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

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

Fast Approximations **Physical Accuracy**

Rasterization:

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

Ray Tracing:

- ≈ 0 FPS
- Physical simulation
- Computationally heavy
- Photorealistic

The Real-Time Graphics Challenge

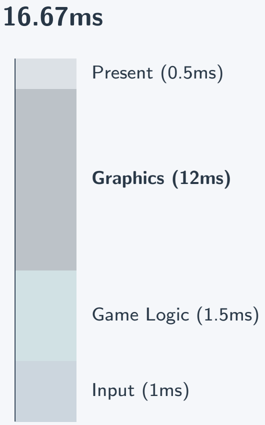
Time Budget at 60 FPS

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

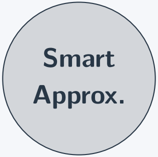
What needs to happen:

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

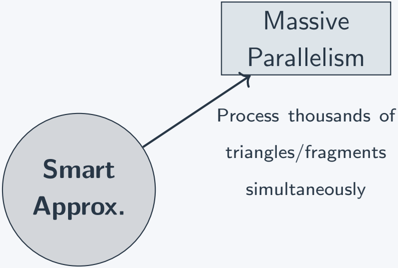
Graphics budget: $\sim$10-12ms



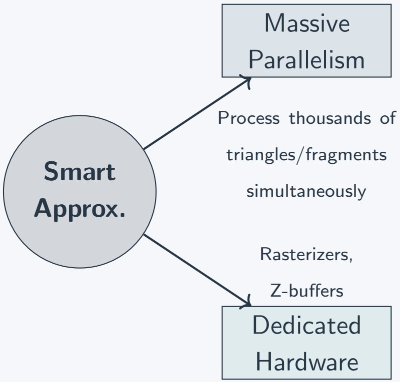
How Rasterization Achieves Speed



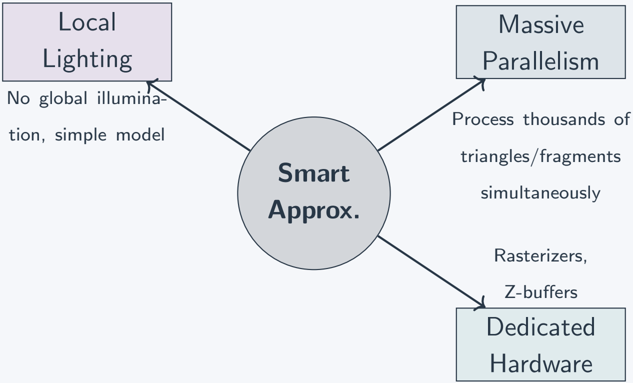
How Rasterization Achieves Speed



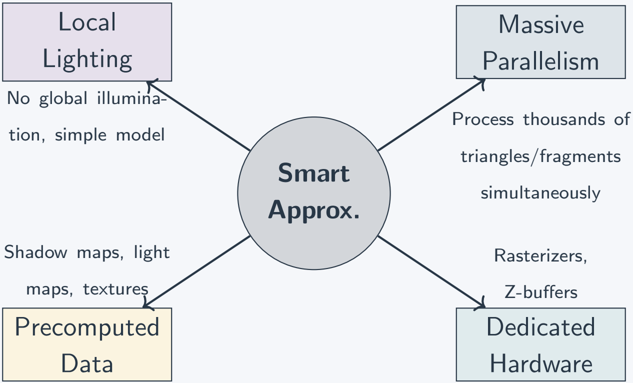
How Rasterization Achieves Speed



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

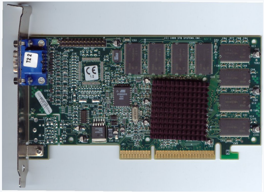
Programmable Shaders

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

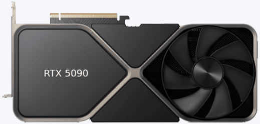
2010s+

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

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

Serial Tasks

Complex logic
Branching
Low latency

GPU

1000s of simple cores
Small caches
SIMD execution
Throughput optimized

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

Serial Tasks

Complex logic
Branching
Low latency

GPU

1000s of simple cores
Small caches
SIMD execution
Throughput optimized

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

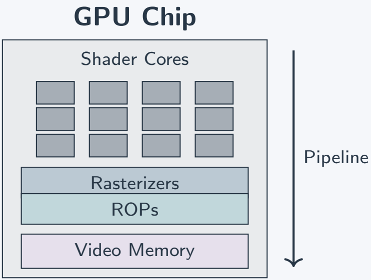
Hardware Implementation

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

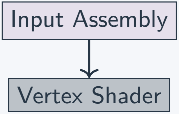
The graphics pipeline is a sequence of stages that process vertices and fragments in parallel, transforming 3D models into 2D images.

Pipeline Stages at a Glance

Input Assembly

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

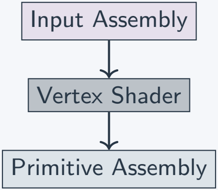
Pipeline Stages at a Glance



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

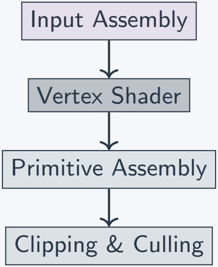


Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

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



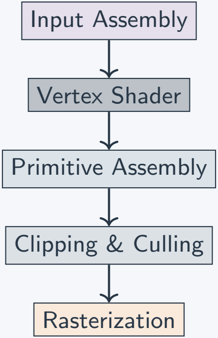
Input Assembly: Pull vertex data (positions, normals, UVs) into the pipeline.

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.

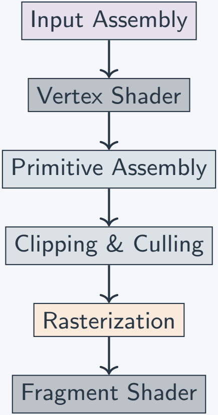
Vertex Shader: `Programmable stage` — transform each vertex from model to clip space.

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.

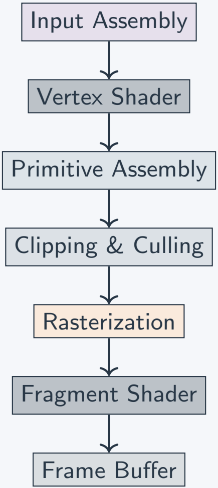
Primitive Assembly: Group vertices into triangles (or other primitives).

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).

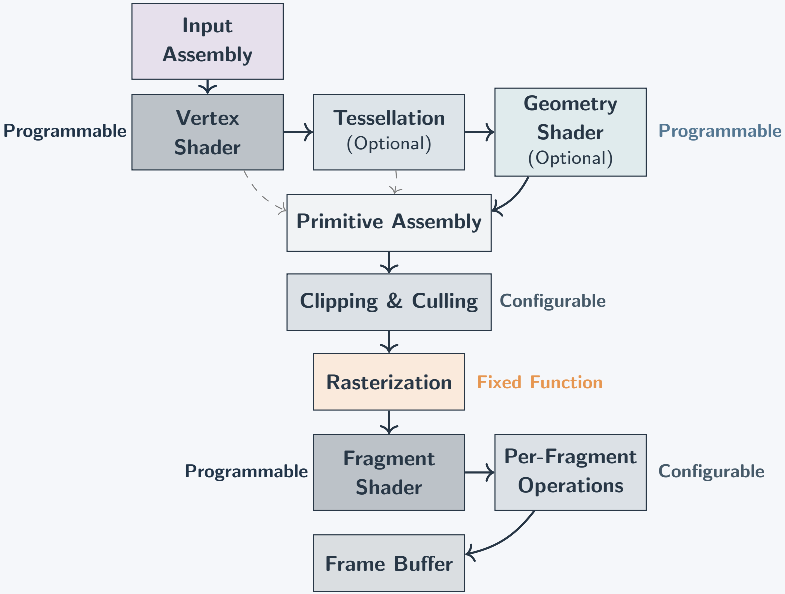
Clipping & Culling: Discard or trim primitives outside the view frustum.

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

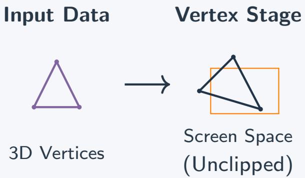


Data Flow Through the Pipeline

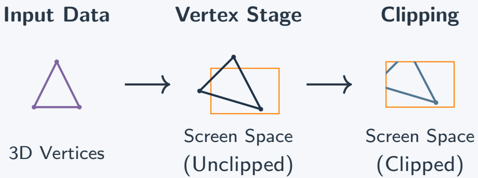
Input Data

3D Vertices

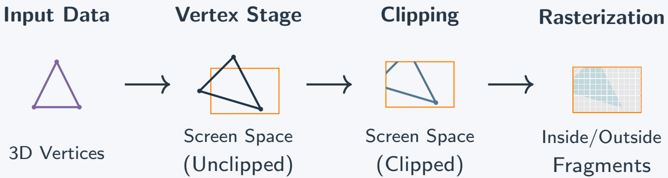
Data Flow Through the Pipeline



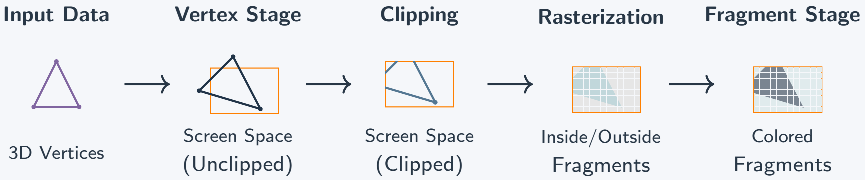
Data Flow Through the Pipeline



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

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