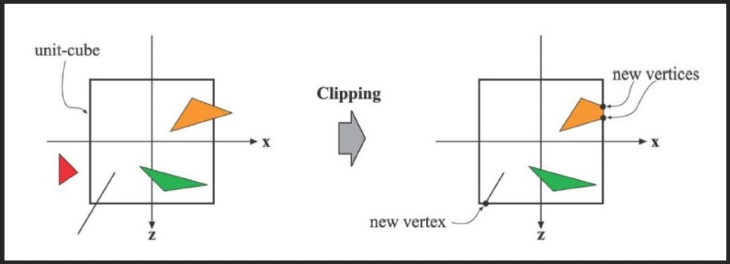
Rendering Basics

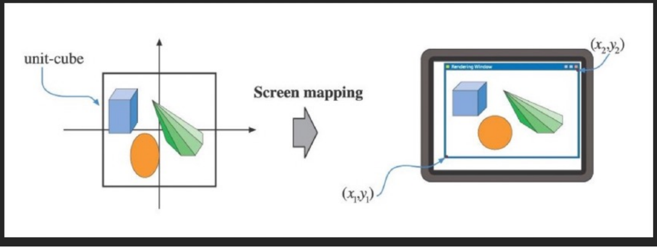
Geometry Processing

- Per-Triangle and Per-Vertex Ops
  - Vertex Shader (and others)
  - Coordinate Transform
  - Clipping
  - Screen Mapping

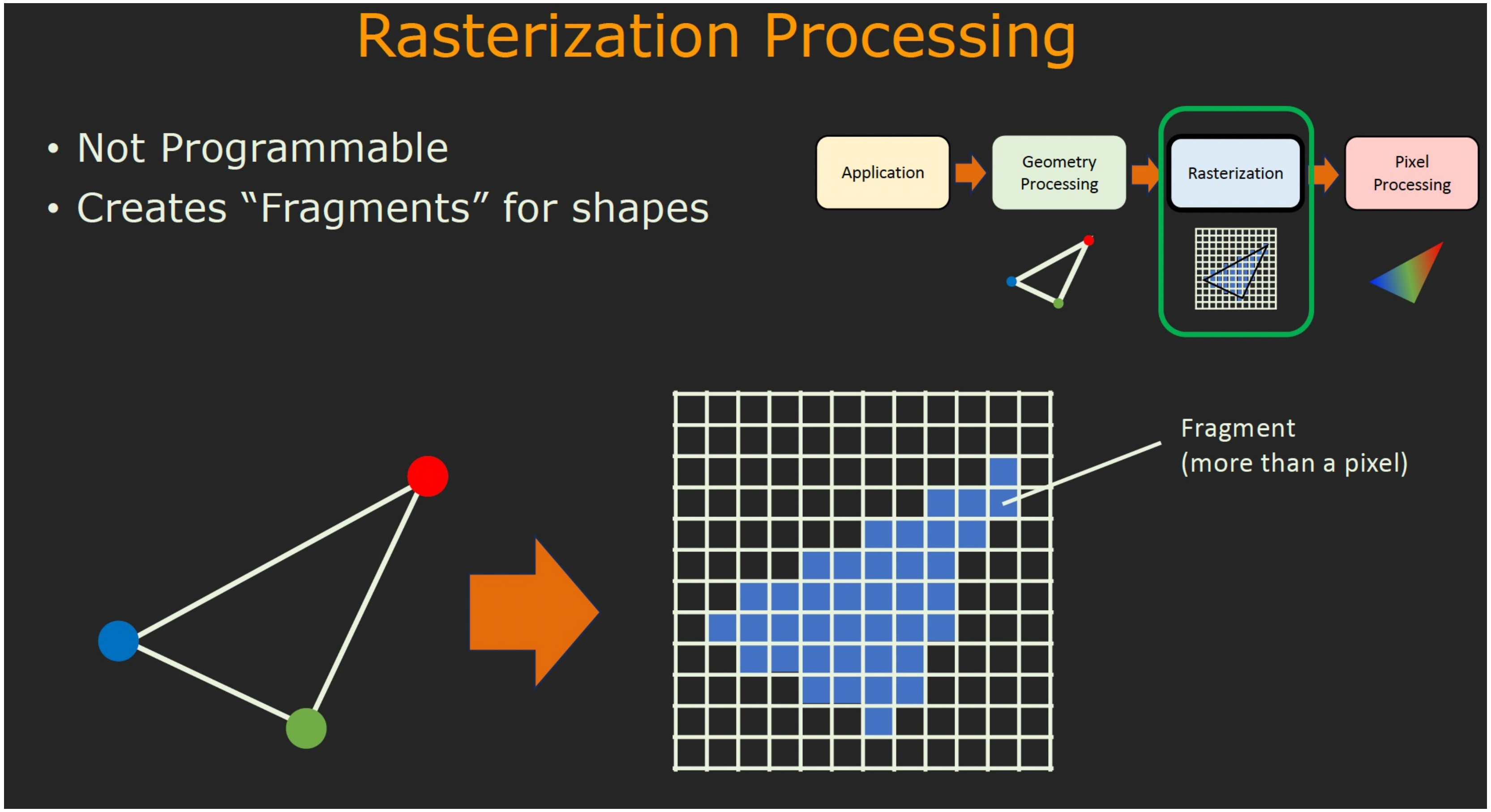








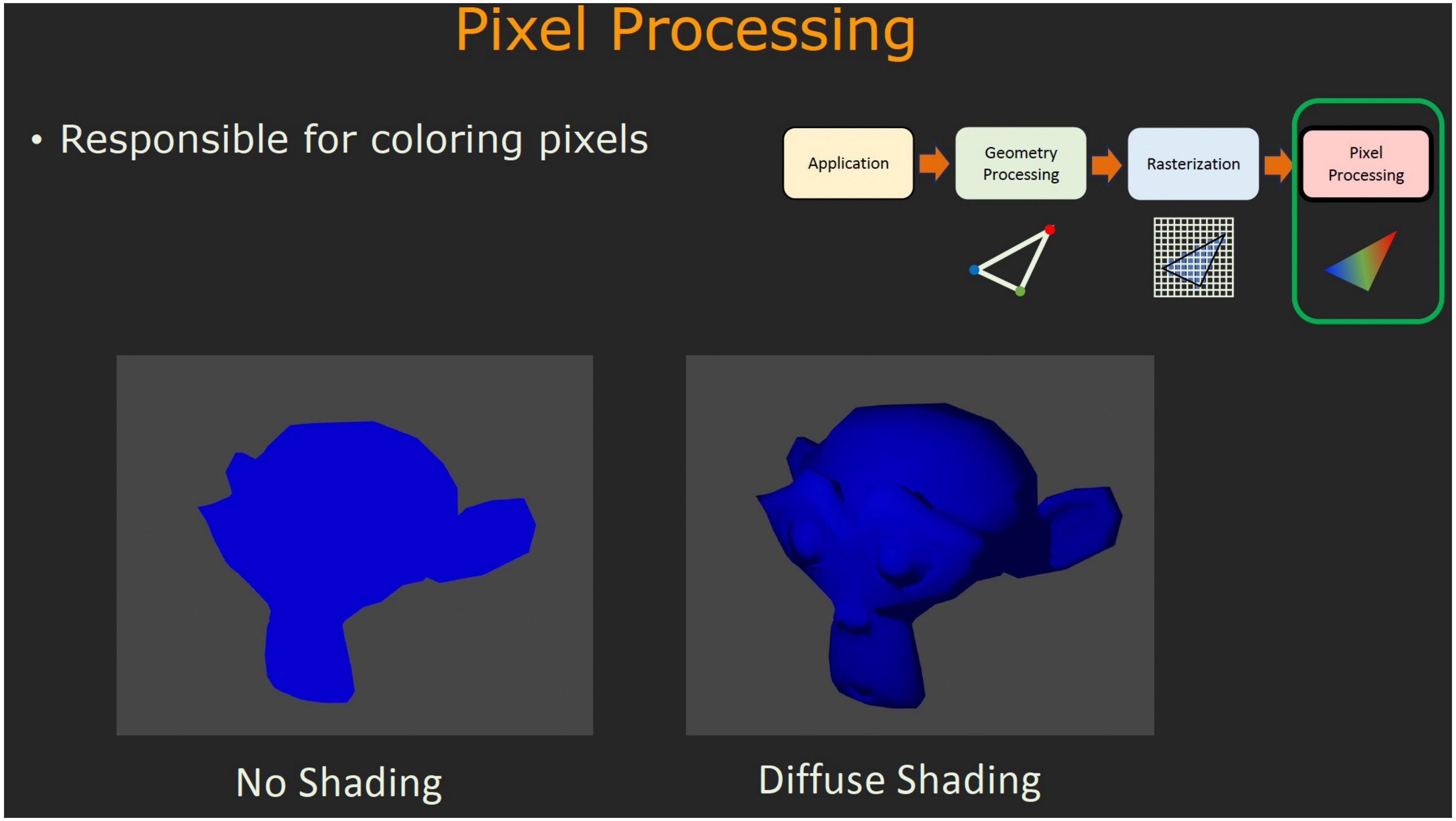
Rendering Basics



Rasterization Processing

- Not Programmable
- Creates “Fragments” for shapes

Rendering Basics



Pixel Processing

- Responsible for coloring pixels

Application
Geometry Processing
Rasterization
Pixel Processing
No Shading
Diffuse Shading

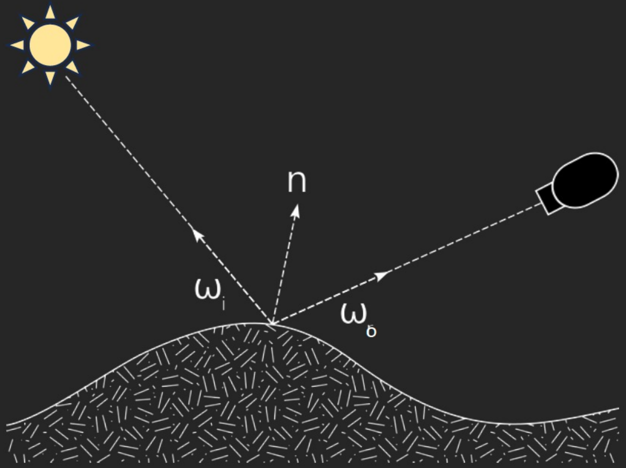
Rendering Basics

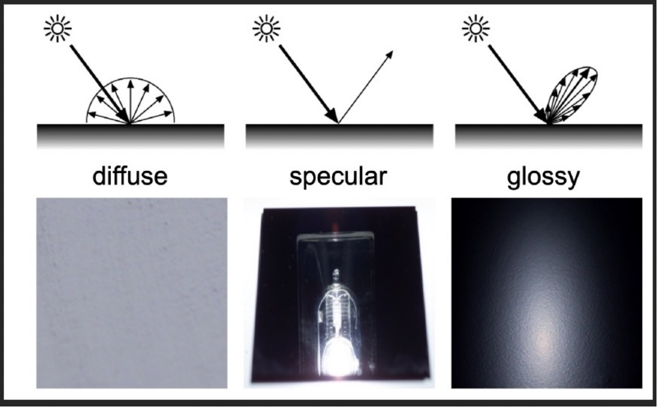
The Rendering Equation (simple)

Bidirectional Reflectance Distribution Function (BRDF)

$$L_o(\omega_o) = \int_\Omega L_i(\omega_i) \cos\theta_i \; f_r(\omega_i, \omega_o) \; d\omega_i$$

Geometry Term





Cem Yuksel, Utah: https://www.youtube.com/watch?v=GOfzX7kRwys

Rendering Basics

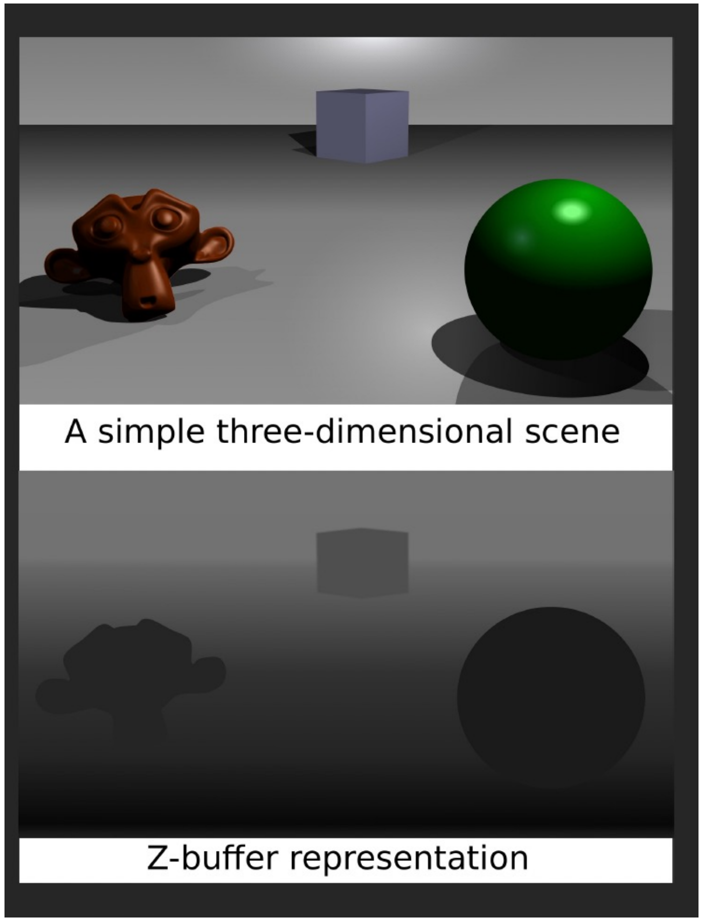
- Lighting and Shadows
  - Lighting Types:
    - Ambient: Soft, directionless light that simulates indirect lighting.
    - Directional: Parallel light rays, mimicking sunlight.
    - Point: Light emitted from a single point, radiating in all directions.
    - Spot: Light emitted in a cone shape, like a flashlight.
  - Shadows: Essential for depth and realism, with techniques like shadow mapping and ray-traced shadows to simulate how objects block light.

Rendering Basics

Z-Buffer – Composite depth map after rasterization

Z-Culling – Early z-test to avoid pixel processing

Z-Test – Depth test in pixel processing stage



Rendering Basics

- Shaders and the GPU
  - Shaders: Small programs that run on the GPU to perform custom rendering effects, such as lighting calculations, texturing, and color adjustments.
  - Types of Shaders: Vertex shaders, Fragment (or Pixel) shaders, Geometry shaders, etc.
  - GPU’s Role: Executes shaders to render images quickly, efficiently handling complex calculations required for realistic rendering.

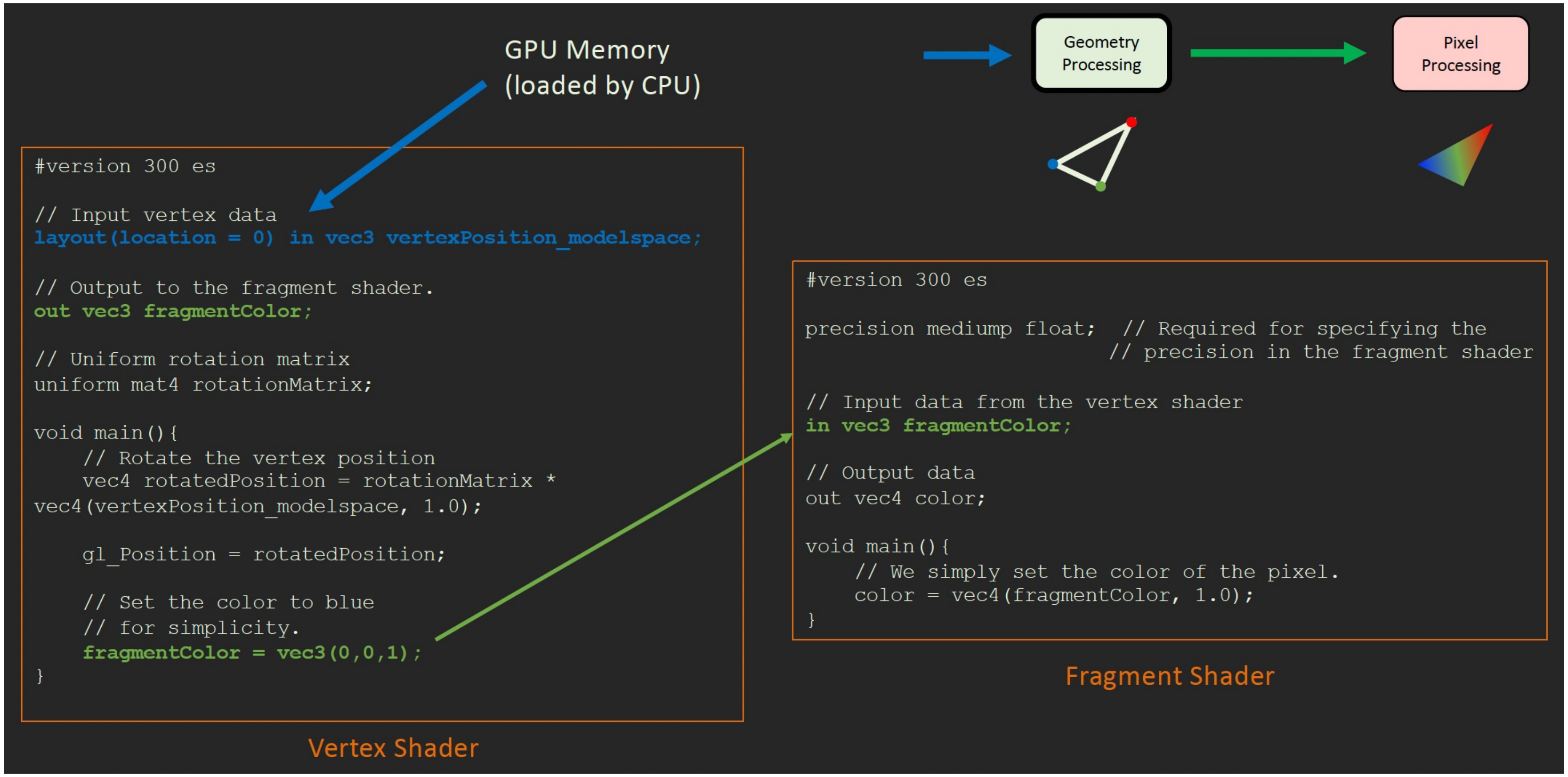
Rendering Basics

- **Vertex Shaders**
- **Purpose**: Vertex shaders process individual vertices of a 3D model. They handle the transformation and lighting calculations needed to project the 3D coordinates of vertices onto the 2D screen space.
- **Functionality**:
    - Apply transformations like translation, rotation, and scaling to vertices.
    - Adjust lighting properties based on the vertex position in the scene.
    - Pass per-vertex data (like position, normal, and texture coordinates) to the next stages in the pipeline, often including the fragment shader.
- **Operation Level**: Operate on each vertex in the model's geometry, executing once per vertex.

Rendering Basics

- **Purpose**: Fragment shaders, also known as pixel shaders, operate on fragments, which are potential pixels on the screen. They determine the final color and other attributes of each pixel, including texturing and lighting effects.
- **Functionality**:
  - Calculate the color of each pixel based on textures, lighting, and the material properties.
  - Implement detailed surface effects like glossiness, roughness, and ambient occlusion.
  - Often used for complex visual effects like bump mapping, specular highlights, and shadow computation.
- **Operation Level**: Execute once per fragment, which can be more frequent than per-vertex due to the rasterization process producing multiple fragments for each vertex, especially when rendering detailed or close-up views.

Rendering Basics



```
#version 300 es

// Input vertex data
layout(location = 0) in vec3 vertexPosition_modelspace;

// Output to the fragment shader.
out vec3 fragmentColor;

// Uniform rotation matrix
uniform mat4 rotationMatrix;

void main(){
    // Rotate the vertex position
    vec4 rotatedPosition = rotationMatrix *
vec4(vertexPosition_modelspace, 1.0);

    gl_Position = rotatedPosition;

    // Set the color to blue
    // for simplicity.
    fragmentColor = vec3(0,0,1);
}
```

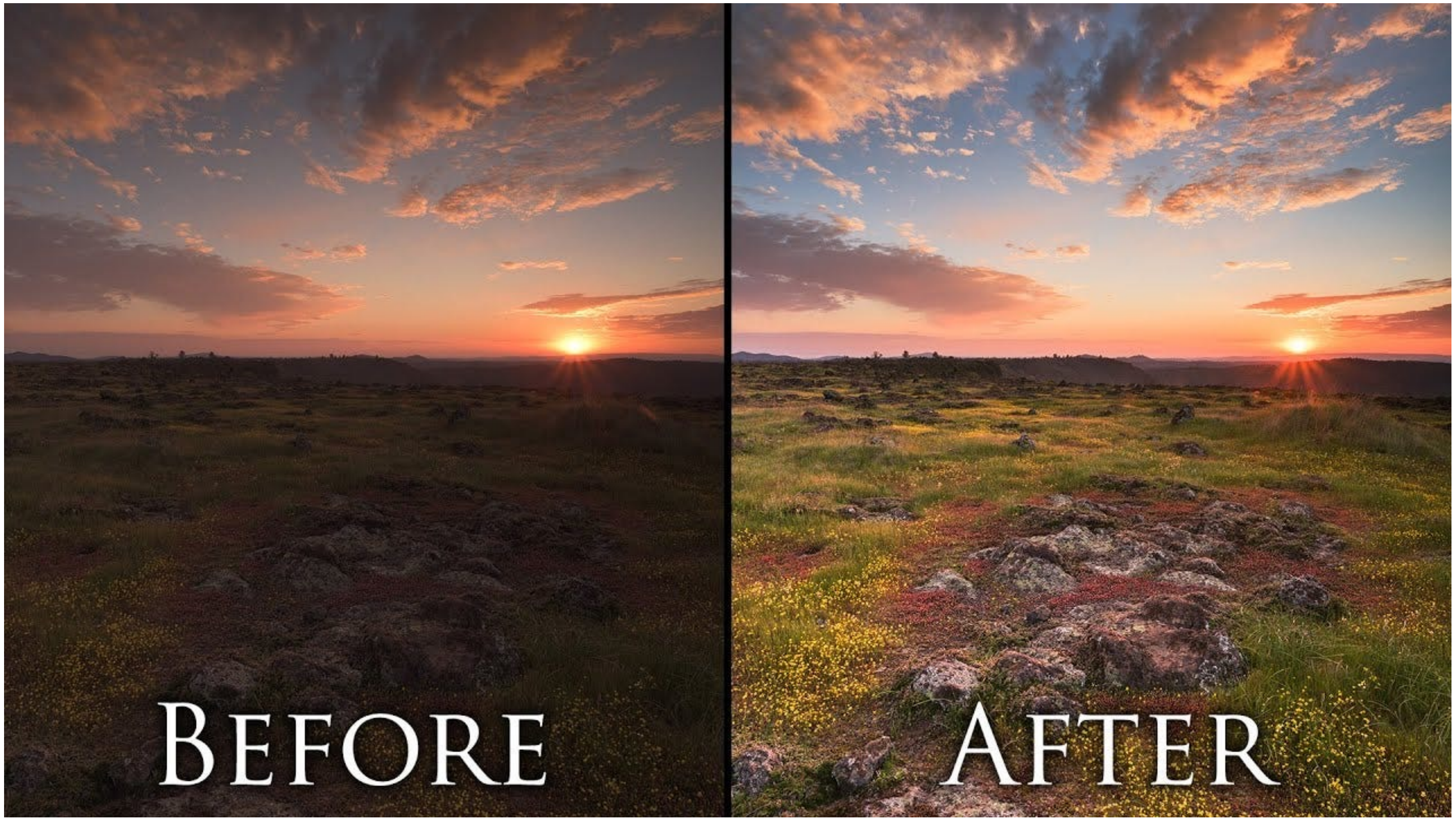
```
#version 300 es

precision mediump float;  // Required for specifying the
                         // precision in the fragment shader

// Input data from the vertex shader
in vec3 fragmentColor;

// Output data
out vec4 color;

void main(){
    // We simply set the color of the pixel.
    color = vec4(fragmentColor, 1.0);
}
```

Rendering Basics

- Post-processing effects: Techniques applied to the rendered image before final output to enhance visual quality or achieve specific artistic styles.
- Common Effects:
  - Bloom: Simulates light bleeding around bright areas.
  - Motion Blur: Blurs objects based on movement, adding realism or speed sensation.
  - Depth of Field: Blurs parts of the scene not in focus, mimicking camera lens effects.
  - Color Grading: Adjusts the color palette to convey mood or time of day.

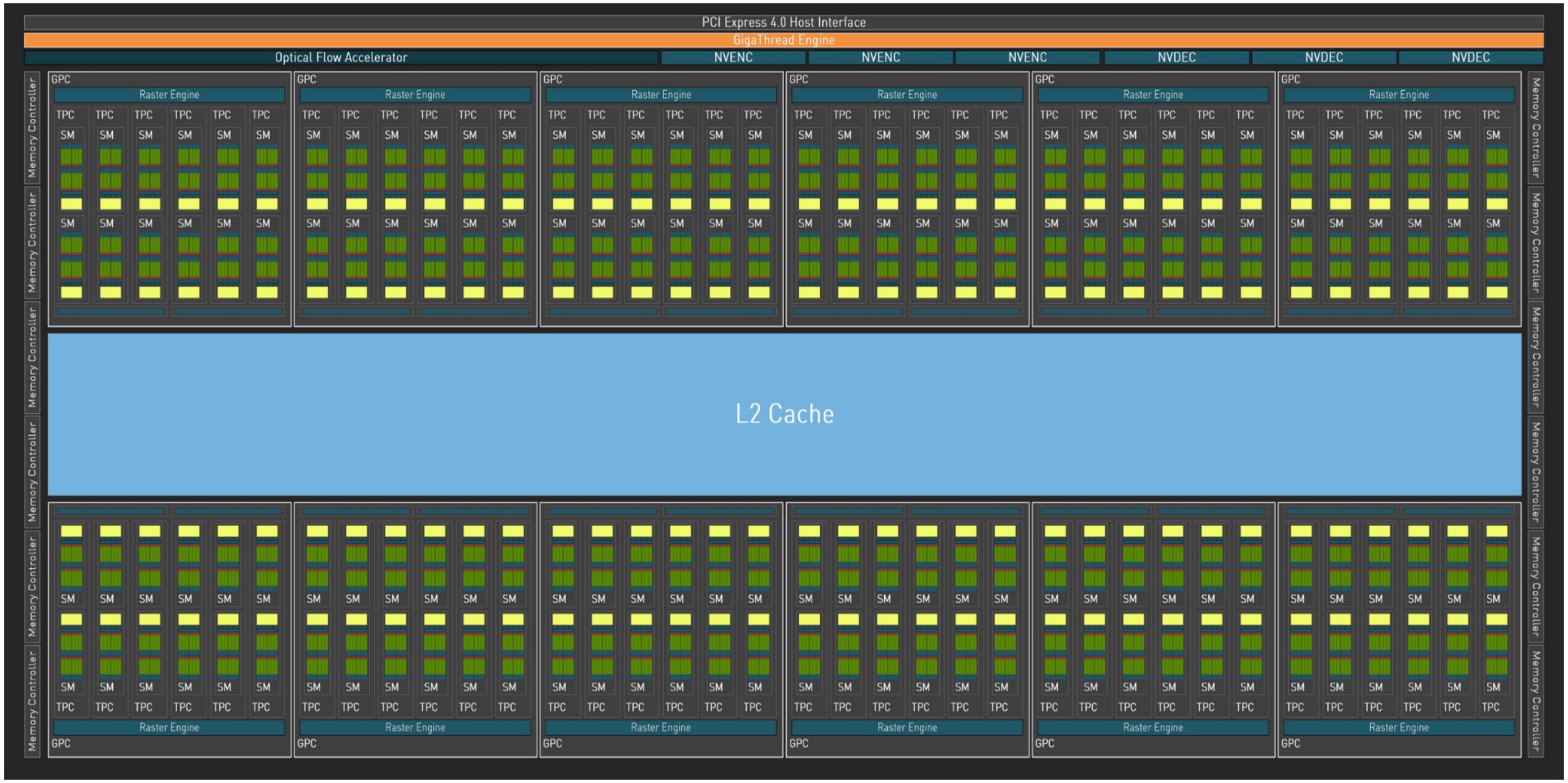
Rendering Basics

- Post-processing effects



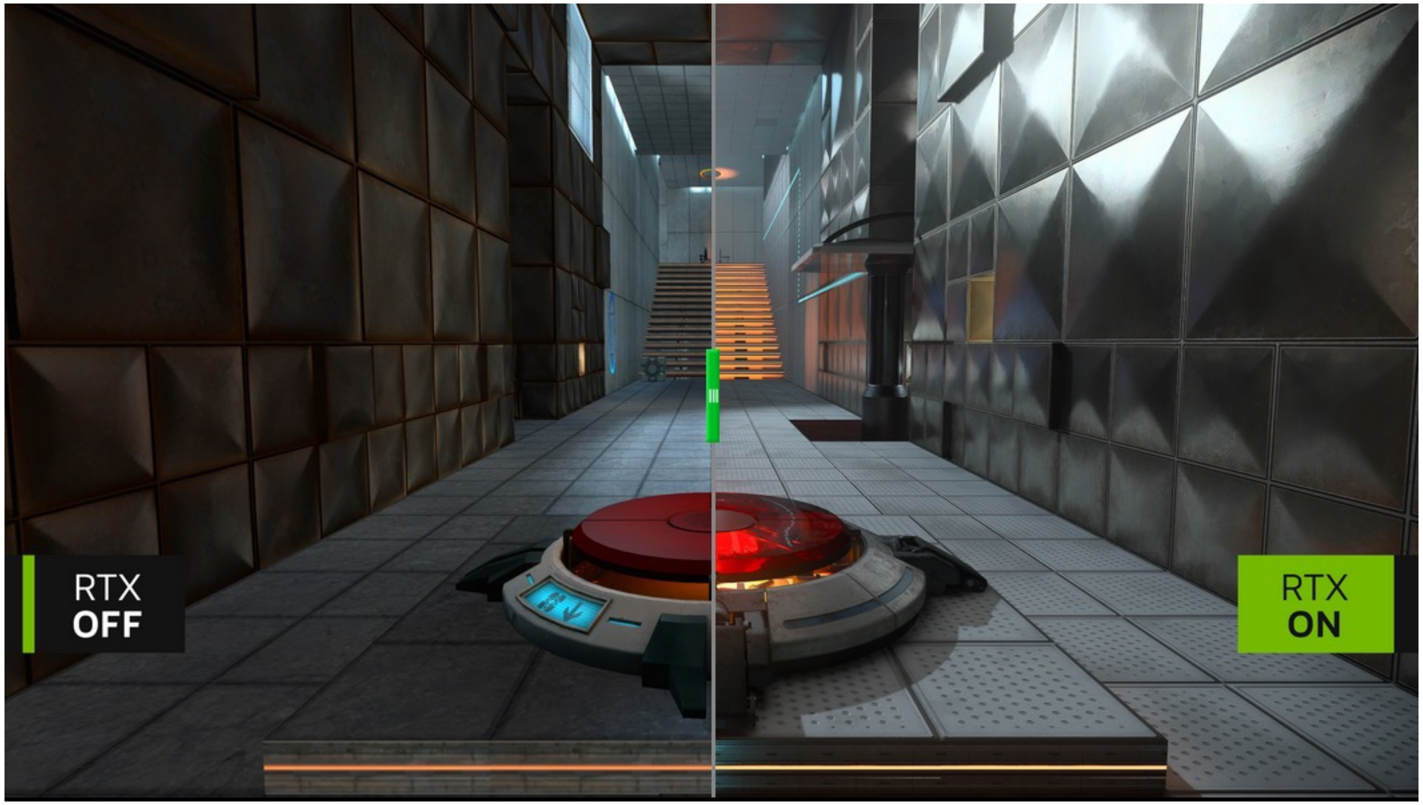
Real-time Rendering

- RTX 4090



Real-time Rendering

- RTX 4090



Real-time Rendering

- RTX 4090

RTX ON
RTX OFF

Summary of the Lecture

- Rendering Basics
    - Types of rendering
    - Rendering pipeline
    - Real-time rendering

Next up: Hybrid Rendering