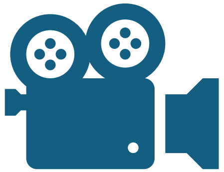
Graphics Pipeline

CSU44052 Computer Graphics

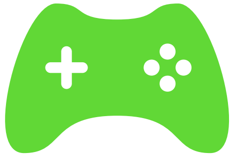
Binh-Son Hua

Graphics Applications



Movies,
Visual effects

Interior designs,
Architectural
visualization



Video games

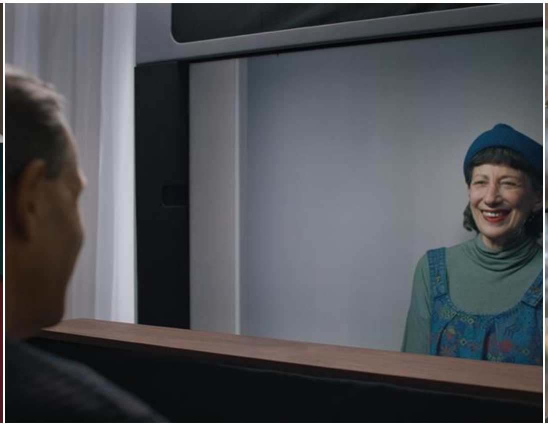
Metaverse,
Telepresence

Artificial
intelligence

Avatar
[2009]

Blender
[2012]

MetaHuman
[2021]

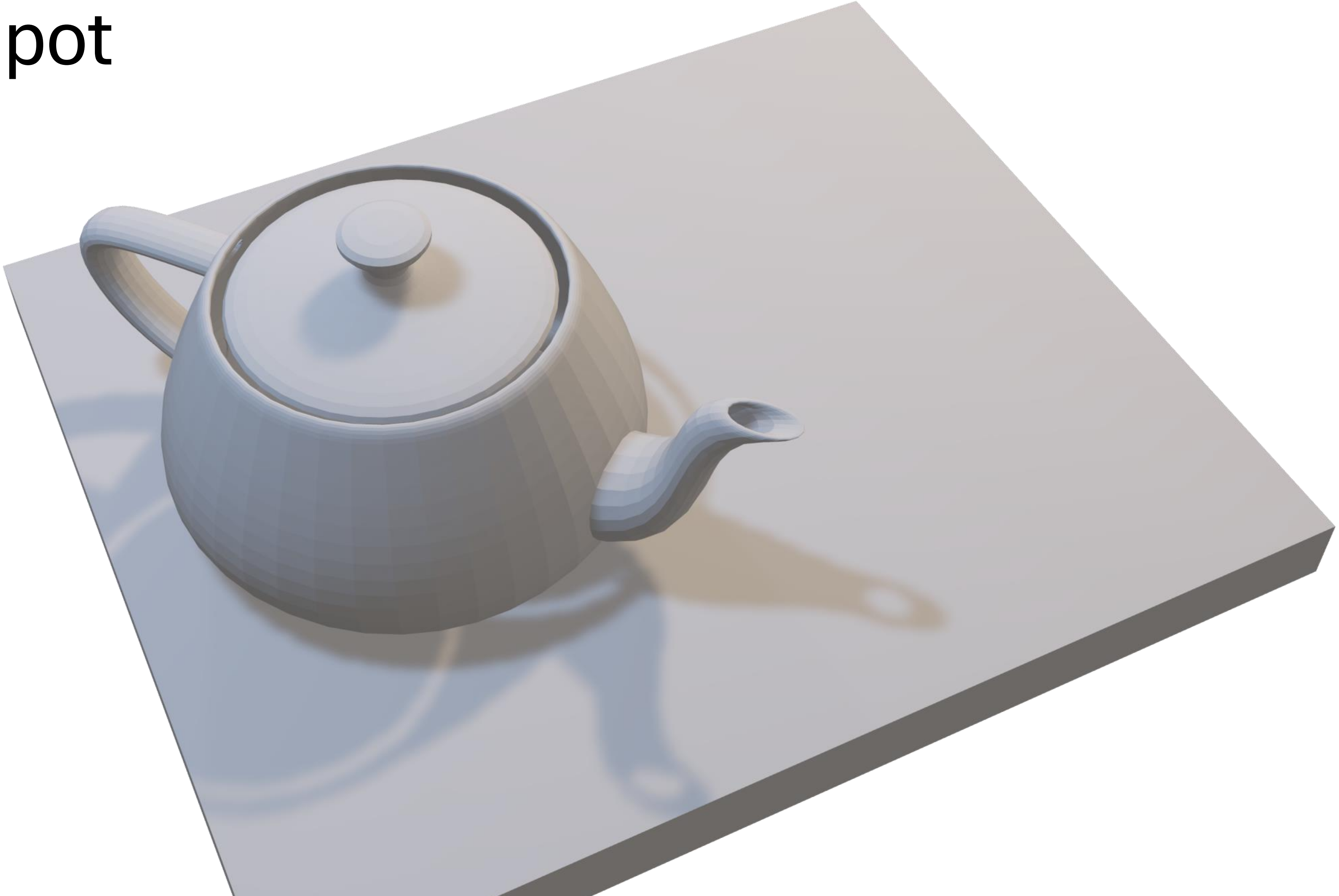


Project Starline
[2021]

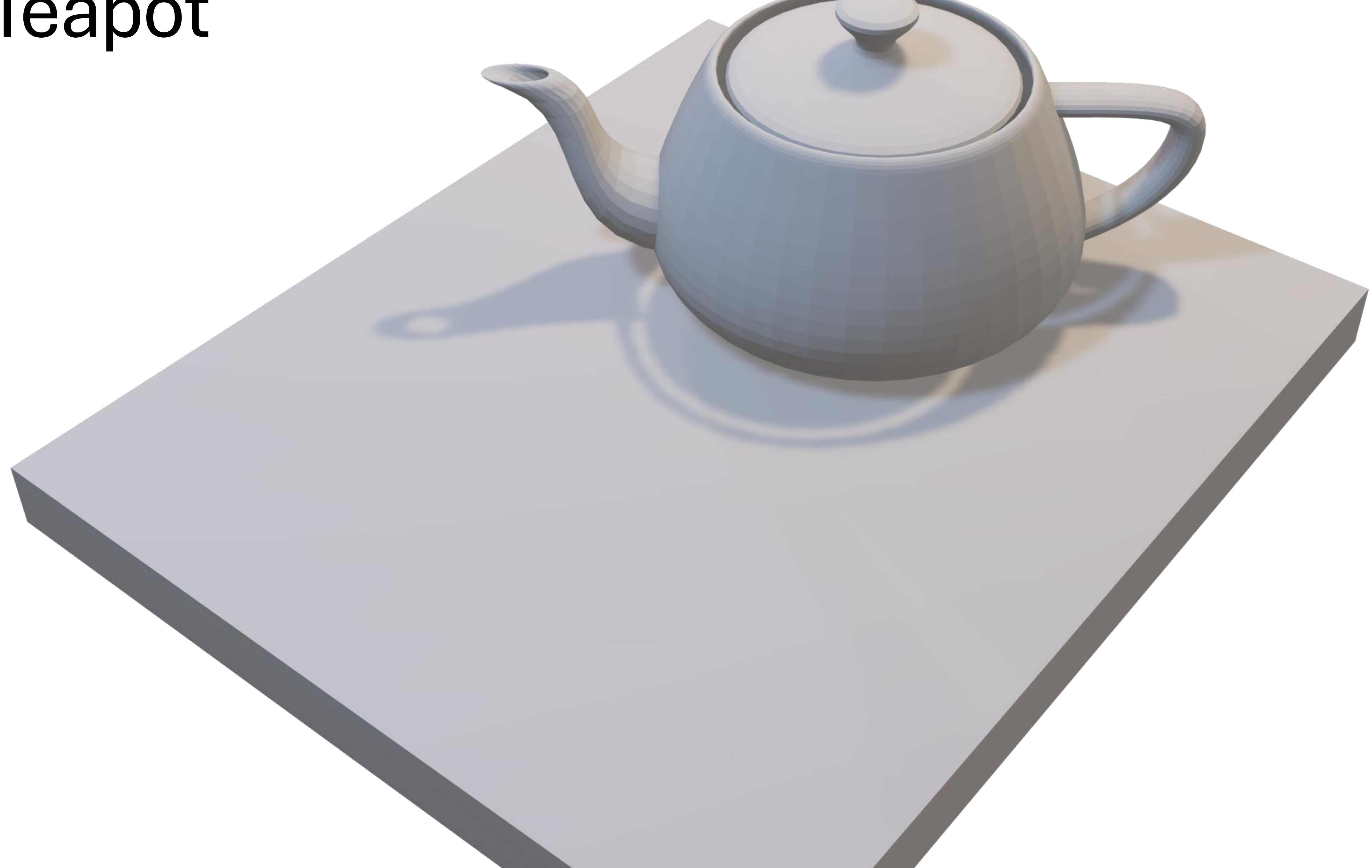
Face Synthetics
[2021]

Teapot

Teapot

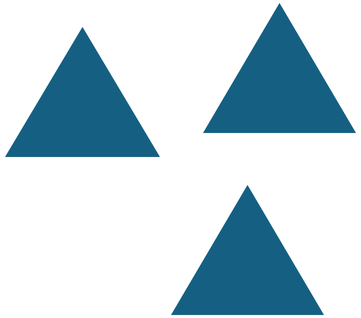


Teapot



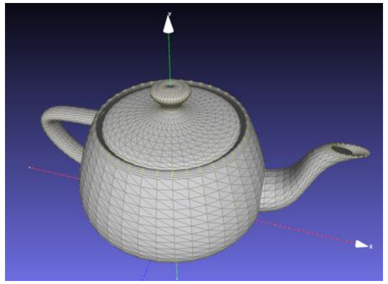
Rendering

- *Rendering* is the process by which a computer creates images from models or objects.
- The final rendered image consists of pixels drawn on the screen



Graphics Pipeline

- Geometry
- Material
- Textures
- Light emitters
- Cameras

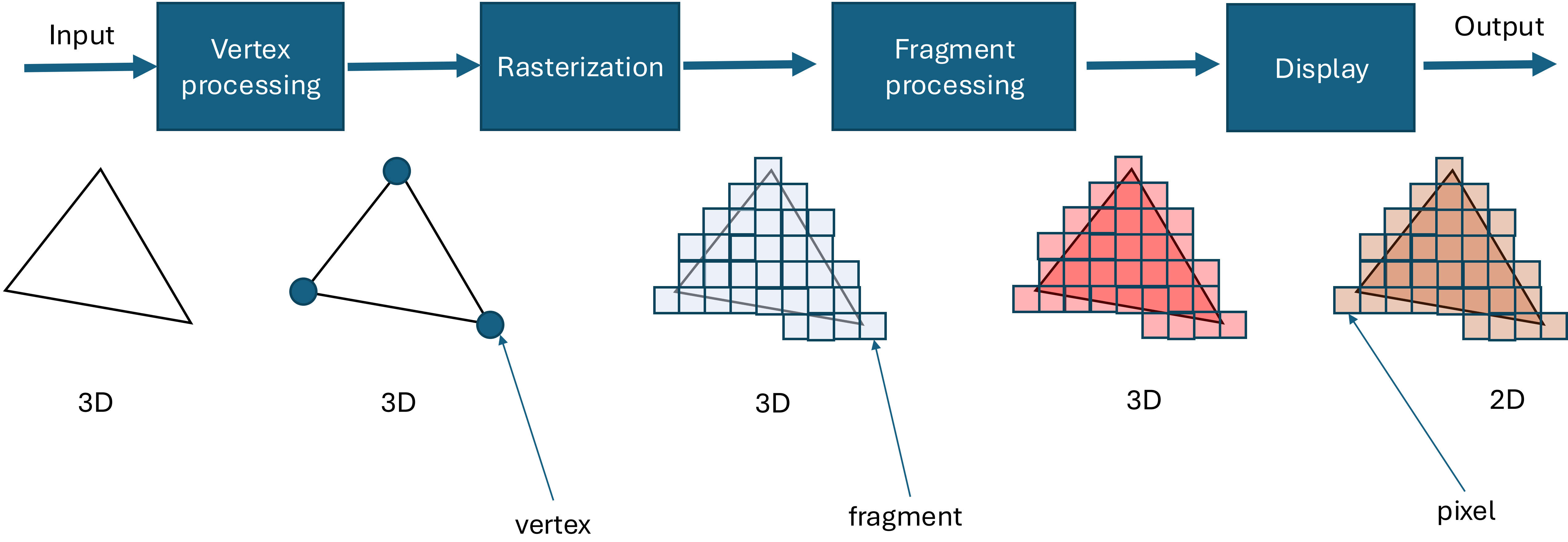


- Image
- Image sequence

Objectives

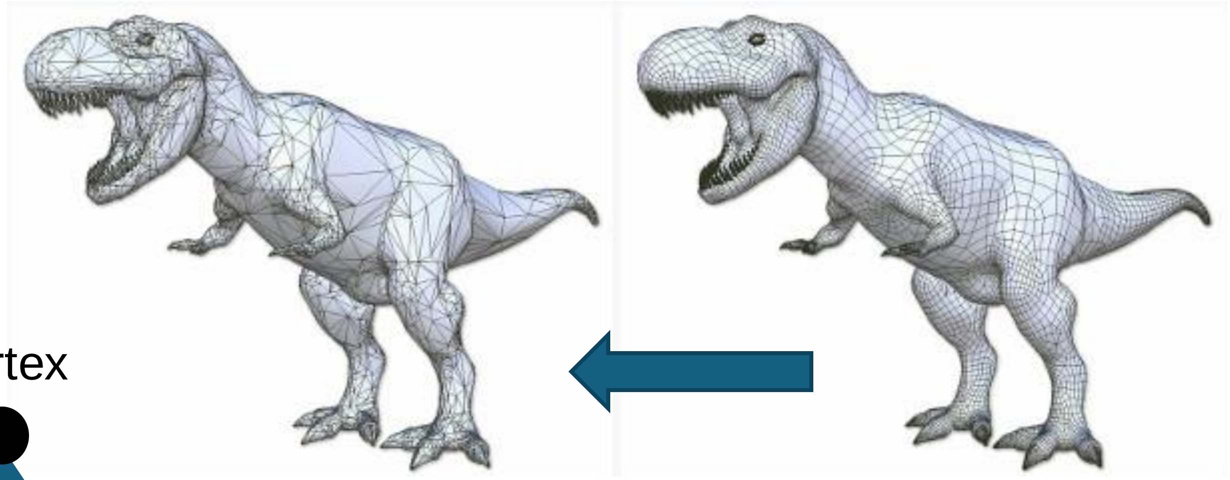
- Introduction to industry-standard rendering pipeline
- Vertex processing
- Fragment processing

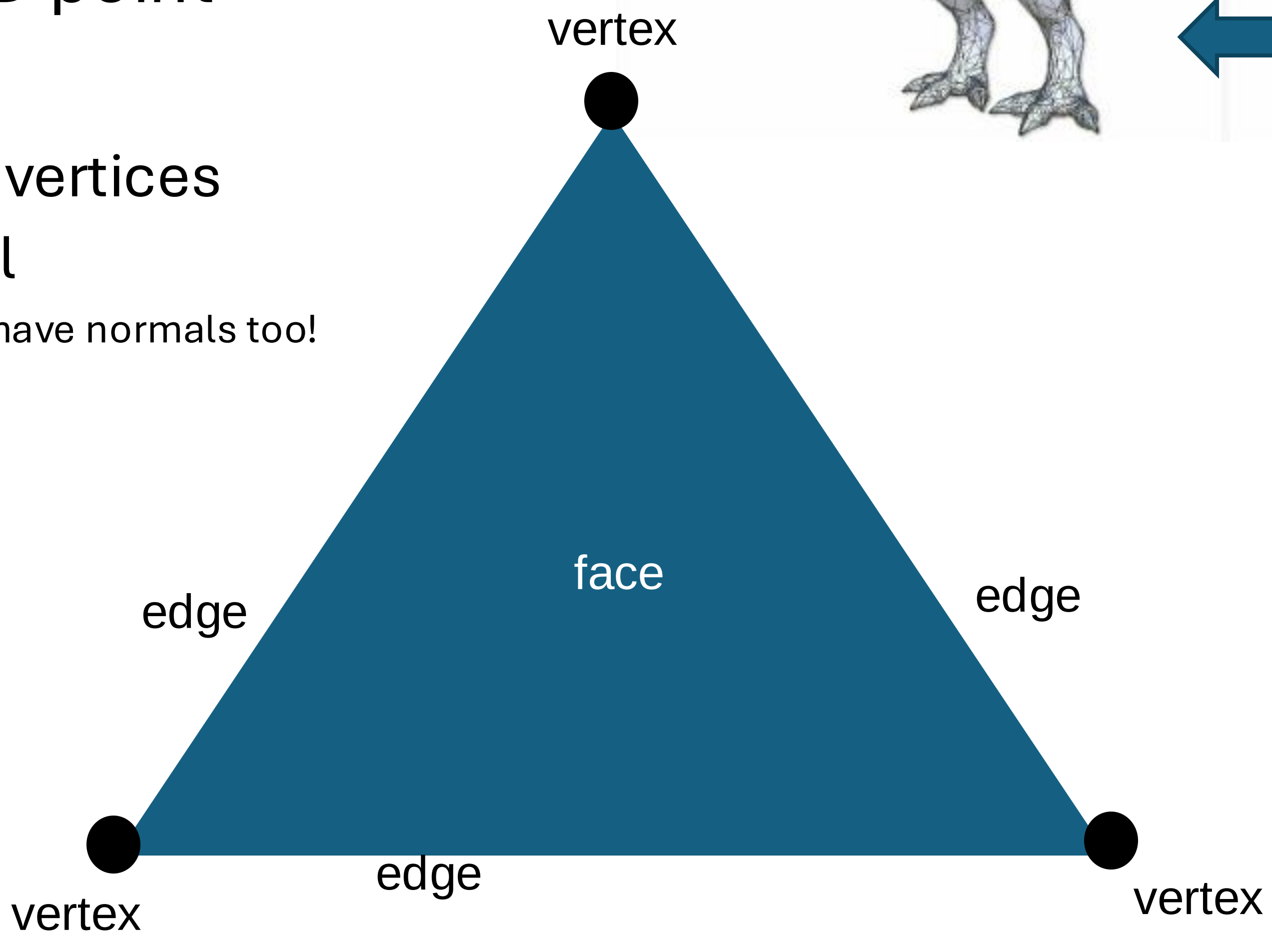
Rendering Pipeline

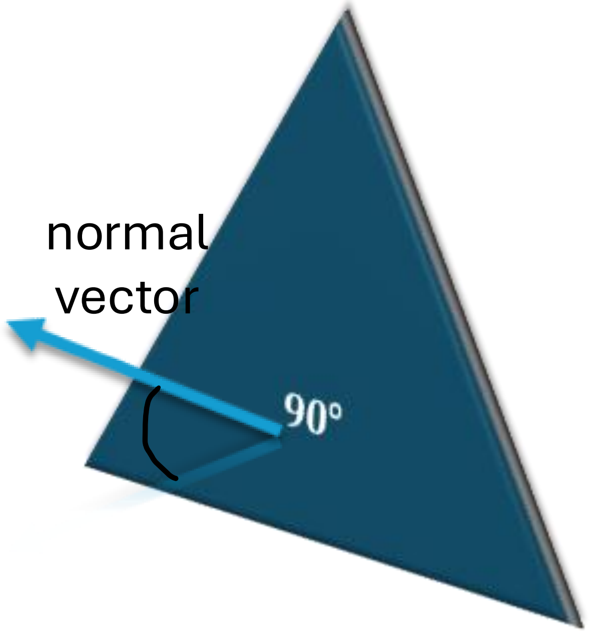


Triangle

- A vertex is a 3D point
- A triangle:
    - Made from 3 vertices
    - Has a normal
    - Note: vertices can have normals too!

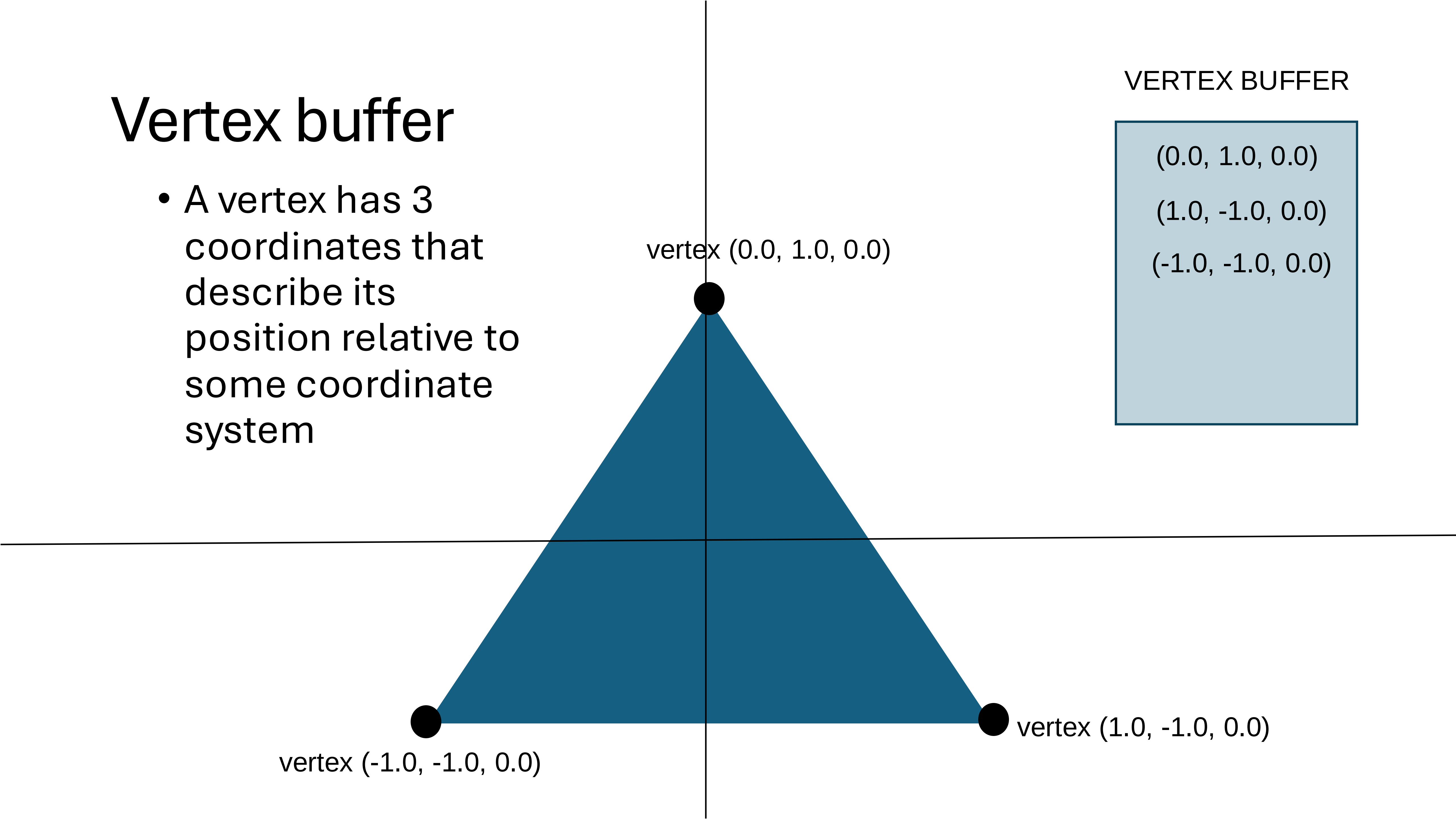




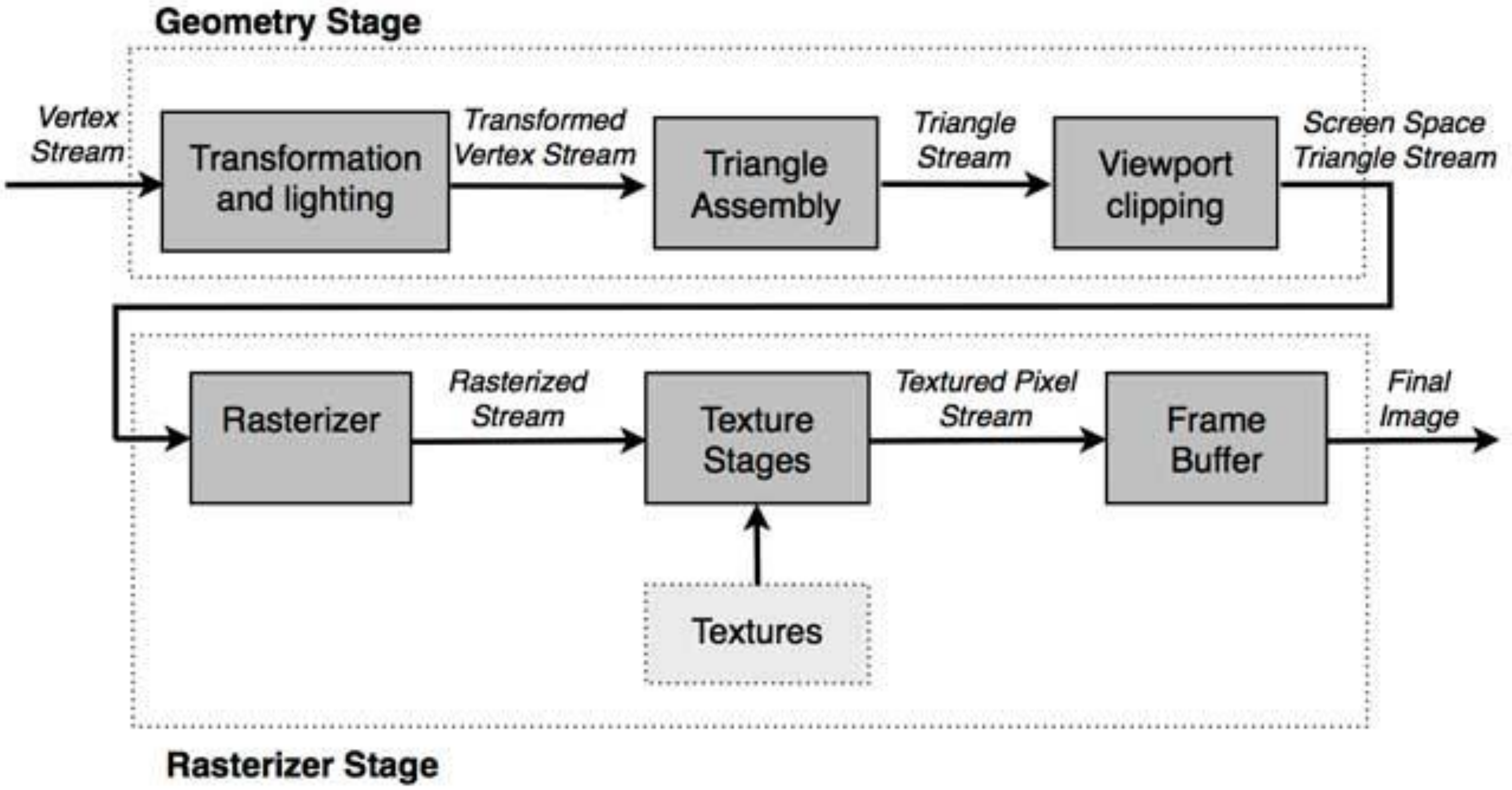


Vertex buffer

- A vertex has 3 coordinates that describe its position relative to some coordinate system

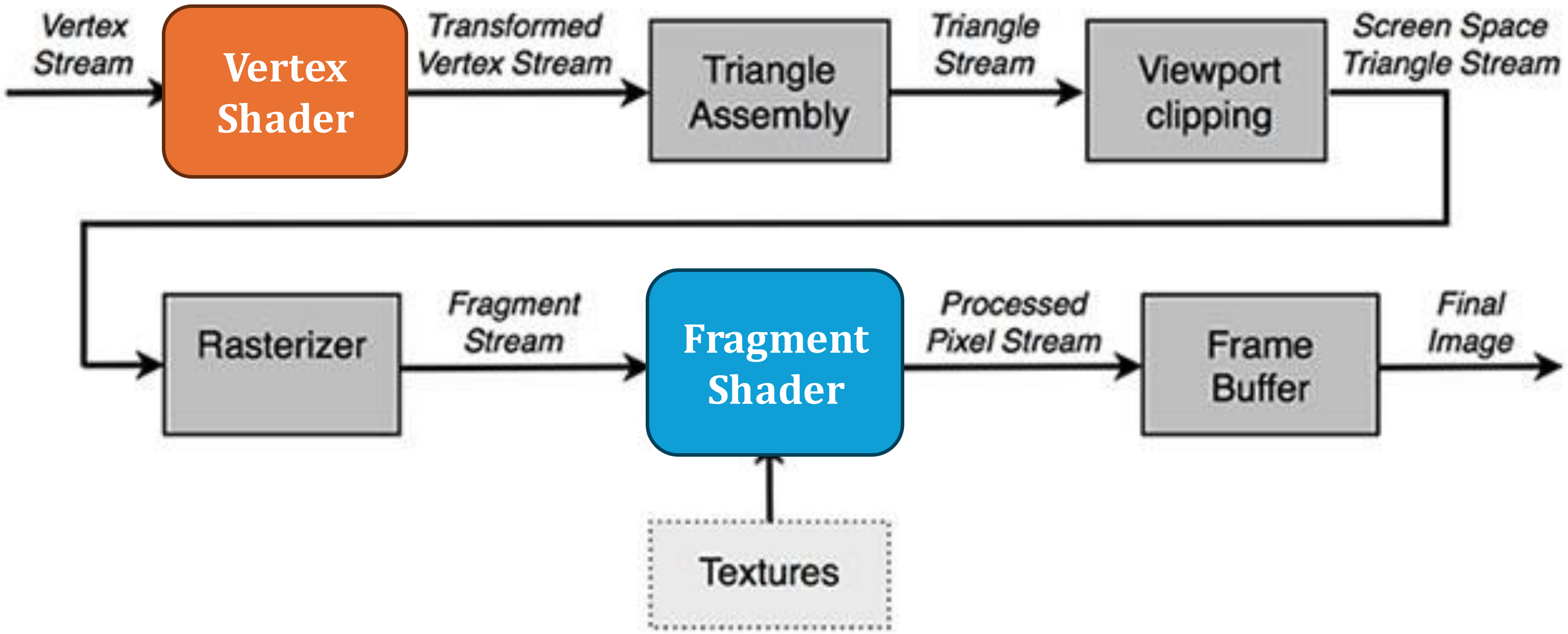


Fixed Function Pipeline



http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html

Programmable Pipeline



What are Shaders?

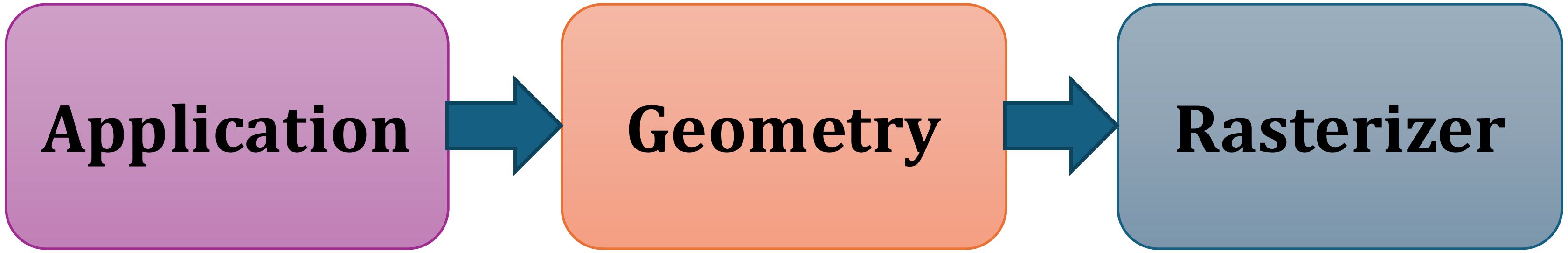
- A small program that runs on the graphics processor (GPU).
- Contribute to some stages of the graphics pipeline.
- Different types of shaders: **vertex shader**, tessellation shader geometry shader, **fragment shader**, compute shader.
- Shader is data parallel, i.e., it runs in parallel on every vertex or pixel.

Shaders in OpenGL

- Shaders in OpenGL are written in GLSL, a programming language similar to C
- Vector data types: vec4, mat4
- Uniform variables, e.g., camera matrix
- Varying variables, e.g., vertex positions, pixel colors
- Built-in variables, e.g., gl_Position, gl_FragColor
- User-defined variables

Graphics Pipeline Overview

- Can be viewed as having three stages
- Each stage is a pipeline in itself



- The slowest pipeline stage determines the *rendering speed (fps)*

Real-Time Rendering, 3rd Edition, Akenine-Moller

The Application Stage

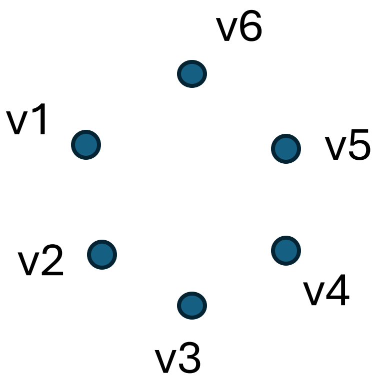
VERTEX BUFFER

-3.3804130	-1.1272367	0.5733036
0.9668296	-1.0737425	-0.8198227
0.0567293	0.8527195	0.3923156
-1.3751742	-1.0212243	-0.0570552
-1.2615018	0.2590713	0.5234135
-0.3068337	-1.6836331	-0.7169344
1.1394235	0.1874122	-0.2700900
0.5602627	2.0839095	0.8251589
-0.4926797	-2.8180554	-1.2094732
-2.6328073	-1.7303959	-0.0060953
-2.2301338	0.7988624	1.0899730
2.5496990	2.9734977	0.6229590
2.0527432	-1.7360887	-1.4931279
-2.4807715	-2.7269528	0.4882631
-3.0089039	-1.9025254	-1.0498023
2.9176101	-1.8481516	-0.7857866
2.3787863	-1.1211917	-2.3743655
1.7189877	-2.7489920	-1.8439205
-0.1518450	3.0970046	1.5348347
1.8934096	2.1181245	0.4193193
2.2861252	0.9968439	-0.2440298
-0.1687028	4.0436553	0.9301094
0.3535322	3.2979060	2.5177747
-1.2074498	2.7537592	1.7203047

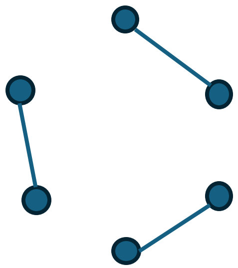
Application

- Developer has full control
- Executes on the CPU
- At the end of the application stage, the rendering primitives are fed to the geometry stage

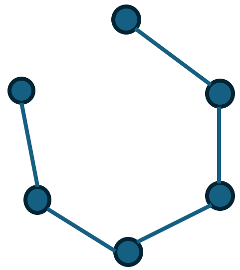
Drawing Primitives



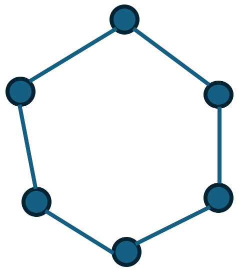
Points



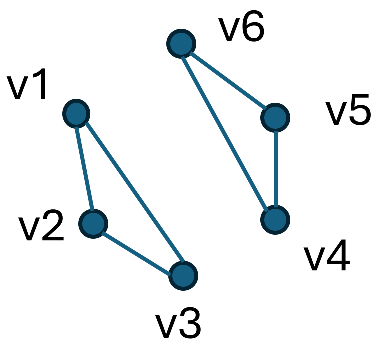
Lines



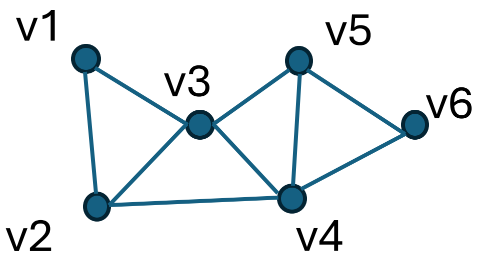
Line strips



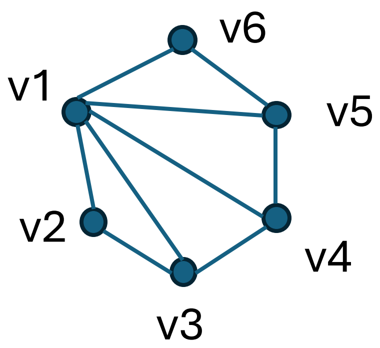
Line loops



Triangles



Triangle strips

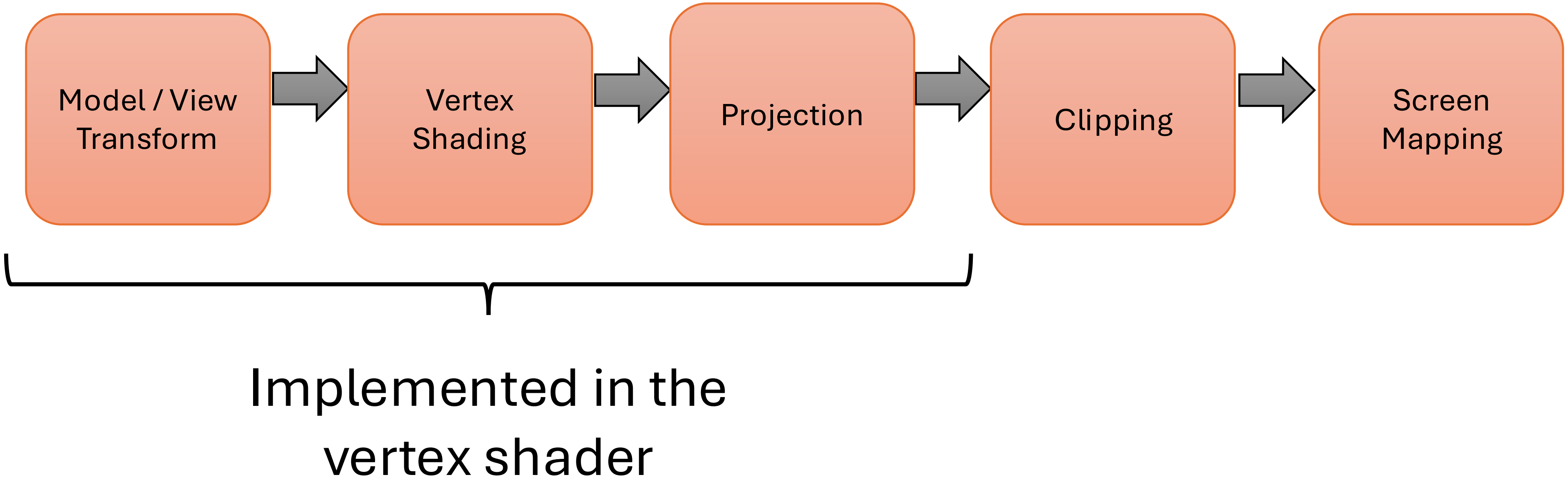


Triangle fans

glDrawArrays
glDrawElements

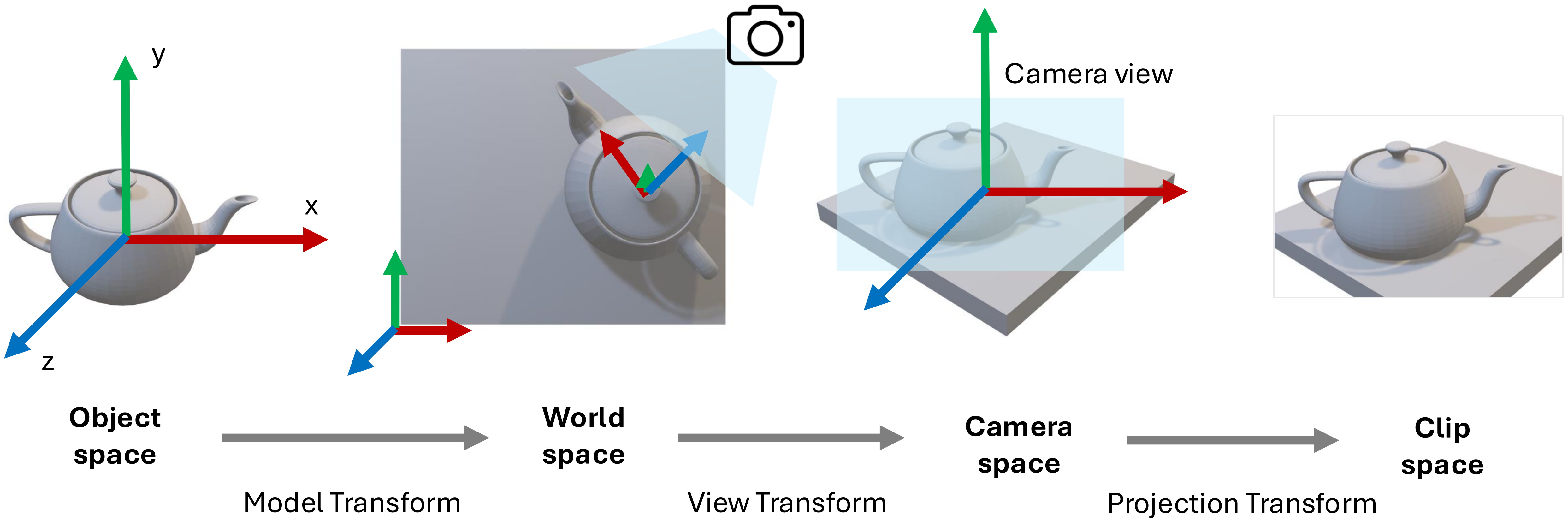
The Geometry Stage

- Responsible for the per-polygon and per-vertex operations



Model / View Transform

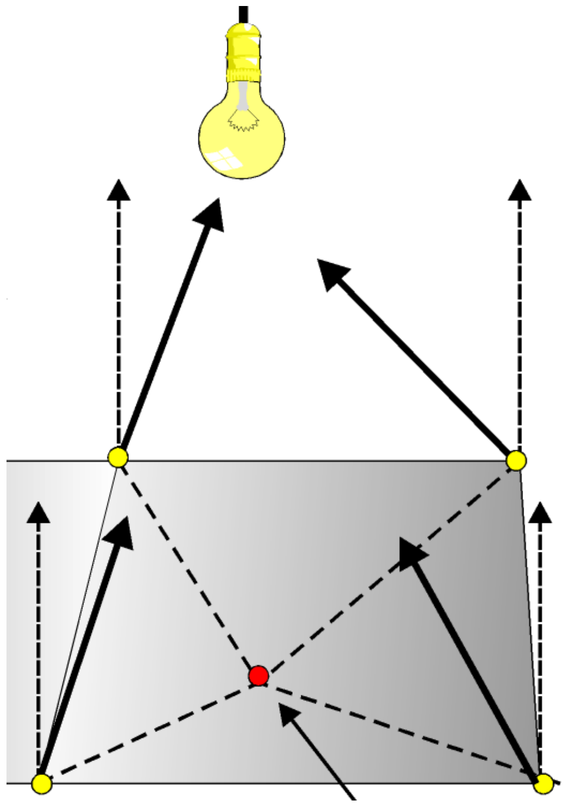
Model & View Transform



Vertex Shading

Vertex Shading

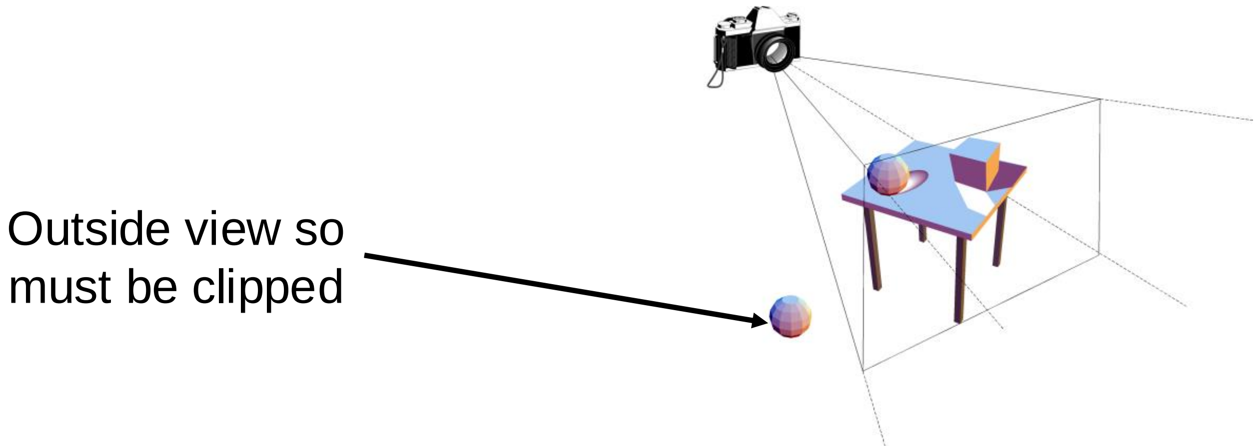
- Shading means determining the effect of a light on a material
- A variety of material data can be stored at each vertex
  - Points location
  - Normal
  - Color
- Vertex shading results (colors, vectors, texture coordinates, or any other kind of shading data) are then sent to the **rasterization** stage to be **interpolated**



Clipping

Clipping

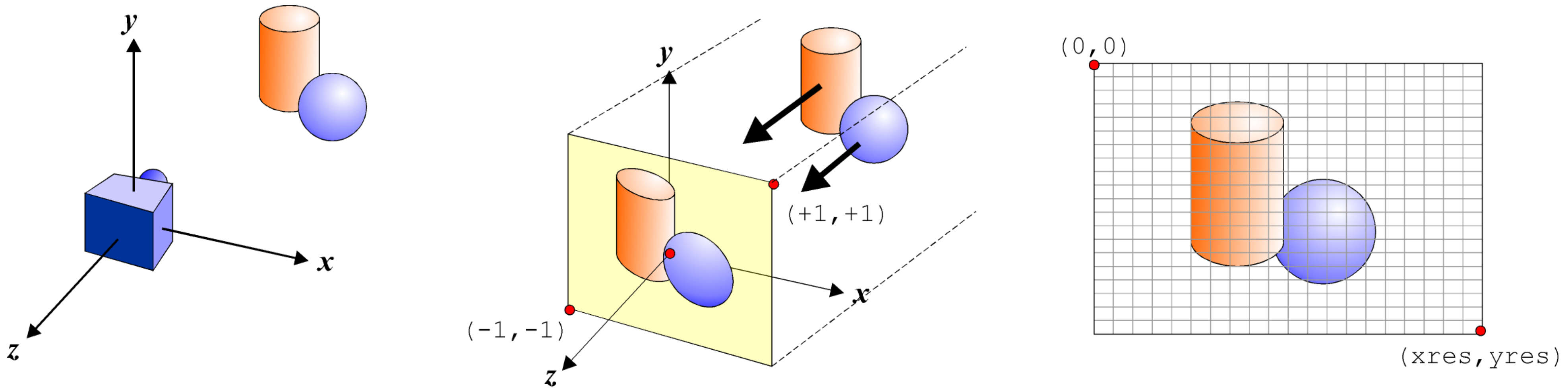
- The computer may have model, texture, and shader data for all objects in the scene in memory
- The virtual camera viewing the scene only "sees" the objects within the field of view
- The computer does not need to transform, texture, and shade the objects that are behind or on the sides of the camera
- A clipping algorithm skips these objects making rendering more efficient



Screen Mapping

Screen Mapping

- Only the clipped primitives inside the view volume are passed to this stage
- Coordinates are in 3D
- The x- and y-coordinates of each primitive are transformed to the *screen coordinates*



Vertex Shader Example

```
#version 330 core

// Input vertex attributes
layout(location = 0) in vec3 vertexPosition;

void main() {
  // Default vertex position to be passed to next step in the pipeline
  gl_Position.xyz = vertexPosition;
  gl_Position.w = 1.0;
}
```

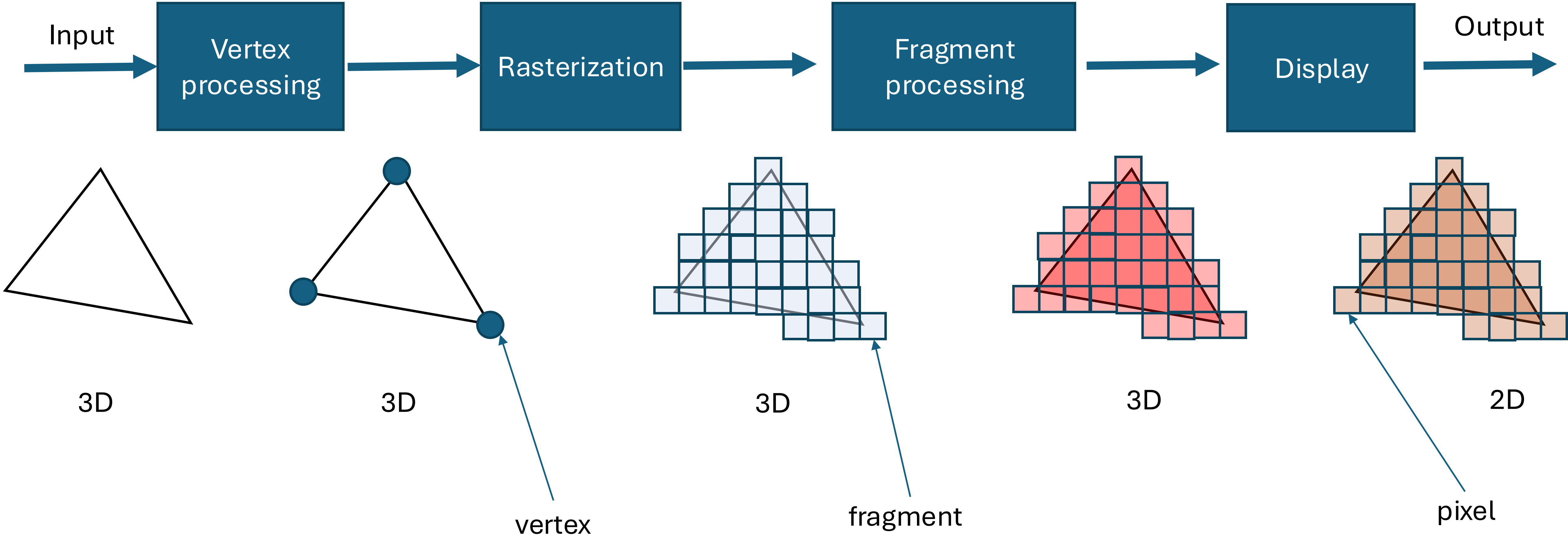
Vertex Shader Example

```
#version 330 core

// Input vertex attributes
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColor;

// Output data, to be interpolated for each fragment
out vec3 color;

void main() {
  gl_Position.xyz = vertexPosition;
  gl_Position.w = 1.0;

  // Pass vertex color to the fragment shader
  color = vertexColor;
}
```

Vertex Shader Example

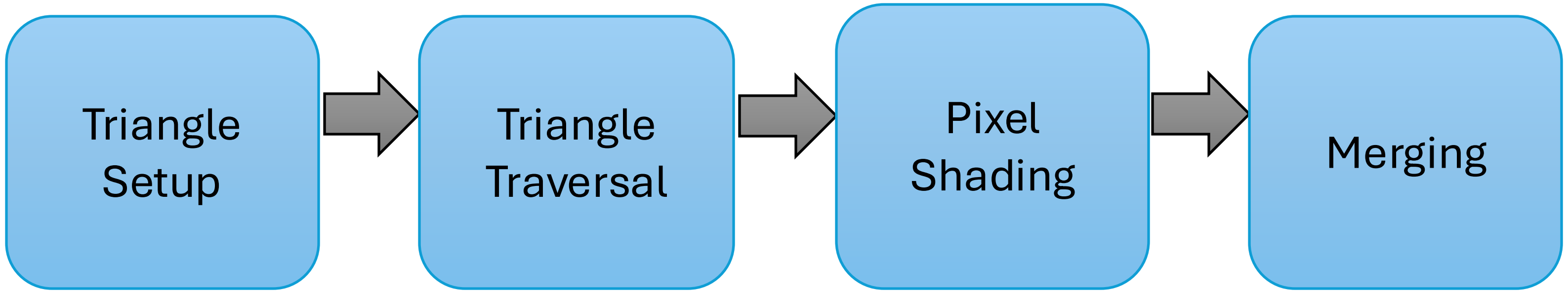
```
#version 330 core

// Input
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColor;

// Matrix for vertex transformation
uniform mat4 MVP;

// Output data, to be interpolated for each fragment
out vec3 color;

void main() {
  // Transform vertex
  gl_Position = MVP * vec4(vertexPosition, 1);
  // Pass vertex color to the fragment shader
  color = vertexColor;
}
```

Rendering Pipeline



The Rasterizer Stage

- Given the transformed and projected vertices with their associated shading data (from geometry stage)



- The goal of the rasterizer stage is to compute and set colors for the pixels covered by the object

Triangle Setup

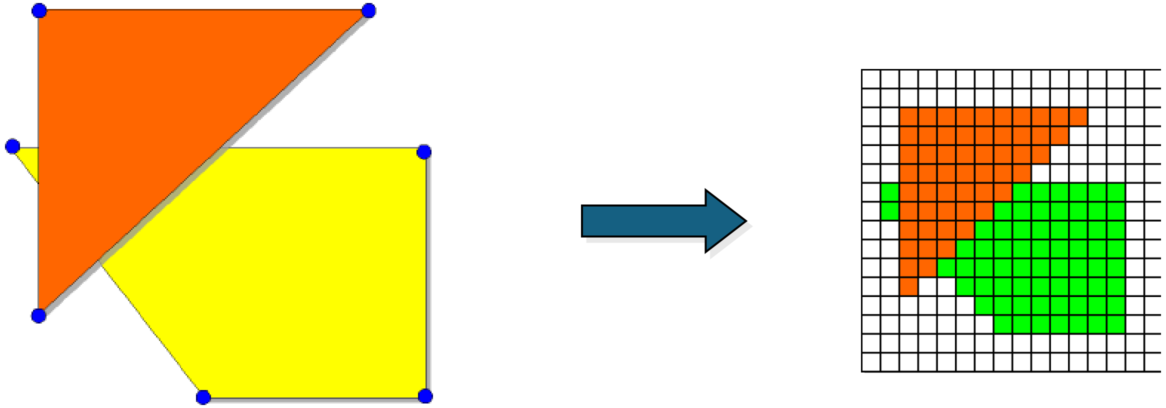
Triangle Setup

- Vertices are collected and converted into triangles.
- Information is generated that will allow later stages to accurately generate the attributes of every pixel associated with the triangle.

Triangle Traversal

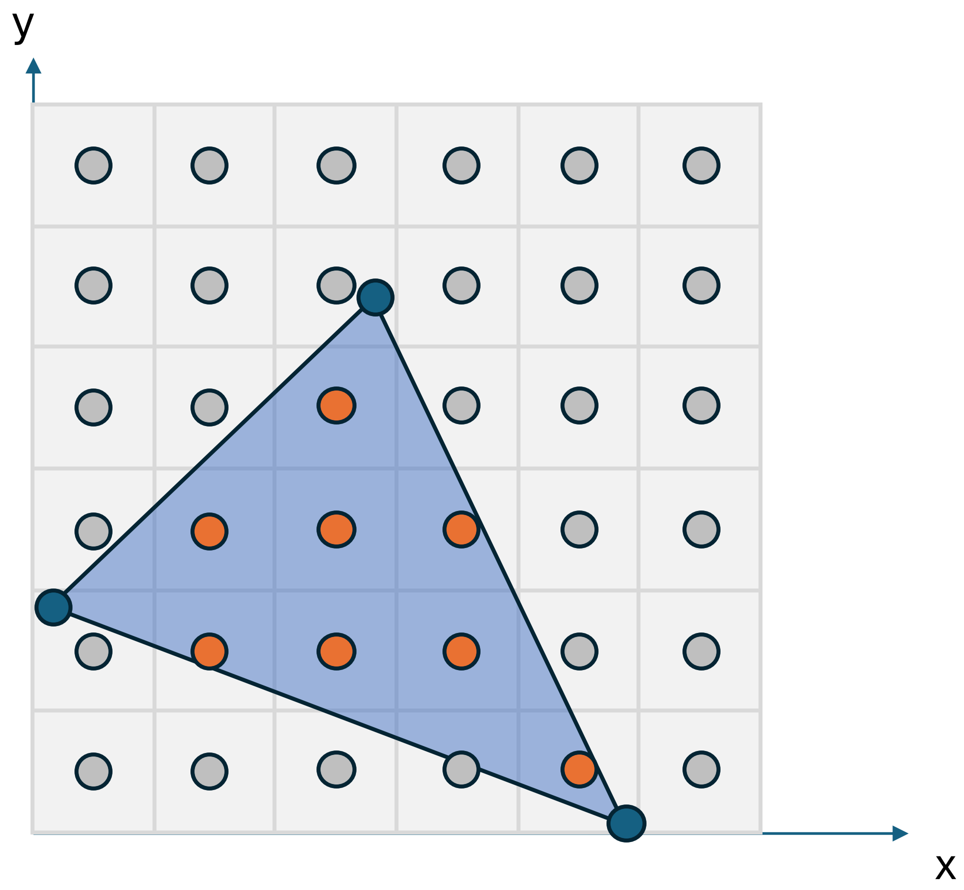
Triangle Traversal

- Which pixels are inside a triangle?
- Each pixel that has its centre covered by the triangle is checked
- A *fragment* is generated for the part of the pixel that overlaps the triangle
- Triangle vertices interpolation



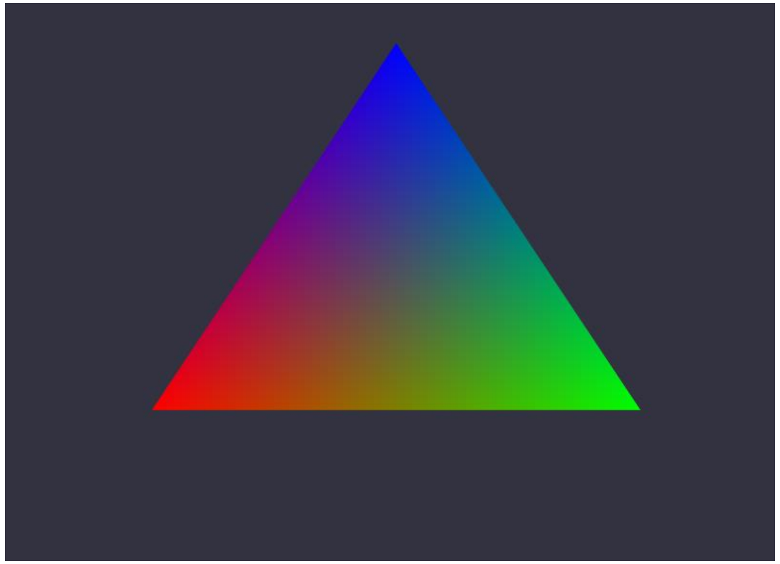
Rasterization

- Sampling of primitives (e.g., triangles) into screen-space pixels.
- Coverage test: mark covered if a pixel center falls into the triangle.
- Implementation: A scanline moving from top to bottom to determine which pixel falls inside the triangle.
- Alternative: Edge function and barycentric interpolation.



Rasterization

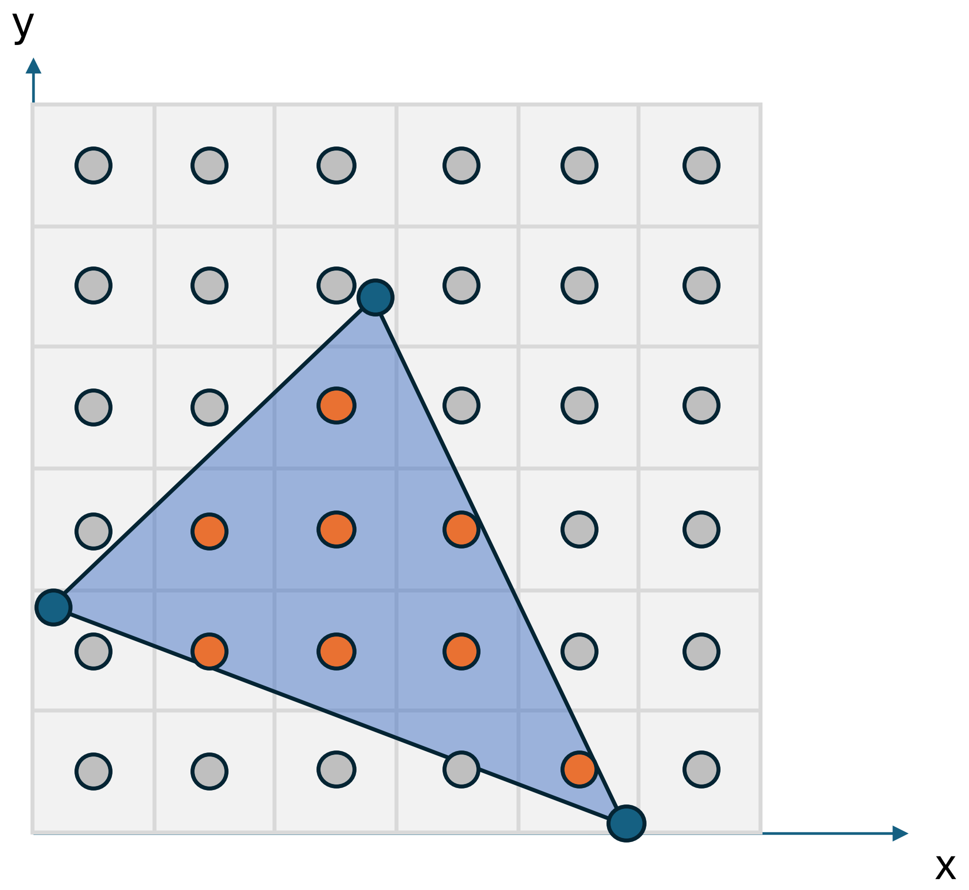
- Rasterization dices a triangle to many fragments
- Attributes on each vertex are interpolated to each fragment
- All fragments can be processed in parallel



Interpolation

- **Input**: position and attributes of each vertex of the triangle

- **Output**: attributes at each fragment in the triangle.



Barycentric Coordinates

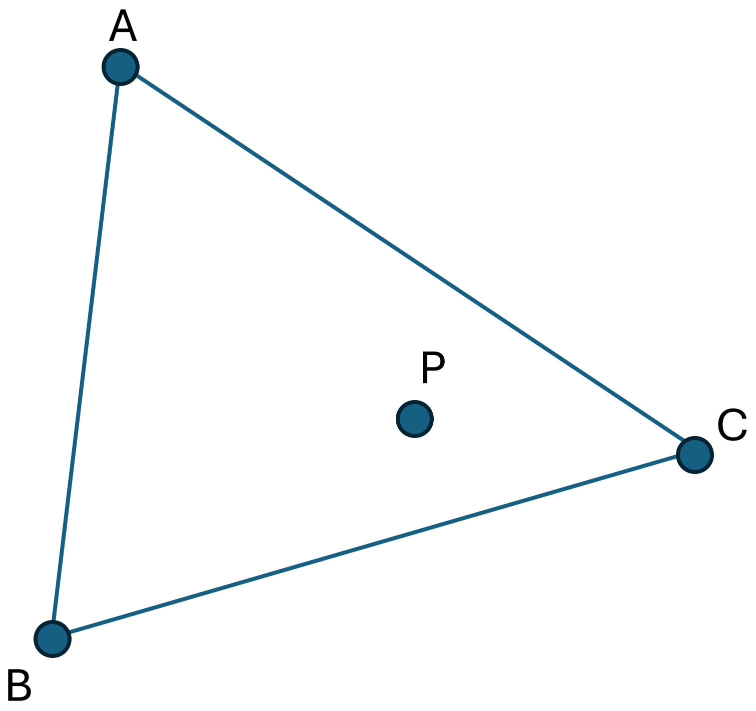
- Given a triangle with 3 vertices A, B, C and a point P in the plane

- The barycentric coordinates of P is (a, b, c) that

$$P = aA + bB + cC$$

with the constraint

$$a + b + c = 1$$

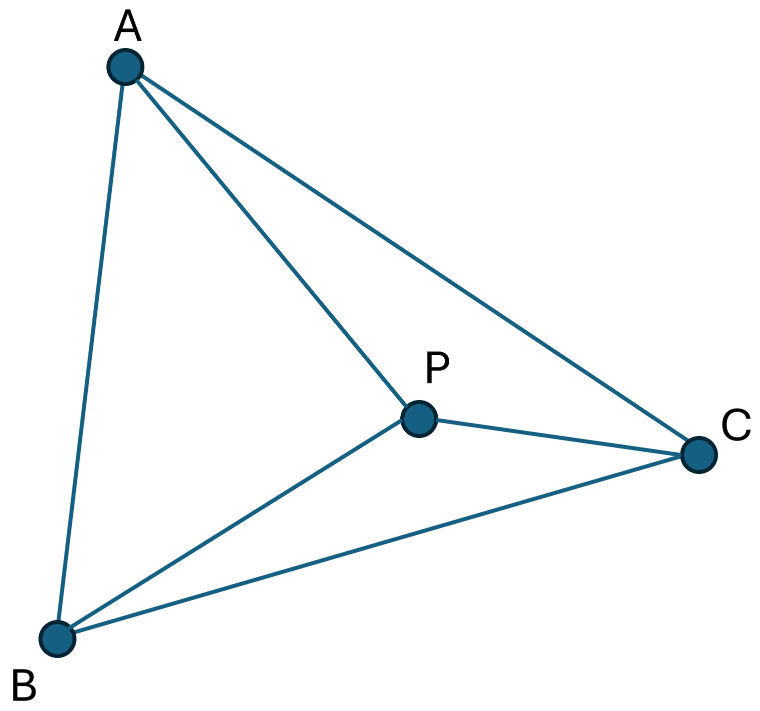


Barycentric Coordinates

- Given a triangle with 3 vertices A, B, C and a point P in the plane
- The barycentric coordinates (a, b, c) of P is

$$a = \frac{area(PBC)}{area(ABC)} \qquad b = \frac{area(PCA)}{area(ABC)}$$

$$c = \frac{area(PAB)}{area(ABC)}$$



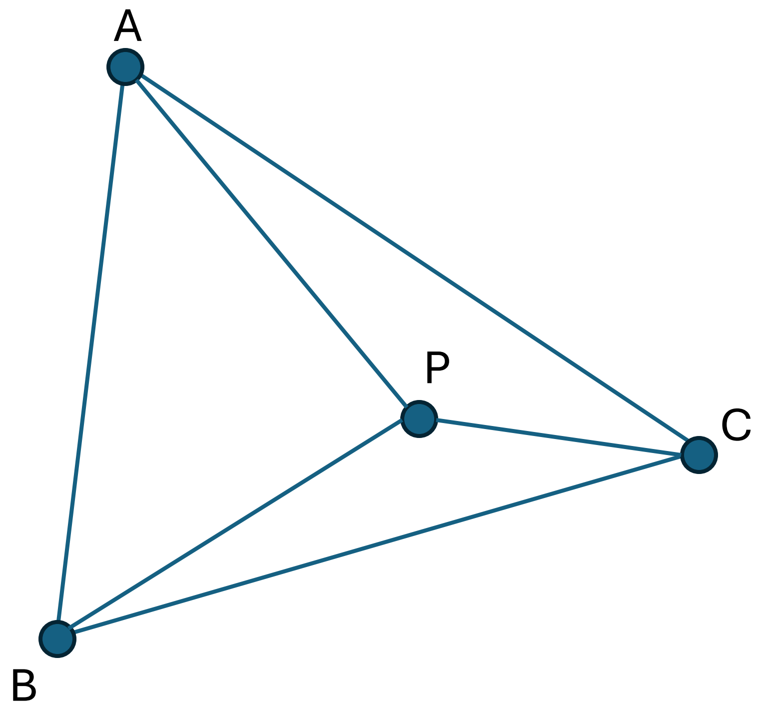
Barycentric Coordinates

- Given a triangle with 3 vertices A, B, C and a point P in the plane
- The barycentric coordinates (a, b, c) of P is

$$a = \frac{area(PBC)}{area(ABC)} \qquad b = \frac{area(PCA)}{area(ABC)}$$

$$c = \frac{area(PAB)}{area(ABC)}$$

$$area(A, B, C) = \frac{1}{2}\overrightarrow{AB} \times \overrightarrow{AC}$$



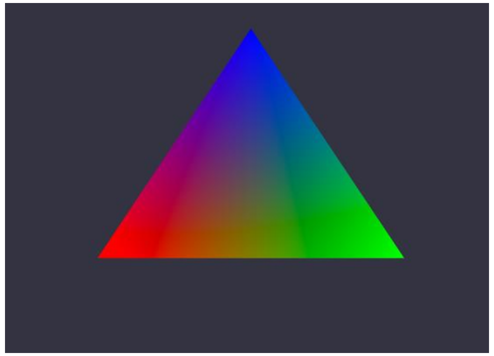
Coefficient a, b, c can be positive or negative. Area is in fact signed area.

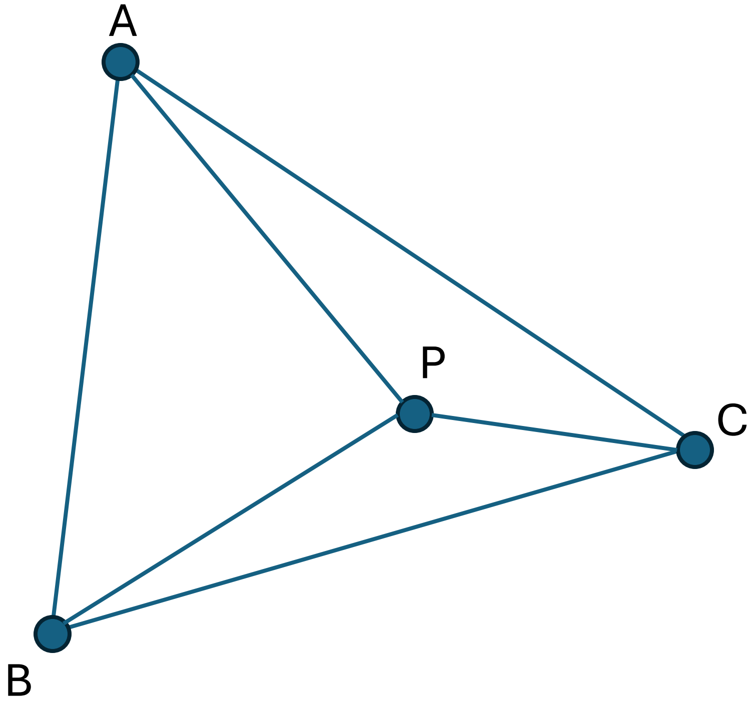
Barycentric Coordinates

- Apply barycentric coordinates (a, b, c) of P for interpolation of vertex attributes!

- Example: interpolate colors

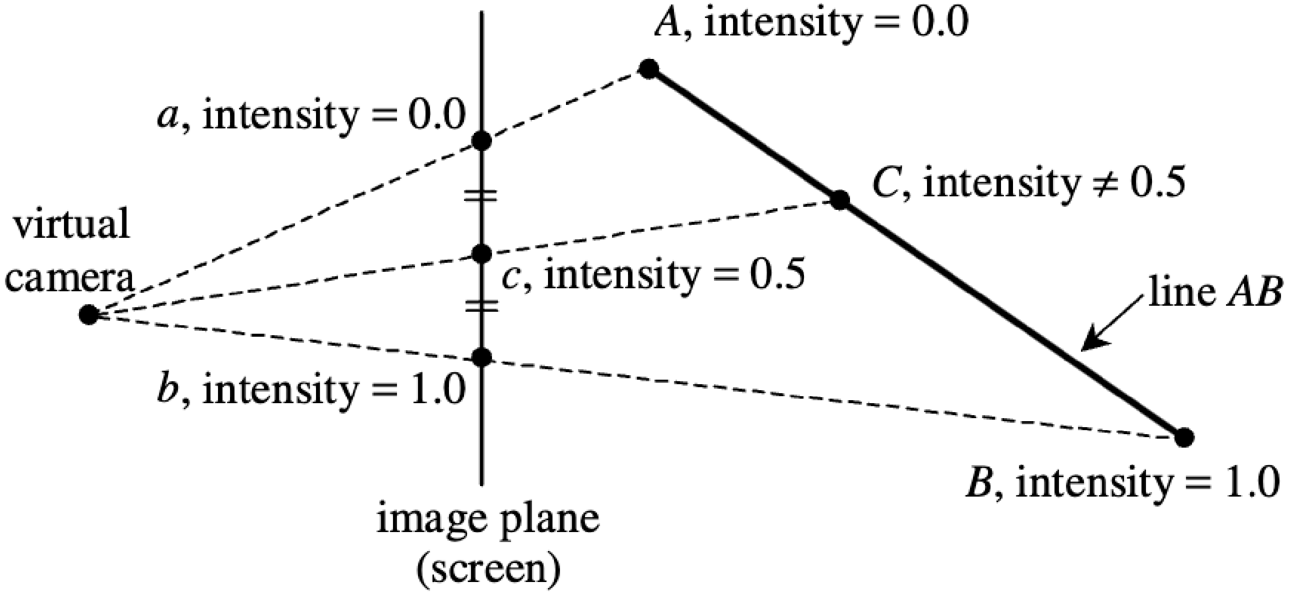
$$f(P) = a * f(A) + b * f(B) + c \ * f(C)$$



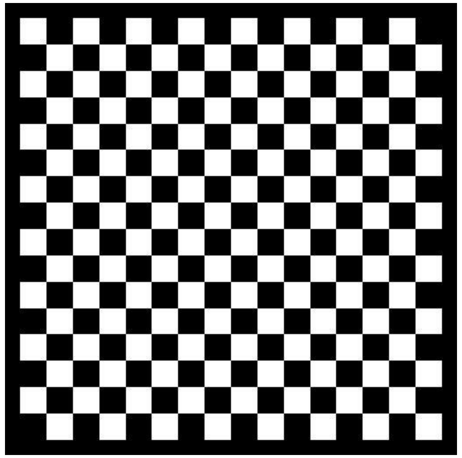


Perspective-Correct Interpolation

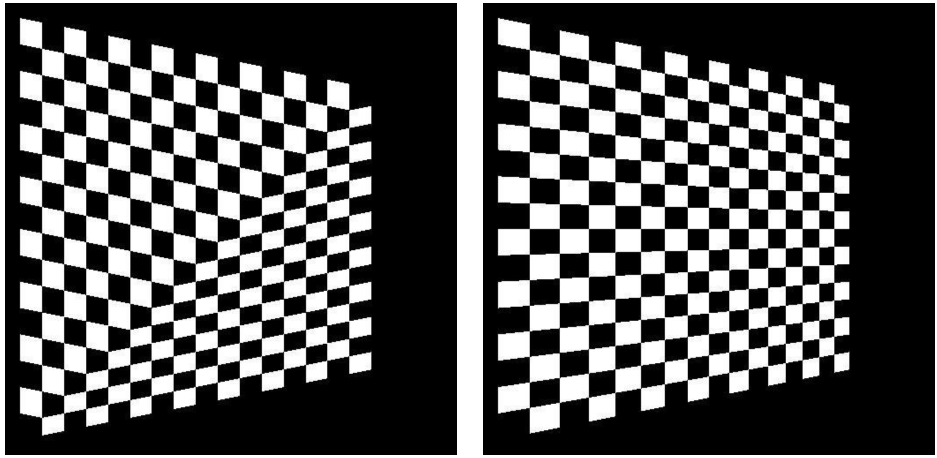
- Screen-space linear interpolation of vertex attributes yields distorted results



Kok-Lim Low, Perspective-Correct Interpolation, 2002



An example plane



Plane at 45-degree view

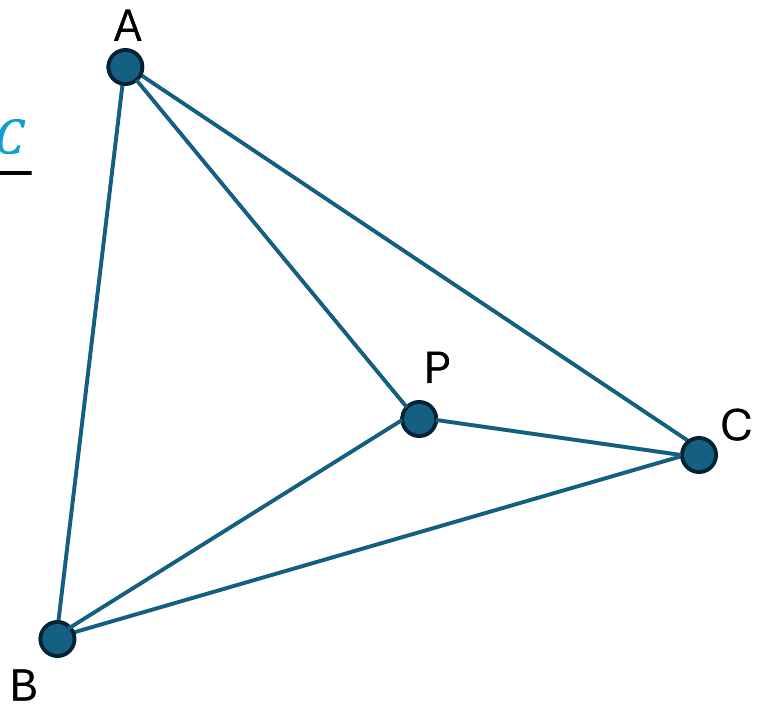
Andrew G. Zaferakis, Perspectively Correct Texture Mapping And Color Interpolation, 2000

Perspective-Correct Interpolation

- Modify barycentric coordinates (a, b, c) of P for perspective correct interpolation

$$f(P) = \frac{a * f(A)/w_A + b * f(B)/w_B + c\ * f(C)/w_C}{a/w_A + b/w_B + c/w_C}$$

w_A, w_B, w_C are the w-component in clip space coordinates of vertex A, B, C.

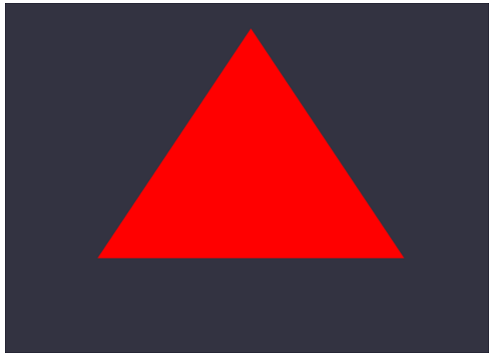


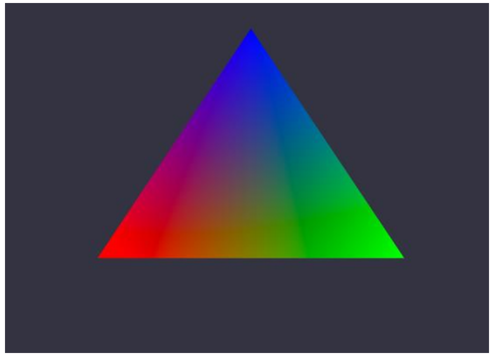
OpenGL Core Profile 4.4, Equation 14.9
https://registry.khronos.org/OpenGL/specs/gl/glspec44.core.pdf

Pixel Shading

Pixel Shading

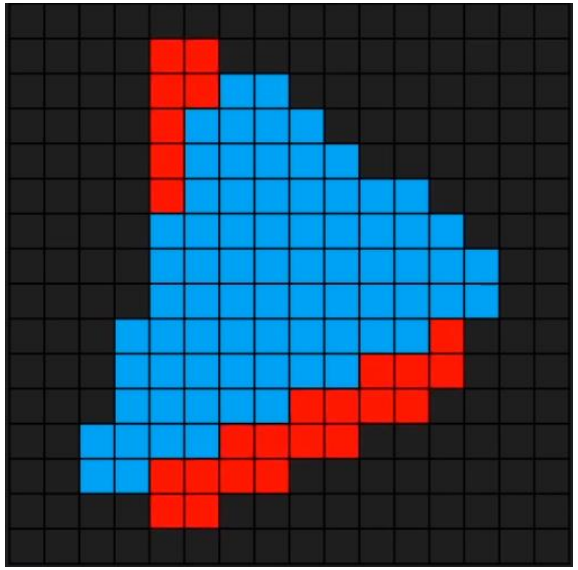
- Per-pixel shading computations are performed here
- Fragment shader outputs the color/depth for each pixel

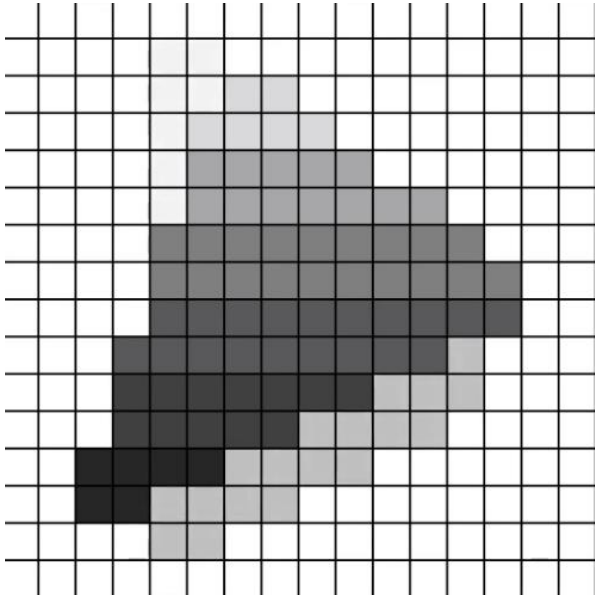




Z-Buffer

- Also known as depth buffer
- A built-in buffer for storing fragment depth (i.e., z-values) from the camera view
- Depth test for checking overlap triangles.
- Before writing a z-value, compare with the current z-value in the buffer.
- By default, store the closest z-value to the camera.





Merging

Merging

- Information for each pixel is stored in the *colour buffer* (a rectangular array of colours)
- Combine the fragment colour produced by the shading stage with the colour currently stored in the buffer: overwrite, alpha blending, etc.
- This stage is also responsible for resolving visibility using the z-buffer

Double Buffering

- The screen displays the contents of the color buffer

- To avoid perception of primitives being rasterized, *double buffering* is used

- Rendering takes place off screen in a *back buffer*
- Once complete, contents are swapped with the *front buffer*

Fragment Shader Example

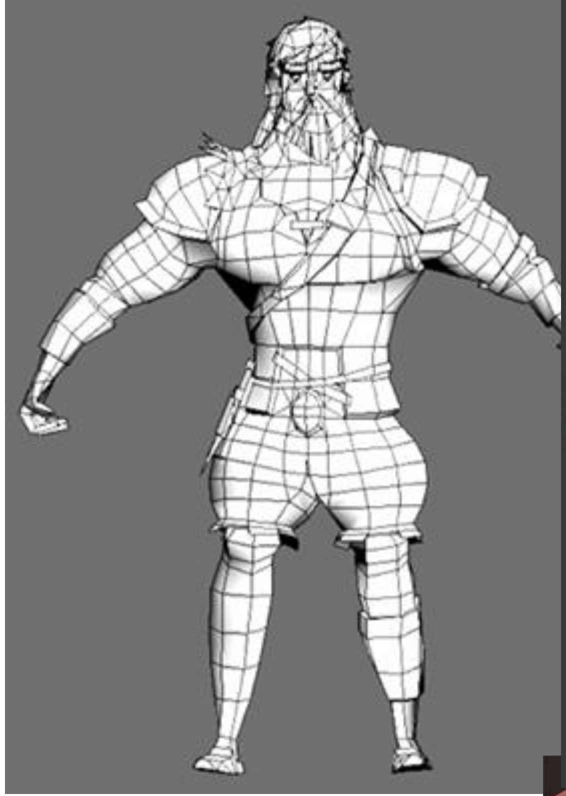
```
#version 330 core

out vec3 finalColor;

void main()
{
  finalColor = vec3(1, 0, 0);
}
```

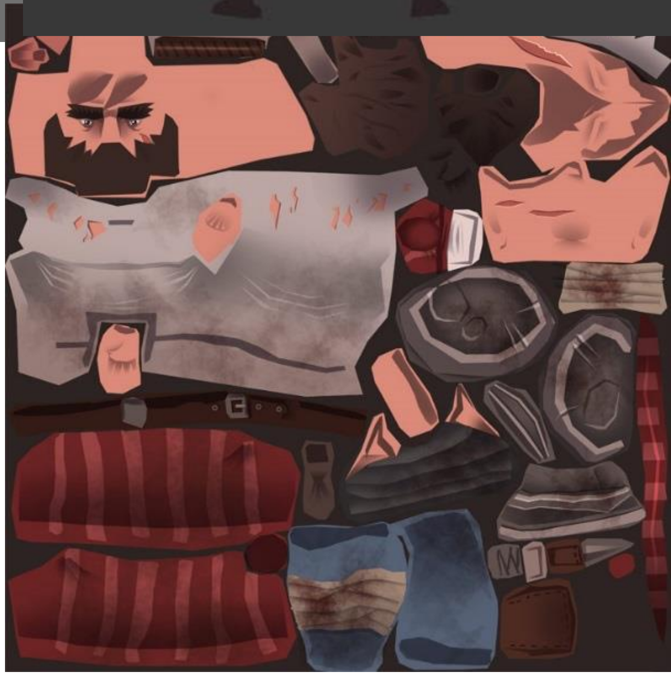
Fragment Shader Example

```
#version 330 core

in vec3 color;

out vec3 finalColor;

void main()
{
  finalColor = color;
}
```

Texture Mapping

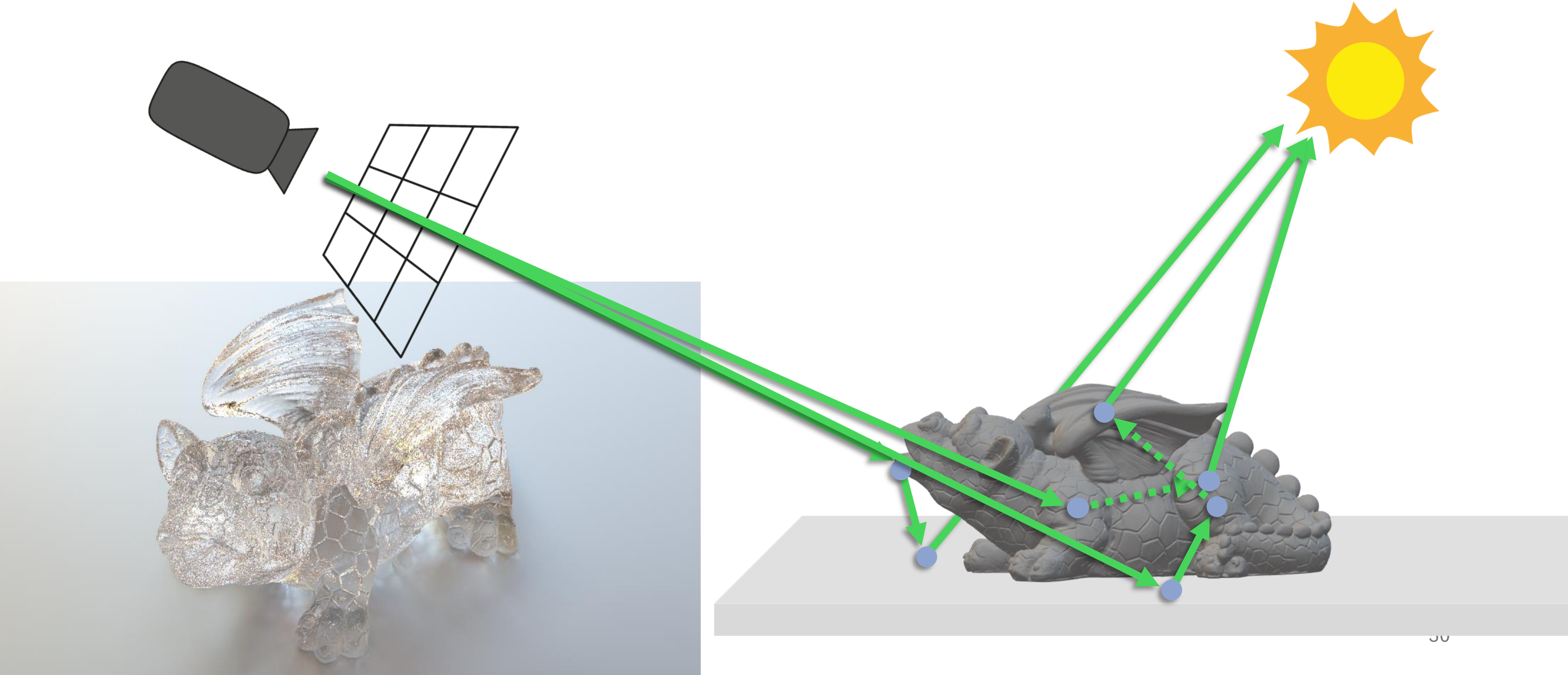
- Texture is a 2D image provided to a 3D geometry to enhance its appearance.
- Texture is often created offline in a 3D modeling software.
- Texture mapping is the process of wrapping the 2D image onto the geometry, often a triangle mesh.





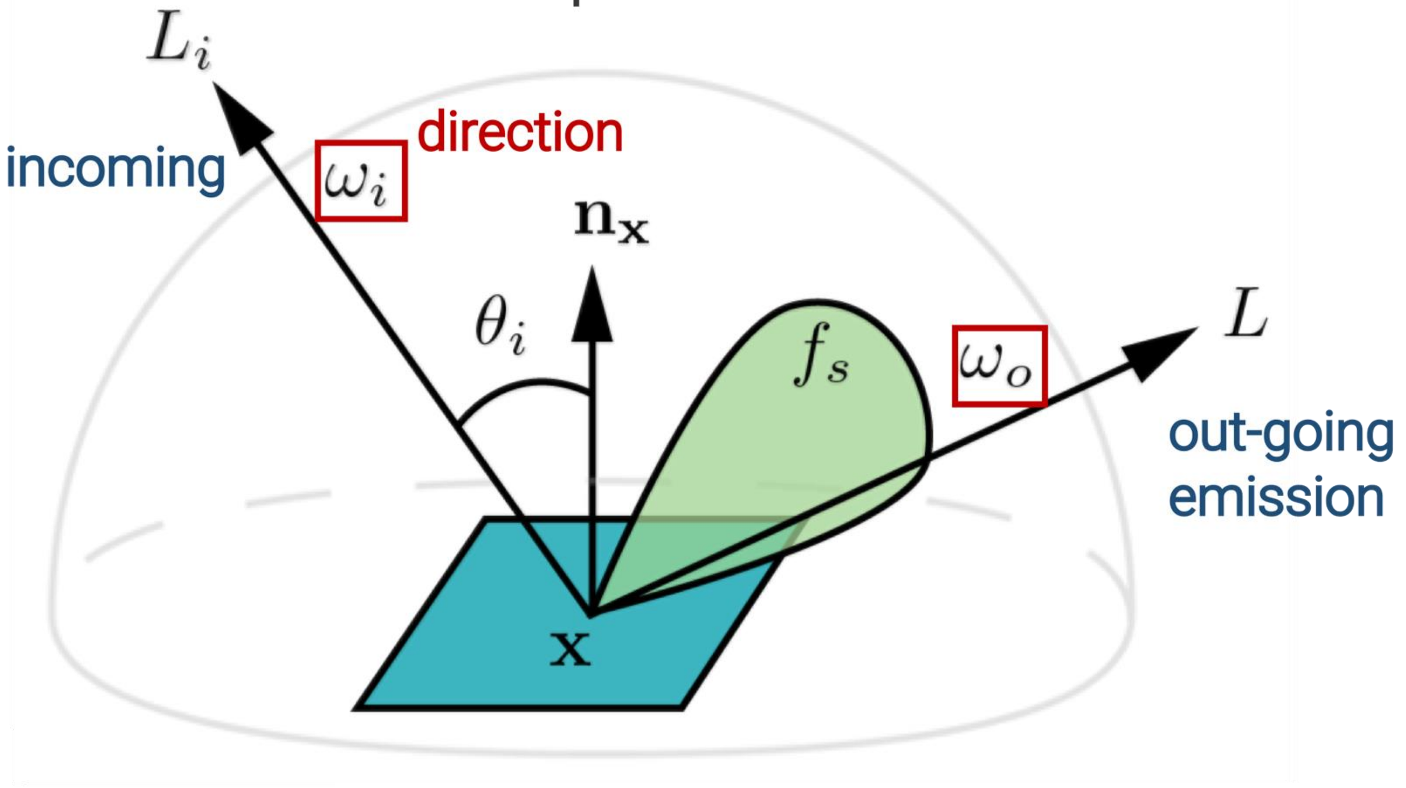
Cross of the Dutchman character

Shading



The Rendering Equation

$$\underbrace{\boxed{L(\mathbf{x},\boxed{\omega_o})}}_{\text{radiance}} = \boxed{L_e(\mathbf{x},\omega_o)} + \underbrace{\boxed{\int_\Omega}}_{\text{hemisphere}} \boxed{L_i(\mathbf{x},\boxed{\omega_i})}\,\underbrace{\boxed{f_s(\omega_i,\mathbf{x},\omega_o)}}_{\text{reflectance}}\,\underbrace{\boxed{\cos\theta_i}}_{\text{orientation}}\,\boxed{\mathrm{d}\omega_i}$$

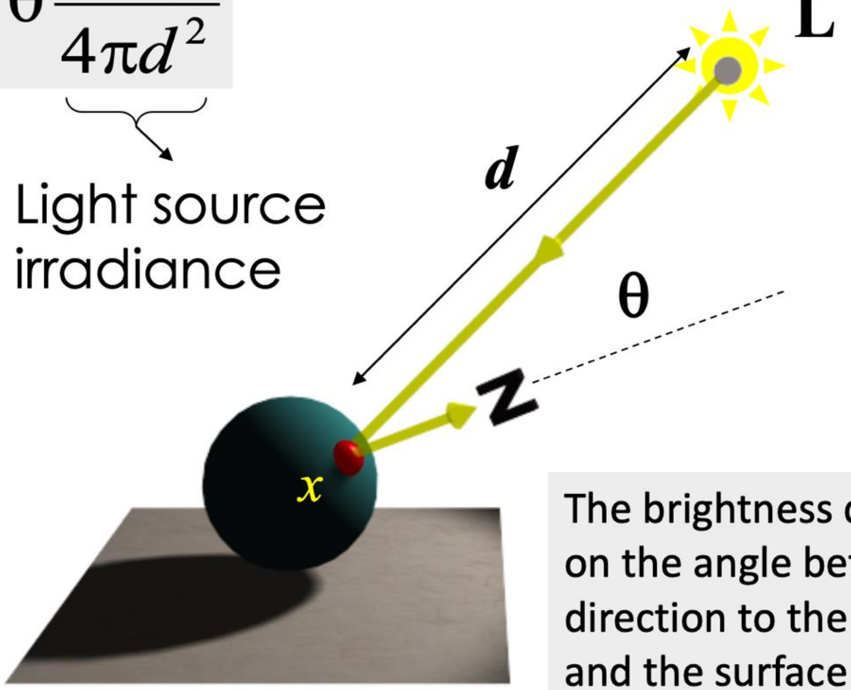


The Rendering Equation

Lambertian Lighting

- A common simplification of the rendering equation is to assume surfaces to be Lambertian (e.g., wooden material).

$$\underbrace{L_{r,d}(x,\cdot)}_{\text{Reflected radiance}} = \underbrace{\frac{\rho_d}{\pi}}_{\text{BRDF}} \cos\theta \underbrace{\frac{\Phi_S}{4\pi d^2}}_{\text{Light source irradiance}}$$



The brightness depends only on the angle between the direction to the light source and the surface normal

Shadow Mapping

- Any point seen by the light is illuminated, otherwise in shadow.

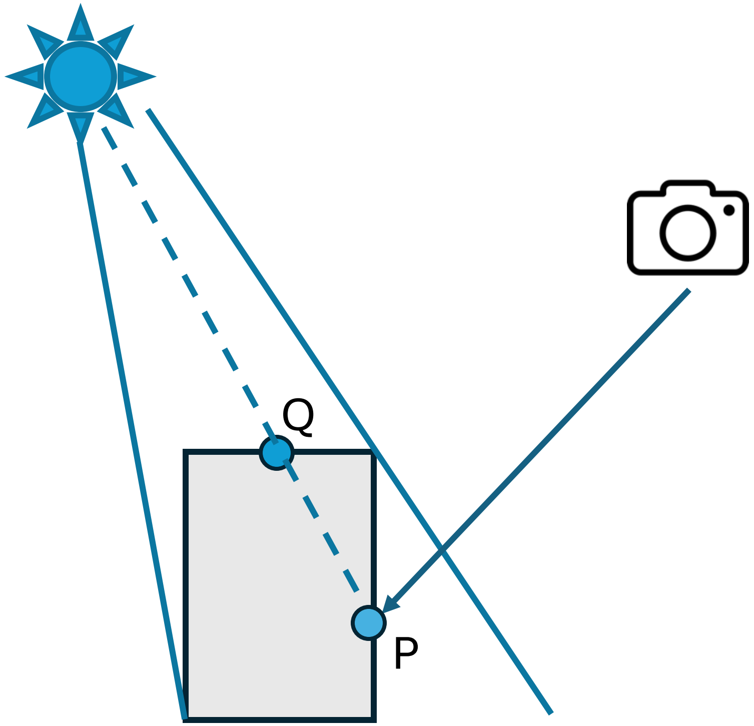


Image by Datta et al., Neural Shadow Mapping, SIGGRAPH

References

- A Whirlwind Introduction to Computer Graphics, SIGGRAPH 2023 course by Mike Bailey.

- Learn OpenGL by Joey de Vries, 2020.

- WebGL examples https://threejs.org/examples/

- Perspective-Correct Interpolation, Kok-Lim Low, 2002.