Rasterization & The Graphics Pipeline

Fast Approximations for Real-Time Graphics

Ashrafur Rahman

Adjunct Lecturer

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)

Index

Introduction

- Philosophy
- The Speed vs Accuracy Trade-off
- Realtime Challenge

The GPU Evolution

- History
- GPU Domination
- Modern GPU Architecture

The Modern Graphics Pipeline

- Definition
- Stages
- Advanced Pipeline
- Data Flow Through the Pipeline
- Pipeline Stage Types

APIs & Shading Languages

- Graphics APIs
- Shading Languages
- GLSL Overview

Input Assembly

- Overview

Introduction

Why Rasterization?

Why Rasterization?

Valorant - 120 FPS Gaming

Why Rasterization?

Valorant - 120 FPS Gaming

Up - 30 hours per frame

- **Real-time constraint:** Games need 60-120 FPS

Why Rasterization?

Valorant - 120 FPS Gaming

Up - 30 hours per frame

- **Real-time constraint:** Games need 60-120 FPS
- **Interactive experience:** User input must feel responsive

Why Rasterization?

Valorant - 120 FPS Gaming

Up - 30 hours per frame

- **Real-time constraint:** Games need 60-120 FPS
- **Interactive experience:** User input must feel responsive
- **Trade-off:** Sacrifice physical accuracy for speed

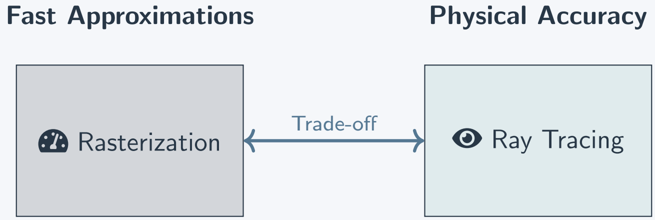
Why Rasterization?

Valorant - 120 FPS Gaming

Up - 30 hours per frame

- **Real-time constraint:** Games need 60-120 FPS
- **Interactive experience:** User input must feel responsive
- **Trade-off:** Sacrifice physical accuracy for speed
- **Goal:** Images that look good enough, delivered fast enough

Rasterization vs Ray Tracing: The Fundamental Choice



Rasterization:

- 60-240 FPS
- Clever approximations
- Hardware optimized
- "Good enough" quality

Ray Tracing:

- $\approx$ 0 FPS
- Physical simulation
- Computationally heavy
- Photorealistic

The Real-Time Graphics Challenge

Time Budget at 60 FPS

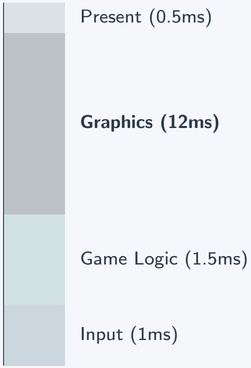
$\frac{1}{60} = 16.67$ milliseconds per frame

What needs to happen:

- Process input
- Update game logic
- Render graphics
- Present to screen

Graphics budget: $\sim$10-12ms

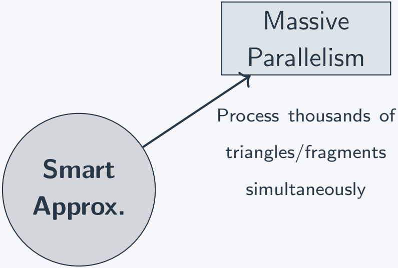
16.67ms



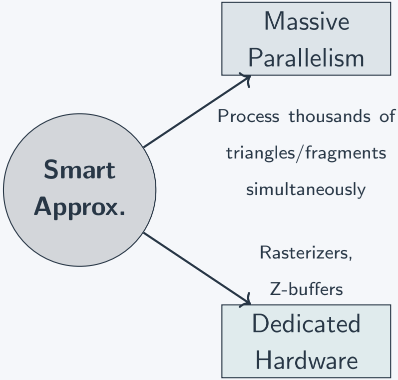
How Rasterization Achieves Speed

Smart Approx.

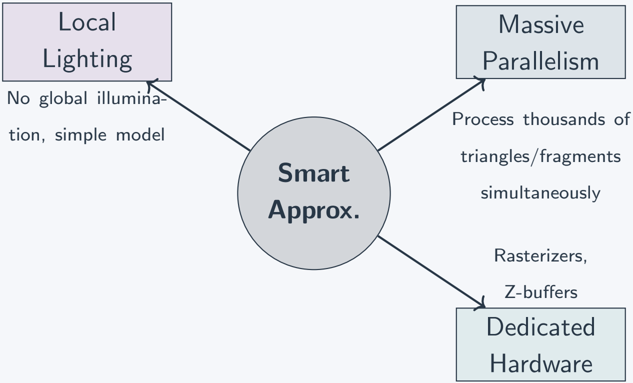
How Rasterization Achieves Speed



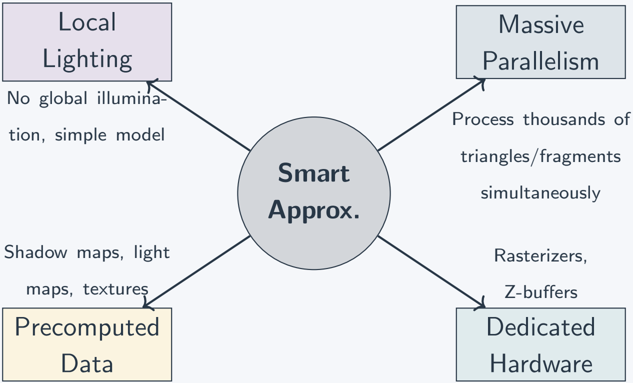
How Rasterization Achieves Speed



How Rasterization Achieves Speed



How Rasterization Achieves Speed



The Clever Approximations

What We Skip

- **Global illumination:** No light bouncing
- **Perfect shadows:** Use shadow maps
- **Perfect reflections:** Use environment maps
- **Complex materials:** Simplified BRDFs

What We Gain

- **Predictable performance:** Linear with triangle count
- **Hardware optimization:** Purpose-built silicon
- **Real-time interaction:** Immediate feedback
- **Scalable quality:** Adjust for performance

The GPU Evolution

A Brief History

1970s-1980s

Software Rendering

- Everything done on CPU
- Frame rates: 1-10 FPS
- Wireframe graphics

1990s

Fixed-Function GPUs

- 3dfx Voodoo, NVIDIA Riva
- Hardware rasterization
- Fixed pipeline stages

2000s

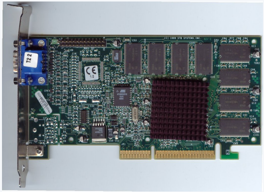
Programmable Shaders

- DirectX 8.0, OpenGL
- Vertex & Fragment shaders
- Creative freedom

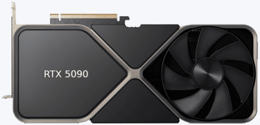
2010s+

Unified Architecture

- CUDA, OpenCL
- Compute shaders
- General-purpose



3dfx Voodoo 3 - 1999



NVIDIA GeForce 5090 - 2025

Why GPUs Dominate Graphics

CPU

4-16 complex cores
Large caches
Branch prediction
Out-of-order execution

Why GPUs Dominate Graphics

CPU	GPU
4-16 complex cores	1000s of simple cores
Large caches	Small caches
Branch prediction	SIMD execution
Out-of-order execution	Throughput optimized

Why GPUs Dominate Graphics

CPU

4-16 complex cores
Large caches
Branch prediction
Out-of-order execution

GPU

1000s of simple cores
Small caches
SIMD execution
Throughput optimized

Serial Tasks

Complex logic
Branching
Low latency

Why GPUs Dominate Graphics

CPU	GPU
4-16 complex cores	1000s of simple cores
Large caches	Small caches
Branch prediction	SIMD execution
Out-of-order execution	Throughput optimized

Serial Tasks

Complex logic
Branching
Low latency

Parallel Tasks

Simple operations
Same instruction
High throughput

Why GPUs Dominate Graphics

CPU

4-16 complex cores
Large caches
Branch prediction
Out-of-order execution

GPU

1000s of simple cores
Small caches
SIMD execution
Throughput optimized

Serial Tasks

Complex logic
Branching
Low latency

Parallel Tasks

Simple operations
Same instruction
High throughput

Perfect Match: Graphics + GPU

Graphics pipeline stages process thousands of vertices/fragments *independently*

$\Rightarrow$ Ideal for massively parallel GPU architecture

Modern GPU: The Graphics Powerhouse

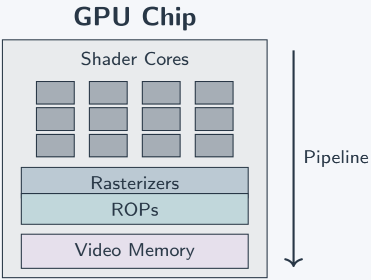
Hardware Implementation

GPU handles entire pipeline:

- **Vertex processing:** Shader cores
- **Rasterization:** Fixed-function units
- **Fragment processing:** Shader cores
- **Memory operations:** ROPs

GPU Driver handles:

- Command submission
- State management
- Resource allocation



The Modern Graphics Pipeline

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)
- Each undergoes **a series of identical operations**

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)
- Each undergoes **a series of identical operations**
- The operations can be divided into **stages**

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)
- Each undergoes **a series of identical operations**
- The operations can be divided into **stages**
- Each stage can be **efficiently** implemented in hardware

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)
- Each undergoes **a series of identical operations**
- The operations can be divided into **stages**
- Each stage can be **efficiently** implemented in hardware
- Stages can be **parallelized** for high throughput

Why a Graphics Pipeline?

The Rasterization Process

- GPU process vast numbers of **vertices** and **pixels** every frame (millions per second)
- Each undergoes **a series of identical operations**
- The operations can be divided into **stages**
- Each stage can be **efficiently** implemented in hardware
- Stages can be **parallelized** for high throughput

The Graphics Pipeline

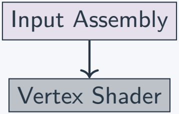
The graphics pipeline is a sequence of stages that process vertices and fragments in parallel, transforming 3D models into 2D images.

Pipeline Stages at a Glance

Input Assembly

Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

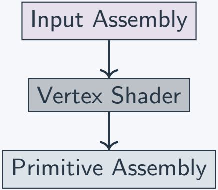
Pipeline Stages at a Glance



Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

Pipeline Stages at a Glance

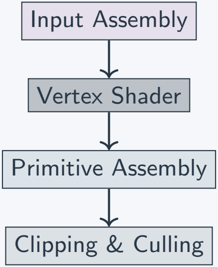


Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

Primitive Assembly: Group vertices into triangles (or other primitives).

Pipeline Stages at a Glance



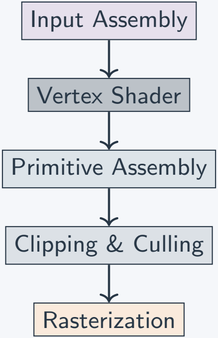
Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

Primitive Assembly: Group vertices into triangles (or other primitives).

Clipping & Culling: Discard or trim primitives outside the view frustum.

Pipeline Stages at a Glance



Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

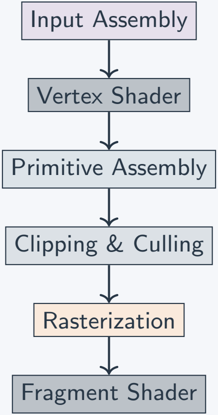
Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

Primitive Assembly: Group vertices into triangles (or other primitives).

Clipping & Culling: Discard or trim primitives outside the view frustum.

Rasterization: Convert triangles into a grid of fragments (potential pixels).

Pipeline Stages at a Glance



Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

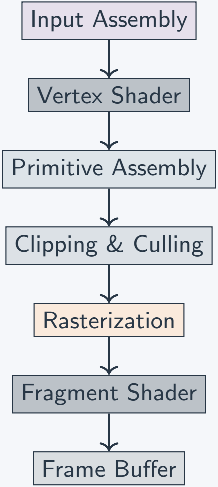
Primitive Assembly: Group vertices into triangles (or other primitives).

Clipping & Culling: Discard or trim primitives outside the view frustum.

Rasterization: Convert triangles into a grid of fragments (potential pixels).

Fragment Shader: `Programmable stage` — compute final color of each fragment.

Pipeline Stages at a Glance



Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

Primitive Assembly: Group vertices into triangles (or other primitives).

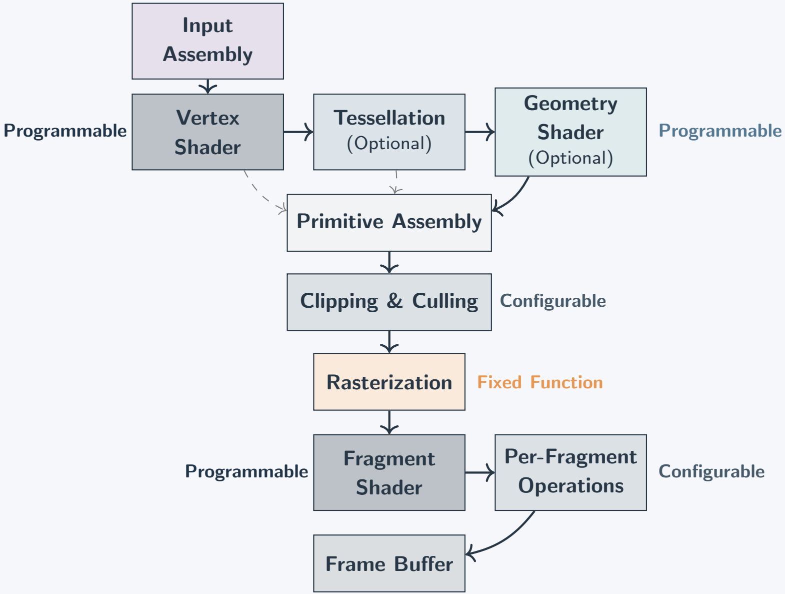
Clipping & Culling: Discard or trim primitives outside the view frustum.

Rasterization: Convert triangles into a grid of fragments (potential pixels).

Fragment Shader: `Programmable stage` — compute final color of each fragment.

Frame Buffer: Blend, depth-test, and write pixels to the screen.

Modern Advanced Pipeline



Data Flow Through the Pipeline

Input Data

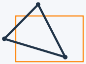
3D Vertices

Data Flow Through the Pipeline

Input Data

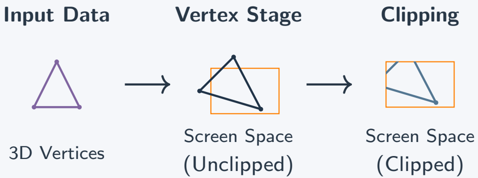
3D Vertices

Vertex Stage

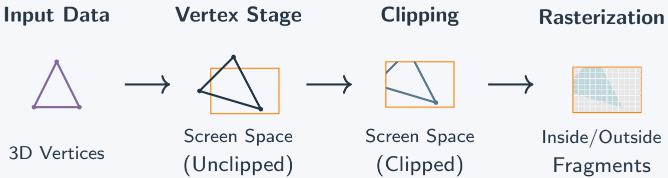


Screen Space
(Unclipped)

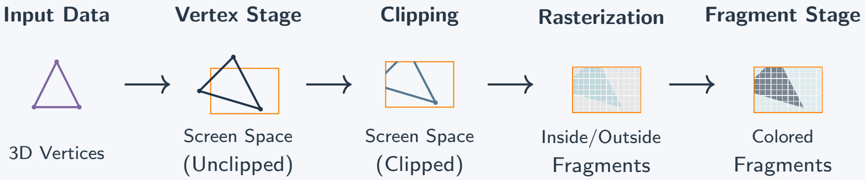
Data Flow Through the Pipeline



Data Flow Through the Pipeline



Data Flow Through the Pipeline



Programmable vs Fixed Function Stages

Programmable Stages

You write the code:

- **Vertex Shader:** Transform positions, compute lighting
- **Tessellation:** Subdivide surfaces adaptively
- **Geometry Shader:** Generate/modify primitives
- **Fragment Shader:** Compute final pixel colors

Maximum flexibility

Programmable vs Fixed Function Stages

Programmable Stages

You write the code:

- **Vertex Shader:** Transform positions, compute lighting
- **Tessellation:** Subdivide surfaces adaptively
- **Geometry Shader:** Generate/modify primitives
- **Fragment Shader:** Compute final pixel colors

Maximum flexibility

Fixed Function Stages

Hardware handles it:

- **Primitive Assembly:** Group vertices into triangles
- **Clipping:** Remove off-screen geometry
- **Rasterization:** Convert triangles to pixels
- **Depth Testing:** Z-buffer comparisons

Maximum performance

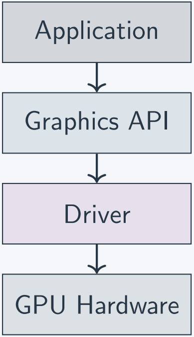
APIs & Shading Languages

Graphics APIs Overview

What is a Graphics API?

An **Application Programming Interface** that provides:

- Commands to control the GPU
- Abstraction over hardware differences
- Standard interface for graphics operations



Major Graphics APIs

OpenGL

Open Graphics Library

Perhaps the most widely used and most beginner-friendly graphics API. OpenGL was designed to be a cross-platform standard for rendering 2D and 3D graphics.

- Stable APIs
- High-level abstraction
- , , Support
- Two modes -
  - **Immediate Mode** - Deprecated, fixed function (Used by iGraphics)
  - **Retained Mode** - Modern OpenGL, uses shaders

Major Graphics APIs

Vulkan

Low-overhead, Cross-platform Graphics API

Developed by the Khronos Group as a modern successor to OpenGL. Designed for high-performance, multi-threaded rendering.

- Low-level control over GPU
- Better CPU-GPU parallelism
- Explicit memory and resource management
- , , Support
- Support via MoltenVK

Major Graphics APIs

DirectX (Direct3D)

Microsoft's Graphics API for Windows and Xbox

A powerful API suite used primarily for game development on Windows platforms.

- Direct3D for 3D rendering
- Deep integration with Windows OS and drivers
- High performance with hardware vendor optimizations
- , Support only

Major Graphics APIs

Metal

Apple's Low-level Graphics API

Designed to maximize performance on Apple devices, replacing OpenGL on Apple platforms.

- Low-overhead, low-level access
- Unified graphics and compute
- Tight integration with Apple hardware
- Support only (macOS, iOS, iPadOS)

Shading Languages

Purpose

Shading languages allow programmers to write code that runs on the GPU for:

- **Vertex processing** (transformations)
- **Fragment processing** (lighting, texturing, effects)
- **Compute operations** (general-purpose GPU computing)

Major Shading Languages:

- **GLSL** (OpenGL)
- **HLSL** (DirectX)
- **MSL** (Metal)
- **SPIR-V** (Vulkan)
  Can be compiled from GLSL or HLSL

GLSL: OpenGL Shading Language

GLSL Characteristics

- C-like syntax
- Built-in vector/matrix types
- Version-specific features

Data Types

- float, int, bool
- vec2, vec3, vec4
- mat2, mat3, mat4
- sampler2D, samplerCube

```
#version 330 core
// Vertex shader inputs
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
// Outputs to fragment shader
out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;
// Uniform variables
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model)))
             * aNormal;
    TexCoord = aTexCoord;

    gl_Position = projection
                  * view
                  * vec4(FragPos, 1.0);
}
```

GLSL Qualifiers

Storage Qualifiers

- in - Input from previous stage
- out - Output to next stage
- uniform - Buffer from CPU

Layout Qualifiers

Used to explicitly assign indices or binding points to resources.

- location - Attribute/output index
- binding - Texture/uniform buffer slot

```
#version 330 core
// Fragment shader
in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;
out vec4 FragColor;
uniform vec3 lightPos;
uniform vec3 viewPos;
uniform sampler2D texture_diffuse1;
void main() {
    vec3 color = texture(texture_diffuse1,
                         TexCoord).rgb;
    // Ambient
    vec3 ambient = 0.1 * color;
    // Diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(
                    lightPos - FragPos);
    float diff = max(dot(norm, lightDir),
                    0.0);
    vec3 diffuse = diff * color;
    // Result
    vec3 result = ambient + diffuse;
    FragColor = vec4(result, 1.0);
}
```

Input Assembly

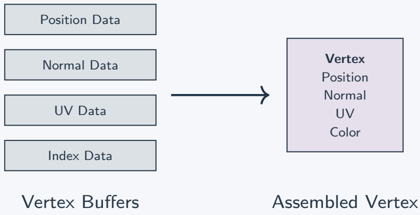
Input Assembly Stage

Input Assembly

Input: Vertex data from application
Output: Organized vertex streams for vertex shader

Purpose:

- Pull vertex data from memory
- Organize data into vertex attributes
- Handle indexed vs non-indexed drawing
- Set up primitive topology



Vertex Attributes

Common Vertex Attributes

- **Position:** 3D coordinates (vec3)
- **Normal:** Surface normal vector (vec3)
- **Texture Coordinates:** UV mapping (vec2)
- **Color:** Vertex color (vec3/vec4)

Vertex Layout Example

$$\text{Vertex} = \begin{cases} \text{Position:} & (x, y, z) \\ \text{Normal:} & (n_x, n_y, n_z) \\ \text{UV:} & (u, v) \\ \text{Color:} & (r, g, b, a) \end{cases} \tag{1}$$

Total size: $3 + 3 + 2 + 4 =$ 12 floats $=$ 48 bytes

Position	Normal	UV	Color
12 bytes	12 bytes	8 bytes	16 bytes

Primitive Topology

Primitive Types

Points: Individual vertices

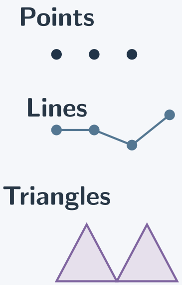
- Used for particle systems
- Point sprites

Lines: Connected line segments

- Wireframe rendering
- Debug visualization

Triangles: Most common primitive

- Standard for 3D surfaces
- Hardware optimized



Indexed vs Non-Indexed Drawing

Non-Indexed Drawing

Direct vertex specification
Each vertex is specified multiple times for shared vertices.

Problem:

- Vertex duplication
- Increased memory usage
- Inefficient for complex meshes

Indexed Drawing

Vertices referenced by indices
Each vertex is stored once, and indices specify how to connect them.

Benefits:

- No vertex duplication
- Lower memory usage
- Vertex cache friendly

Non-Indexed

Vertices: A, B, C, B, C, D

33% less data $\longrightarrow$

Indexed

Vertices: A, B, C, D

Indices: 0, 1, 2, 1, 2, 3

Vertex Shader

Vertex Shader Stage

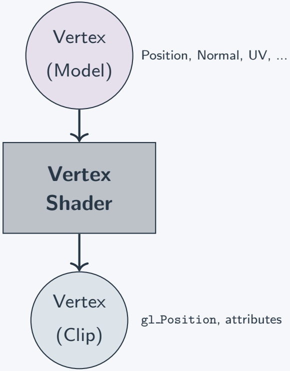
Vertex Shader

Input: Individual vertices with attributes
Output: Transformed vertices in clip space

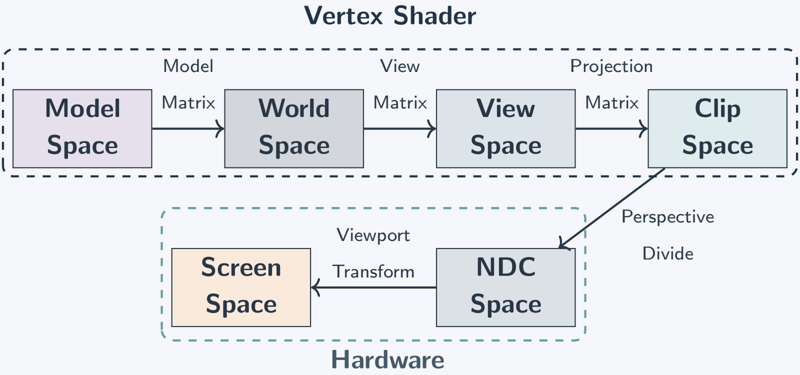
Purpose:

- Transform vertex positions through coordinate spaces
- Pass attributes to next stage
- Optional - Apply animations and deformations
- Optional - Calculate per-vertex lighting (Gouraud shading)

Programmable stage - you write the code!



The Transformation Pipeline



Transformation Matrices

Model Matrix

Purpose: Object-to-world transformation

$$\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$$

- **T**: Translation
- **R**: Rotation
- **S**: Scale

Transforms from model's local coordinates to world coordinates.

View Matrix

Purpose: World-to-camera transformation

$$\mathbf{V} = \text{lookAt}(\mathbf{eye}, \mathbf{target}, \mathbf{up})$$

Transforms from world coordinates to camera/eye coordinates.

Camera is at origin, looking down -Z axis.

Transformation Matrices

Projection Matrix

Purpose: Camera-to-clip space transformation

Perspective:

$$\mathbf{P}_{\text{persp}} = \begin{pmatrix} \frac{1}{\tan(\mathsf{fov}/2)\cdot\mathsf{aspect}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\mathsf{fov}/2)} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Orthographic:

$$\mathbf{P}_{\text{ortho}} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Clipping and Perspective Division

Clipping Phase

Purpose: Remove primitives outside the viewing frustum

- Operates in **clip space** (homogeneous coordinates)
- Keeps only geometry inside the cube: $-w \leq x, y, z \leq w$
- Output goes through **perspective division**:

$$\text{NDC} = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

Normalized Device Coordinates (NDC)

Canonical cube from $(-1, -1, -1)$ to $(1, 1, 1)$

- Independent of screen resolution
- Defines final visible region before pixel conversion
- x: left/right, y: bottom/top, z: near/far

Viewport Transformation and Screen Space

Viewport Transformation

Maps NDC coordinates to screen pixels:

$$\text{screen}_x = \left(\frac{\text{ndc}_x + 1}{2}\right) \cdot \text{width} + x_{\text{offset}}$$
$$\text{screen}_y = \left(\frac{\text{ndc}_y + 1}{2}\right) \cdot \text{height} + y_{\text{offset}}$$

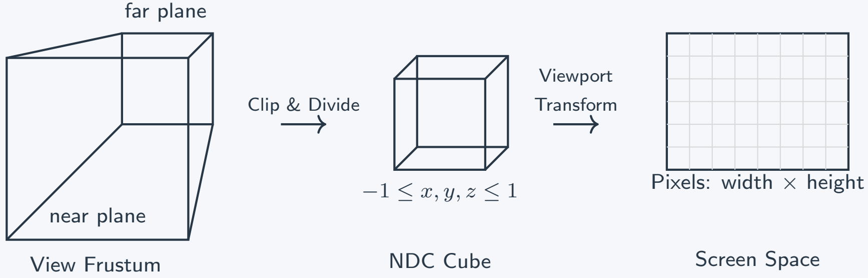
- Converts from $[-1, 1]$ range to pixel coordinates
- Accounts for viewport size and position

Screen Space

Final 2D position used for rasterization

- Measured in pixels
- Used by rasterizer to generate fragments

Frustum $\rightarrow$ NDC $\rightarrow$ Screen Space



Vertex Shader Example

```
#version 330 core
// Vertex shader inputs
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
// Outputs to fragment shader
out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;
// Uniform variables
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model)))
             * aNormal;
    TexCoord = aTexCoord;

    gl_Position = projection
                * view
                * vec4(FragPos, 1.0);
}
```

Output Variables

`gl_Position` - Clip space position

Gouraud Shading

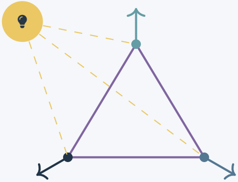
Gouraud (Per-Vertex) Shading

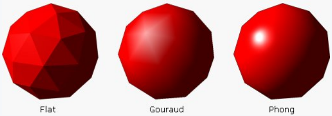
Compute lighting at vertices, interpolate across triangles

Process:

1. Calculate lighting at each vertex
2. Output vertex color
3. Hardware interpolates colors across triangle

Pros: Fast, good for distant objects
Cons: Poor specular highlights, faceted appearance





Tessellation Shader

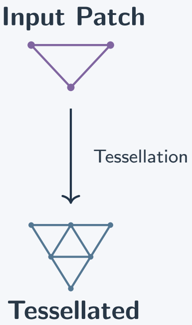
Tessellation Shader Stage

Tessellation Shader

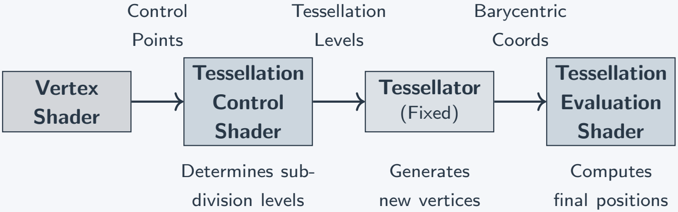
Input: Primitives
Output: Subdivided primitives with more vertices

Purpose:

- Subdivide low-poly meshes into high-poly
- Level-of-detail (LOD) based on distance
- Smooth curved surfaces
- Optional - Displacement mapping



Tessellation Sub-Stages



Tessellation Use Cases

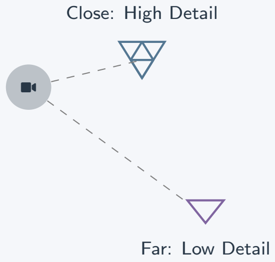
Level of Detail (LOD)

Adaptive subdivision based on:

- Distance from camera
- Screen space size
- Surface curvature
- Performance requirements

Benefits:

- Optimal vertex count
- Smooth transitions
- Better performance



Geometry Shader

Geometry Shader Stage

Geometry Shader

Input: Primitives (points, lines, triangles)
Output: New primitives (can generate or discard)

Purpose:

- Generate new geometry from existing primitives
- Discard primitives based on conditions
- Transform primitive types
- Add detail or effects

Billboarding: Use Case

Billboarding

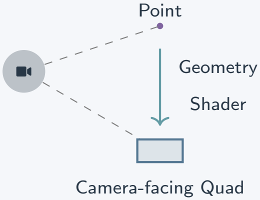
Convert points to camera-facing quads

Applications:

- Grass, leaves
- Particle systems
- Sprites and icons

Process:

1. **Input**: Single point
2. Calculate camera-facing orientation
3. **Output**: 4 vertices forming a quad



Primitive Assembly

Primitive Assembly Stage

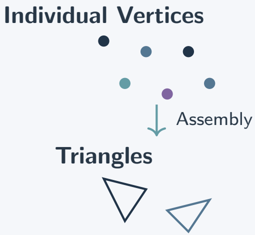
Primitive Assembly

Input: Individual vertices from vertex/geometry shaders

Output: Complete primitives (triangles, lines, points)

Purpose:

- Group vertices into geometric primitives
- Establish winding order
- Prepare for clipping and culling



Winding Order & Face Orientation

Winding Order

Determines which side of triangle is "front"

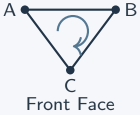
Counter-Clockwise (CCW):

- OpenGL default
- Front-facing when viewed from front
- Vertices ordered: A $\rightarrow$ B $\rightarrow$ C

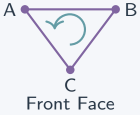
Clockwise (CW):

- DirectX default
- Front-facing when viewed from front
- Vertices ordered: A $\rightarrow$ C $\rightarrow$ B

Counter-Clockwise



Clockwise



Back-Face vs Front-Face

Face Determination

Process:

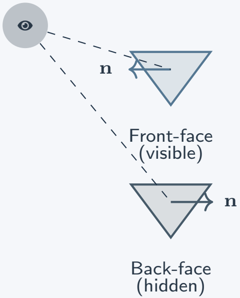
1. Calculate triangle normal using cross product
2. Determine viewing direction
3. Compare normal and view direction
4. Classify as front-face or back-face

Mathematical test:

$$\mathbf{n} = (\mathbf{v_1} - \mathbf{v_0}) \times (\mathbf{v_2} - \mathbf{v_0})$$

$$\text{facing} = \mathbf{n} \cdot \mathbf{view\_dir}$$

If facing $> 0 \rightarrow$ Front-face
If facing $< 0 \rightarrow$ Back-face



Questions?

References & Further Reading

- Peter Shirley and Steve Marschner et al. *Fundamentals of Computer Graphics (4th Edition)*. CRC Press, 2016.
  Available as PDF
- Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation (4th Edition)*. Morgan Kaufmann, 2023.
  Availabe online
- Peter Shirley. *Ray Tracing in One Weekend*. Self-published, 2016–2020.
  Project Website
- MIT OpenCourseWare: 6.837 Computer Graphics.
  ocw.mit.edu/6-837
- Scratchapixel: Learn Computer Graphics Programming.
  scratchapixel.com