A Tech Artist's Guide to

Frame Creation

From CPU Cycles to GPU Execution

This book bridges the gap between low-level computer architecture and real-time rendering workflows through the eyes of a Technical Artist. A detailed, beginner-friendly breakdown of how a single rendered frame begins its journey — from CPU instruction execution and pipelining, all the way to GPU draw calls, deferred rendering, and modern lighting systems.

Table of Contents

Table of Contents 2

Chapter 1: Learning Path 4

Tech Artist Learning Path for 3D Environment Artists 4

Chapter 2: CPU-level Micro-operations 6

Multi-Cycle Execution Model 6

Pipelining 6

CPU Components Reference Glossary 7

Step-by-Step: Multi-Cycle Instruction Lifecycle 7

Chapter 3: CPU–GPU Connection 9

1. Command Buffer 9

2. State Changes 9

3. CPU → Driver → RHI 9

4. Memory Descriptors 9

5. The Ring Buffer & Synchronization 10

6. Hand-off Point 10

Chapter 4: GPU-level Micro-operations 11

1. GPU Execution Model — Warps & Threads 11

2. GPU High-Level Pipeline 11

3. GPU Caches & Memory Hierarchy 11

4. Fragment Shading & Overdraw 12

5. GPU Microhardware 12

Chapter 5: Frame Creation – CPU Role 13

Frame Initialization (CPU) 13

Culling 13

Draw Call Generation (CPU) 14

Optimization Techniques for Draw Calls 15

Monitoring and Tools for Draw Call Optimization 16

State Changes — Detailed Breakdown 16

Chapter 6: Frame Creation – GPU Role 18

Geometry Pass . . . 18

Rasterization . . . 18

Fragment (Pixel) Shader . . . 19

G-buffer Storage . . . 19

Lighting Pass . . . 20

Chapter 7: Lighting Techniques & Unreal Engine 5 22

Fundamentals . . . 22

Advanced Lighting Techniques . . . 22

Global Illumination . . . 23

Lumen (Unreal Engine 5) . . . 23

Shadow Casting in Unreal Engine . . . 24

Screen-Space Techniques . . . 25

Chapter 8: Advanced Deferred Rendering Techniques 26

Multiple Render Targets (MRTs) . . . 26

Clustered and Tiled Lighting . . . 26

Shader Complexity Reduction . . . 26

Temporal Techniques . . . 26

Nanite Virtualized Geometry . . . 27

Virtual Shadow Maps (VSM) . . . 27

Post-Processing Pipeline Optimization . . . 27

Ray Tracing Integration . . . 27

Compute Shaders for Custom Passes . . . 28

Chapter 1: Learning Path

Tech Artist Learning Path for 3D Environment Artists

A structured progression from foundational graphics concepts through advanced specialization, designed specifically for 3D Environment Artists moving into Technical Art.

Stage 1: Foundation Reinforcement (0–1 Month)

Goal: Solidify graphics programming concepts and art-technical overlap.

- GPU Architecture Basics (SIMD, shaders, VRAM, buffers)
- Render Pipeline stages (deferred, forward and hybrid)
- Shader math (dot/cross product, matrix transforms, world/view/projection spaces)

Tools

- Unreal Engine Shader Complexity view
- RenderDoc or Nsight for pipeline inspection

Stage 2: Shader & Material Mastery (1–3 Months)

Goal: Learn how to write and optimize custom shaders.

- HLSL / GLSL Basics
- Material Nodes vs Custom Nodes in Unreal
- UV manipulation, WorldAligned textures
- Dynamic parameters (time-based, user-driven, vertex color-based)

Practice

- Create dissolve, Fresnel, toon, triplanar, and layered blend shaders
- Use Material Functions and Instances properly
- Dive into Custom Expressions in UE Materials

Stage 3: Real-Time VFX & Niagara Systems (3–4 Months)

Goal: Use shaders and data flow to build compelling effects.

- Niagara System structure and particle modules
- GPU vs CPU particles
- Using mesh data (normals, UVs, etc.) in particle effects
- Blueprint-driven VFX triggers

Practice

- Make explosion, dissolve, energy pulse, and portal effects
- Use custom vector fields
- Interact with materials using dynamic parameters

Stage 4: Tools & Pipeline Scripting (3–6 Months)

Goal: Build automation tools and pipelines to support artists and level designers.

- Python for Blender / Maya
- Unreal Python API + Editor Utility Widgets
- Blueprint Scripting for Tools
- P4V source control & asset dependency management

Projects

- Automate mesh cleanup, LOD generation, batch material setup
- Write a batch export/import system
- Build a custom UE Editor Tool (e.g., spawn foliage, rename batch, fix pivots)

Stage 5: Advanced Topics + Specialization (6+ Months)

Rendering Optimization (Advanced)

- Clustered Forward+ lighting
- Temporal Super Resolution (TSR)
- Nanite/Lumen tuning for MR/VR

Proceduralism

- Houdini for level art and foliage generation
- Blueprint procedural tools in Unreal

Performance Profiling

- Unreal's GPU Profiler, stat commands
- Draw call cost, shader complexity, overdraw, GPU time

Debugging & Visualization

- Debugging shader output using G-buffer visualization
- Heatmaps for overdraw and lighting cost

Chapter 2: CPU-level Micro-operations

Multi-Cycle Execution Model

The multi-cycle concept means each instruction is broken into smaller steps (cycles), each executed over multiple clock cycles. Instead of completing an entire instruction in one cycle (single-cycle CPUs), multi-cycle CPUs divide work across dedicated units.

- Instruction: A machine code command telling the CPU to perform a specific task (e.g. ADD r1, r2).
- Cycle: A single step in the process of executing an instruction.
- Clock Cycle: The smallest unit of time recognized by the CPU, defined by the system clock.

Pipelining

Pipelining overlaps multiple instruction stages across multiple instructions simultaneously. While instruction A is executing, B is decoding, and C is being fetched. Think of it as cars on a highway — each in a different lane but moving forward at the same time.

CPU Instruction Pipeline — 5 Stages



Pipeline Stalls

A stall occurs when the CPU pipeline must pause due to data dependencies, control flow uncertainty, or resource conflicts.

Type	Description	Example
Data Hazard	Instructions depend on results from previous ones.	ADD r1, r2 → MUL r3, r1 (r1 not ready)
Control Hazard	Branch/jump — next instruction unknown.	if (x == 1) — CPU doesn't know where to go
Structural Hazard	Two instructions compete for the same hardware.	Two memory accesses simultaneously

Solutions: Hardware Techniques

Technique	Fixes	Explanation
Out-of-Order Execution	Data & structural	Executes later instructions while earlier ones resolve.
Tomasulo's Algorithm	Data hazards	Dynamic scheduling via reservation stations & ROB.
Speculative Execution	Control hazards	CPU guesses branch result and executes ahead.
Branch Prediction	Control hazards	Predicts branch path to avoid pipeline flushing.
Register Renaming	Data hazards (WAW/WAR)	Allocates new physical registers to eliminate false dependencies.
Superscalar Execution	Structural & data	Executes multiple instructions per clock cycle.
Prefetching	Memory/data stalls	Loads instructions or data into cache before needed.

Technique	Fixes	Explanation
Caching (L1/L2/L3)	Memory/data stalls	Keeps frequently used data close to the CPU.
SMT	All types	Thread-level parallelism — stalled thread yields to another.

CPU Components Reference Glossary

ID	Name	Function
PC	Program Counter	Holds the address of the next instruction to fetch. Increments each cycle.
MAR	Memory Address Reg.	Receives the PC address to access instruction/data in RAM.
MDR	Memory Data Reg.	Temporarily holds data fetched from or written to RAM.
IR	Instruction Register	Holds the current instruction being decoded/executed.
CU	Control Unit	Decodes the instruction and orchestrates the CPU with control signals.
ALU	Arithmetic Logic Unit	Performs math and logical operations (ADD, SUB, AND, etc.).
ROB	Reorder Buffer	Holds instruction results to commit them in-order (OoO CPUs).
RS	Reservation Station	Buffers instructions until operands are ready — enables dynamic scheduling.

Step-by-Step: Multi-Cycle Instruction Lifecycle

1. Instruction Fetch

- Program Counter (PC) points to the memory address of the next instruction.
- MAR loads this address; instruction is read from RAM into the IR.
- PC is incremented (usually by 4) to point to the next instruction.

2. Instruction Decode

- Control Unit (CU) reads the opcode from the IR.
- Decides the type of operation (arithmetic, branch, memory, etc.).
- Operands (registers like eax, ebx) are fetched — no computation yet.

3. Execute / Address Calculation

- ALU performs operations if arithmetic (ADD, MUL, etc.).
- For memory instructions (LOAD/STORE), effective address = base + offset.

4. Memory Access / Result Preparation

- LOAD: value fetched from memory into MDR.
- STORE: value written to memory. ALU ops hold result for commit.

5. Write-Back

- Result committed to a register (e.g. eax ← result) or memory.
- PC advances to next instruction unless stalled.

Component Activity per Cycle

Cycle	Active Hardware	Description
1	PC, MAR, Memory → IR	Fetch instruction from memory
2	CU, Register File	Decode + operand fetch
3	ALU	Execute or address calculation
4	Memory Controller	Load/store/move result
5	Register File	Write-back result

Chapter 3: CPU–GPU Connection

The CPU and GPU are separate processors with distinct memory spaces and execution models. Understanding how they communicate is critical for optimizing real-time rendering performance.

1. Command Buffer

The CPU does not talk directly to GPU hardware on every draw call — that would stall the CPU. Instead, the driver builds a command buffer: a list of GPU commands stored in memory. Think of it as a to-do list for the GPU.

- Bind vertex buffer at address X
- Bind index buffer at address Y
- Bind texture at address Z
- Use shader program #123
- Draw N primitives

What's Inside the Command Buffer?

- Set pipeline state — shader binding, blend mode, render target
- Bind resources — vertex buffer, index buffer, texture samplers (memory pointers only)
- Issue the draw — DRAW_INDEXED_PRIMITIVE (count=3,000, base=0)

The command buffer does NOT store actual mesh data. It stores pointers and instructions telling the GPU where to find resources in VRAM/system memory.

2. State Changes

Any change in pipeline state — shaders, blend mode, textures, render targets — creates a new draw call. GPU states must be fully serialized; you cannot half-switch. State changes force the GPU to flush current work, rebind data, and restart.

- As a Tech Artist: minimize material and shader changes to reduce draw call count.
- Group objects by material to allow batching into fewer draw calls.

3. CPU → Driver → RHI

In Unreal Engine, the RHI (Rendering Hardware Interface) is the abstraction layer that translates engine rendering code into the graphics API (DirectX 12, Vulkan, Metal, etc.). The driver then translates API calls into actual ISA (Instruction Set Architecture) machine code that the GPU understands.

- Shaders are pre-compiled into GPU machine code at build time.
- At draw time, the driver simply references the compiled shader — no recompilation.
- RHI abstracts hardware differences, letting Unreal run on multiple platforms.

4. Memory Descriptors

The CPU provides pointers to GPU resources — vertex buffers, index buffers, textures. These are typically addresses in VRAM or PCIe-accessible shared memory.

- The GPU executes exactly what the descriptor specifies — it does not search for resources.
- Misaligned or non-mipmapped textures → more bandwidth pressure → slower sampling.
- Descriptor sets (Vulkan) or root signatures (DX12) define what resources the GPU can access.

5. The Ring Buffer & Synchronization

The GPU runs asynchronously relative to the CPU. The CPU writes commands to a ring buffer (circular queue) and the GPU consumes them independently.

Scenario	What Happens	Result
CPU too far ahead of GPU	Ring buffer fills up	CPU stalls waiting — CPU bound
GPU lags too far behind CPU	Ring buffer drains too slowly	GPU bound — GPU is the bottleneck
CPU and GPU balanced	Buffer flows smoothly	Optimal frame pacing

6. Hand-off Point

Once a draw call is committed to the command buffer, the GPU scheduler pulls it and begins execution:

- Input Assembler fetches vertices and indices from the bound buffers.
- Shader cores begin executing the vertex shader.
- Rasterizer and fragment pipeline continue from there.

The CPU is now free to prepare more work — unless GPU sync (e.g. readback, fences) is required. This async nature is what allows the CPU and GPU to run in parallel across multiple frames.

Chapter 4: GPU-level Micro-operations

Where the CPU executes a small number of complex threads sequentially, the GPU executes thousands of simpler threads in massive parallel. Understanding this execution model is fundamental to writing performant shaders and materials.

1. GPU Execution Model — Warps & Threads

- GPUs run in SIMT (Single Instruction, Multiple Threads) fashion.
- Threads are grouped into warps (NVIDIA: 32 threads) or wavefronts (AMD: 64 threads).
- All threads in a warp execute the same instruction simultaneously on different data.
- Thread divergence (e.g. if/else branches in a shader) forces serialization → kills efficiency.

Key Implication for Tech Artists

If a material shader branches based on a texture value (e.g. if alpha > 0.5), threads in the same warp that take different paths must execute both paths serially. Avoid dynamic branching in shaders where possible — use static switches instead.

2. GPU High-Level Pipeline

GPU Rendering Pipeline — Fixed & Programmable Stages



Stage	Type	Description
Input Assembler	Fixed	Reads vertex and index buffers. Assembles primitives.
Vertex Shader	Programmable	Transforms vertices to clip space. Outputs normals, UVs, colors.
Rasterizer	Fixed	Converts triangles into fragments. Interpolates attributes.
Fragment/Pixel Shader	Programmable	Executes material logic — texture sampling, lighting, math.
Output Merger / ROPs	Fixed	Depth/stencil test, blending, writes final pixel to framebuffer.

3. GPU Caches & Memory Hierarchy

Memory latency is the single biggest performance bottleneck on the GPU. The cache hierarchy exists to hide this latency.

Level	Size	Speed	Notes
L1 Texture Cache	~16–64 KB / SM	Very fast	Optimized for 2D spatial locality. Per shader core.
L2 Cache	~1–4 MB	Moderate	Shared across all shader cores. Slower than L1.
VRAM	4–24+ GB	High latency	Main GPU memory. PCIe transfer to CPU is very slow.

Level	Size	Speed	Notes
System RAM	Shared	Slowest	PCIe bandwidth limits; avoid frequent CPU↔GPU transfers.

L1 cache misses per pixel are a killer on mobile GPUs (e.g. Quest 3). Texture sampling patterns, mip bias, and sampler sharing directly impact cache efficiency.

4. Fragment Shading & Overdraw

A fragment is a candidate pixel generated by the rasterizer. Multiple fragments may map to the same pixel — this is overdraw. Each fragment runs the full fragment shader, so overdraw multiplies workload. ROPs then resolve which fragment survives via depth test and blending.

- Overdraw of 2–4x is common in scenes with transparency, particles, or foliage.
- On Quest 3, fragment shading is the #1 bottleneck — even moderate overdraw saturates bandwidth.
- Fix: sort transparent objects back-to-front, use Early-Z, reduce particle opacity layers.

5. GPU Microhardware

Unit	Role
Shader Cores (ALUs, EFUs)	Perform math — dot products, trig, matrix multiplies, adds.
Texture Units (TMUs)	Fetch and filter texels from cache/VRAM. Sample multiple layers simultaneously.
Rasterizer	Fixed-function hardware. Converts triangles into fragments, interpolates attributes.
ROPs (Render Output Units)	Apply depth test, stencil test, blending. Write final pixels to framebuffer.
Warp Scheduler	Issues warps to shader cores. Switches to other warps when one stalls on memory.

Step-by-Step GPU Data Flow

- 1. CPU submits draw call → enters command buffer.
- 2. Input Assembler fetches vertices/indices → Vertex Shader runs on ALUs.
- 3. Rasterizer generates fragments → interpolates normals, UVs, position.
- 4. Fragment Shader (warps of threads) → ALUs + TMUs execute material logic.
- 5. Texture Units fetch texels (L1 hit → fast; L2 → slower; VRAM → very slow).
- 6. Fragments resolved at ROPs — depth test, blend, write surviving pixel.
- 7. Final pixels written to framebuffer (or G-Buffer in deferred rendering).

Chapter 5: Frame Creation – CPU Role

Frame Initialization (CPU)

1. Game Logic Update

- Computes physics, AI, and gameplay logic.
- Processes player inputs and determines object responses.
- Updates transformations (position, rotation, scale) for all objects.

2. Visibility Determination

The CPU identifies visible objects based on the camera's position and view frustum:

- Frustum Culling: Eliminates objects outside the camera's field of view.
- Occlusion Culling: Removes objects blocked by other objects using occlusion queries.

3. Scene Graph Traversal

- CPU organizes the scene into a data structure (BVH or Octree) for efficient processing.
- Visible objects are sorted by material, texture, and properties to minimize state changes.

Culling

A — CPU-Based Frustum/Occlusion Culling

The CPU performs frustum and occlusion culling before sending draw calls to the GPU. When a large building blocks a smaller object, the CPU identifies the occlusion and skips the smaller object entirely.

- Technique: Bounding box checks or hierarchical depth checks to determine occlusion.
- Advantages: Reduces GPU workload by eliminating unnecessary draw calls.
- Disadvantages: Can be expensive with large numbers of objects without optimization.

Optimization Topics

- Hierarchical Z-Buffer or Bounding Volume Hierarchies (BVH) for faster culling.
- Combine culling with LOD to reduce high-detail objects during rendering.
- Manual Culling: Set up culling volumes in Unreal for large open-world scenes.

B — GPU-Based Occlusion Culling

The GPU performs occlusion culling using hardware occlusion queries or depth testing. It renders a simplified version of the scene (bounding volumes) and checks visibility.

- Technique: Occlusion queries check visibility after rendering simplified proxies.
- Advantages: GPU processes data in parallel — much faster than CPU for complex scenes.
- Disadvantages: Creates some overhead for initial queries; requires careful implementation.

C — Hybrid Approaches

Modern engines like Unreal Engine 5 use hybrid approaches: CPU handles basic frustum culling and initial bounding-box tests, while the GPU handles detailed occlusion and depth testing.

- UE5 combines hardware occlusion queries + temporal occlusion culling + CPU-based initial culling.

Frustum Culling (CPU)

- Removes objects outside the camera's view frustum entirely.
- Works at the object level — cannot resolve per-pixel visibility conflicts.
- After frustum culling, visible objects proceed to the GPU for Z-buffer depth sorting.

Z-buffer (GPU)

- Performs precise per-pixel depth testing during rasterization.
- Handles partial occlusions where only parts of objects are visible.
- Manages self-occlusion within a single mesh.

Draw Call Generation (CPU)

The CPU generates draw calls for each visible object. Each draw call contains:

- Geometry data: Vertex buffers, index buffers.
- Material data: Shader programs, textures, uniforms.
- State changes: Render targets, blend modes, depth testing settings.

Anatomy of a Draw Call

1. Geometry Data

- Vertex buffers (positions, normals, UVs).
- Index buffers (defines how vertices form triangles).

2. State Changes

- Shaders (vertex, fragment, etc.).
- Textures, materials, and sampler settings.
- Blend modes, depth testing, and stencil configurations.
- Render target settings (e.g., where the output goes).

3. Transformation Data

- Object transformations (position, rotation, scale).
- Camera matrices (view and projection).

4. Other Information

- Light properties and environment settings.
- Post-processing effect parameters.

Why are Draw Calls Expensive?

CPU Overhead

The CPU must process and send each draw call to the GPU. Too many draw calls bottle-neck the CPU if it cannot keep up with the GPU's throughput.

State Changes

Each draw call may involve state changes (switching shaders, materials, render targets). State changes force the GPU to pause, reconfigure, and restart — causing inefficiencies.

Factors That Require Unique Draw Calls

- 1. Different Materials: Each material requires its own draw call.
- 2. Multiple Shaders: Unique shaders or variations require separate processing.
- 3. State Changes: Blending, depth testing, or stencil state transitions.
- 4. Unique Render Targets: Switching render targets forces a new draw call.
- 5. Separate Textures: Each unique texture (not in an atlas) triggers new draw calls.
- 6. Instancing Incompatibility: Non-instanced objects each require individual calls.
- 7. Different Meshes: Each unique mesh requires a separate draw call unless instanced.
- 8. Dynamic Shadows: Shadow mapping passes generate additional draw calls.
- 9. Transparency: Transparent objects must be sorted and rendered individually.
- 10. Post-Processing: Effects like bloom or SSAO trigger dedicated render passes.

Optimization Techniques for Draw Calls

1. Batch Rendering

Combine multiple objects with the same material into a single draw call.

- Merge static meshes that share a material using pre-computation tools.
- Dynamically group objects into batches during runtime.
- Benefit: Reduces draw calls by minimizing the number of separate state changes.

2. Mesh Instancing

Render multiple instances of the same mesh using a single draw call.

- Apply different transformations (position, rotation, scale) per instance.
- Share geometry and material data between all instances.
- Use Case: Ideal for foliage, rocks, debris, and repeated environmental assets.
- Benefit: Drastically reduces draw calls for repeated assets.

3. Texture Atlases

Combine multiple textures into a single large texture (atlas).

- Map UV coordinates for multiple objects into the atlas.
- Use the same material/shader for all objects sharing the atlas.
- Benefit: Reduces texture bindings (expensive state changes). Allows multiple objects per draw call.

4. Material Optimization

- Share materials between objects wherever possible.
- Use material instances to vary properties (e.g. color, roughness) without duplicating materials.
- Benefit: Fewer unique materials → fewer state changes → fewer draw calls.

5. Level of Detail (LOD)

- Dynamically switch between high and low LOD models based on camera distance.
- Benefit: Reduces vertex processing load. Improves performance for large scenes.

6. Merge Static Meshes

- Combine static objects that share a material into a single mesh at bake time.
- Benefit: Reduces draw calls for geometry that never moves.

7. Use Simplified Collision Meshes

- Use simplified or proxy meshes for physics/collision calculations.
- Avoid collision calculations for non-interactable objects.
- Benefit: Reduces collision-related overhead.

8. Cull Hidden Objects

- Frustum culling: remove objects outside the camera's view.
- Occlusion culling: skip objects blocked by other geometry.
- Benefit: Reduces unnecessary draw calls for invisible objects.

9. Multi-Draw Indirect (MDI)

Issue multiple draw calls in a single API call using a GPU-side buffer of commands.

- Prepare an array of GPU commands in memory instead of individual CPU draw calls.
- Benefit: Reduces CPU-GPU communication overhead significantly.

10. Use GPU Instancing APIs

- DirectX 12: Explicit resource and state control.
- Vulkan: Efficient batching with minimal driver overhead.
- Metal: Optimized for Apple silicon.
- Benefit: Low-level hardware access reduces draw call bottlenecks.

Monitoring and Tools for Draw Call Optimization

Unreal Engine

- Profiler: Monitor draw calls, material usage, and culling performance.
- Instanced Static Meshes: Enable for foliage and repeated assets.
- Shader Complexity View: Visualize expensive materials in the viewport.

Third-Party Tools

- RenderDoc: Capture and debug individual draw calls. Inspect pipeline state.
- NVIDIA Nsight: GPU debugging and performance profiling with hardware counters.

State Changes — Detailed Breakdown

1. Material Changes

Applying a new material forces the GPU to load and configure associated shaders, textures, and settings — triggering a new draw call.

2. Shader Changes

Switching between vertex, geometry, or fragment shaders requires a full pipeline reconfiguration — a new draw call.

3. Texture Bindings

Binding a new texture updates the GPU's texture samplers. This state change adds overhead and necessitates a separate draw call.

4. Render Target Changes

Writing to a different render target (e.g., switching from G-buffer to shadow map) forces the GPU to flush the pipeline and reconfigure — another draw call.

5. Blend Modes and Depth Settings

Adjusting blend modes (e.g., for transparency) or toggling depth/stencil testing requires a state change, triggering a new draw call.

6. Object-Specific Variations

Objects using the same material but differing in a property (e.g. a color variation via unique material parameters) each require a separate draw call unless material instancing is used.

Chapter 6: Frame Creation – GPU Role

Once the CPU has finished culling and generating draw calls, the GPU takes over. This chapter walks through every stage of the GPU rendering pipeline from geometry processing to the final lit pixel.

Deferred Rendering Pipeline Overview



Geometry Pass

The scene's geometry is processed and essential data — positions, normals, colors, and texture coordinates — is captured and stored in the G-buffer.

Vertex Shader

- Transforms vertices from model space → world space → clip space.
- Outputs attributes like normals, UVs, and colors for each vertex.
- Tessellation (Optional): Subdivides coarse geometry into finer details.
- Geometry Shader (Optional): Can add, remove, or modify geometry dynamically.
- Primitive Assembly: Transformed vertices assembled into triangles.
- Backface Culling: Removes primitives facing away from the camera (fixed hardware).

Vertex Splits

Edge splitting occurs during vertex processing to handle smoothing group boundaries, UV seams, material edges, or hard edges. A single logical vertex may split into multiple physical vertices.

Causes of Vertex Splitting

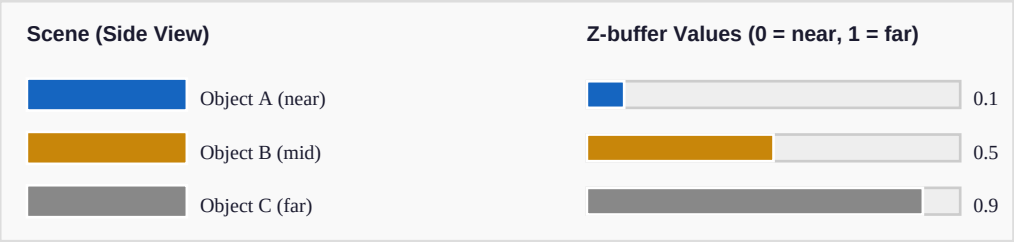
- Discontinuities in Normals — sharp edges need distinct normals per face.
- Multiple Materials — boundary vertices need separate physical vertices per material.
- UV Mapping Seams — UV seam vertices duplicated for different UV coordinates.
- Smoothing Groups — split at boundaries for correct shading transitions.
- Hard Edges / Custom Normals — split along hard edges for sharp normal transitions.
- GPU Optimization — triangle adjacency and cache alignment may require additional splits.

Rasterization

Primitives are converted into 2D fragments (potential pixels). Each fragment corresponds to a pixel in the render target and is passed to the fragment shader.

Z-buffer (Depth Buffer)

Z-buffer Depth Comparison — Objects at Different Depths



- Depth values normalized to range 0 (near clipping plane) to 1 (far clipping plane).
- Values distributed non-linearly — more precision allocated near the camera.
- If new fragment depth is closer than current Z-buffer value → replace it.
- If new fragment depth is farther → discard fragment (hidden surface removal).
- Precision: 16, 24, or 32-bit fixed/float texture in VRAM. Higher = better depth accuracy.

Applications of Z-buffer Values

- Hidden Surface Removal — ensures only closest surfaces are rendered.
- Post-Processing — depth of field, ambient occlusion, fog all read Z-buffer values.
- Shadow Mapping — depth info used for generating and applying shadow maps.
- Early-Z Testing — discards fragments before the fragment shader saves resources.

Fragment (Pixel) Shader

- Texture Sampling: Reads albedo (color), normal maps, roughness from G-buffer.
- Shading Model Calculations: Applies material attributes (metallic, roughness) for light interaction.
- Fragment-Specific Operations: Normal mapping, displacement mapping, UV-based effects.

G-buffer Storage

The G-buffer is generated containing multiple render targets that store all the surface data needed for the lighting pass. This is what makes deferred rendering work.

Common G-buffer Attributes

Attribute	Contents	Notes
Position	3D world/view position	Often reconstructed from depth + camera matrices in modern pipelines.
Normals	Surface normal at each pixel	Used for lighting and shading calculations.
Depth/Stencil	Z-buffer depth values	Can be shared or reconstructed.
Albedo	Base color without lighting	The diffuse color of the surface.
Specular	Specular reflectivity	Controls highlight intensity.
Roughness	Microsurface detail	Smooth (0) vs. rough (1) reflections.
Metallic	Metal vs. dielectric	Determines PBR reflection behavior.
Emissive	Self-emitting light value	Surfaces that glow without a light source.

Multiple Render Targets (MRT)

MRT is a graphics API feature allowing a single rendering pass to output to multiple textures simultaneously. All G-buffer attributes are written in a single geometry pass.

- Efficiency: Reduces the number of geometry passes required.
- Decouples geometry and lighting — geometry data stored first, lighting computed separately.
- Limitation: Most GPUs support 4–8 render targets simultaneously.
- Memory cost: MRT significantly increases VRAM usage at high resolutions.
- Transparency: Deferred rendering does not inherently handle transparent objects (needs a forward pass).

G-buffer Optimization

- Texture Packing: Store roughness, metallic, and AO in separate channels of one texture.
- Bit Depth Reduction: Use 16-bit for non-critical channels like roughness or metalness.
- Compression: Use BC formats for G-buffer storage where quality permits.

Lighting Pass

This step computes the lighting for each pixel using the data stored in the G-buffer. All lighting calculations happen here — not in the geometry pass.

Core Aspects of Lighting

Types of Lighting

- Direct Lighting: Light from a source directly to a surface — highlights, immediate illumination.
- Indirect Lighting: Light that bounced off surfaces — contributes to Global Illumination (GI).
- Ambient Lighting: Uniform base light approximating indirect illumination.

Properties of Light

- Color: Wavelength of light determines color (warm sunlight, cool fluorescent).
- Intensity: Brightness of the light source.
- Falloff: Diminishing intensity as light travels farther from the source.

Light Sources

- Point Light: Emits equally in all directions from a single point.
- Directional Light: Mimics sunlight — parallel rays, no specific origin.
- Spotlight: Cone-shaped emission with controlled angles and falloff.
- Area Light: Simulates a physical surface emitting light — produces soft shadows.

Step 1 — Collect Light Data

- Dynamic Light Sources: Gather position, direction, intensity, range, shadow properties.
- Static Lighting: Use precomputed lightmaps for static geometry — saves runtime cost.

Step 2 — Shadow Mapping

- Compute depth maps from each light's perspective (shadow cascades for directional lights).
- Depth maps identify which areas are in shadow based on occlusion.
- PCF (Percentage Closer Filtering): Smooth shadow edges by averaging multiple samples.
- VSM (Variance Shadow Maps): Statistical approach for efficient soft shadows.

- Cull unnecessary shadow calculations based on view frustum or distance.

Step 3 — Per-Light Calculations

Directional Lights

- Applied across the entire screen.
- Calculate light intensity and direction. Integrate with GI or ambient occlusion.

Point Lights

- Calculate illumination in a spherical range around the light source.
- Use a bounding volume to restrict processing to pixels within range.

Spotlights

- Compute a cone of influence — only pixels inside the cone receive light.
- Includes falloff and spot angle adjustments.

Area Lights

- Simulate light from a physical surface (e.g., a glowing panel or window).
- Approximated via precomputed textures or ray-traced solutions.

Step 4 — BRDF Shading (Bidirectional Reflectance Distribution Function)

BRDF uses PBR models to compute how light interacts with each surface:

- Diffuse Reflection: Light scattered uniformly. Calculated using albedo.
- Specular Reflection: Reflected highlight. Based on roughness, metallicity, and light angle.
- Fresnel Effect: Modulates reflectivity based on viewing angle (stronger at grazing angles).
- Advanced microfacet models consider surface microgeometry for specular accuracy.

Step 5 — Ambient Lighting & Indirect Illumination

- SSAO (Screen-Space Ambient Occlusion): Approximates small-scale light occlusion in crevices.
- Lumen GI: Real-time ray tracing or surface cache for indirect lighting (UE5).
- Voxel GI: Voxelizes the scene and computes indirect light contributions per voxel.

Step 6 — Shadow Application

Shadows are combined with lighting calculations. Shadow results from depth maps or ray-tracing are multiplied with diffuse and specular contributions to darken occluded areas.

Step 7 — Additive Blending

- Lights are rendered additively — each light's contribution blended per pixel.
- Each light is processed independently in its own pass and accumulated.

Step 8 — Post-Processing for Lighting

- Bloom: Add glow around bright areas above a threshold.
- Lens Flares: Simulate optical artifacts from bright light sources.
- Tonemapping: Map HDR lighting data to displayable LDR range (e.g. ACES).
- Color Grading: Adjust final color balance, contrast, and saturation.

Chapter 7: Lighting Techniques & Unreal Engine 5

This chapter covers the full spectrum of lighting techniques used in modern real-time rendering, from fundamental concepts through to UE5's Lumen system and advanced screen-space methods. Chapters previously covering these topics separately have been merged here to avoid repetition.

Fundamentals

- Real-Time Lighting: Calculated dynamically every frame. Used in video games.
- Precomputed Lighting: Baked into lightmap textures for static scenes. Faster, but inflexible.
- Forward Rendering: Lights calculated per object — simpler but inefficient with many dynamic lights.
- Deferred Rendering: Lights calculated after geometry pass — enables large numbers of dynamic lights.

Advanced Lighting Techniques

Ambient Occlusion (AO)

Approximates how light is occluded in crevices, adding depth by darkening corners and edges.

- SSAO (Screen-Space AO): Fast screen-space approximation using depth and normal info.
- HBAO (Horizon-Based AO): Higher quality than SSAO, better handles surface details.
- Ray-Traced AO: High-quality method tracing rays dynamically — computationally expensive.

Ray Tracing

Simulates the physical behavior of light for realistic reflections, shadows, and GI.

- Ray Traced Reflections (RTR): Traces rays from camera to find reflective surfaces.
- Ray Traced Shadows (RTS): Soft, accurate shadows based on geometry and light size.
- Ray Traced GI (RTGI): Simulates light bouncing in real-time. Highest quality, most expensive.

Reflection Probes & Cube Maps

Precomputed data structures storing environment reflections. Six-sided cube maps capture the scene from a fixed point. Used where full ray tracing is too expensive.

Light Propagation Volumes (LPV)

Stores light information in a 3D grid that propagates through the environment. A real-time GI approximation that trades accuracy for performance — good for dynamic scenes.

Volumetric Lighting

- Volumetric Fog: Simulates light diffusing through fog, smoke, or dust.
- Light Shafts / God Rays: Visible beams of light through atmospheric conditions.

Distance Field Shadows

Uses Signed Distance Fields (SDF) to represent geometry. Enables soft, accurate dynamic shadows for complex and dynamic objects. Higher memory cost than shadow maps.

Deferred Decals

Applied after lighting calculations to add surface details (dirt, damage) in a deferred pipeline. Screen-space decals blend with the G-buffer without re-rendering geometry.

Lightmass (Baked Lighting)

Unreal's GI solution for static environments. Precomputes and stores indirect light in lightmaps. High quality but cannot handle dynamic objects or changing light.

Planar Reflections

Render reflections for flat surfaces (water, mirrors) using a dedicated render pass. More performant than full ray tracing for flat reflective surfaces.

Global Illumination

GI simulates how light bounces throughout an environment — both direct and indirect light interactions. Without GI, scenes have harsh unrealistic shadows and flat lighting.

Components of GI

- Diffuse Inter-reflection: Light bouncing off surfaces illuminating indirect areas.
- Specular Reflection: Reflective light off shiny surfaces — mirrors, water.
- Ambient Occlusion: Soft shadows in crevices where light can't easily reach.
- Caustics: Light patterns formed by curved surfaces focusing light (e.g. glass of water).

Methods of Implementing GI

Method	Type	Trade-off
Ray Tracing	Dynamic	Most accurate. High computational cost.
Radiosity	Pre-computed	Static scenes only. Good for architecture.
Photon Mapping	Pre-computed	Traces photon hits from light sources. Offline rendering.
Voxel GI (Lumen)	Dynamic	Real-time friendly. Uses voxel grid + ray tracing.
SSGI	Screen-Space	Cheapest. Limited to visible content.
Baked Lightmaps	Pre-computed	Fastest at runtime. No dynamic objects.

Lumen (Unreal Engine 5)

Lumen is UE5's fully dynamic global illumination and reflections system. It does not rely on baked lightmaps or pre-computed data — everything updates in real time.

Scene Representation for Lighting

- Distance Fields: Objects represented as volumetric distance fields capturing shape and spatial relationships.
- Voxelized Scene Data: Scene divided into a 3D voxel grid for localized light propagation.
- Surface Cache: Surface details (normals, material properties) cached for precise calculations.
- Screen-Space Data: Visible pixels from the G-buffer used to approximate local lighting.

Lumen — Direct Lighting

- Updates direct lighting in real time — reacts immediately to changes in light source position, intensity, or color.
- Traces rays from light sources through scene using G-buffer data (normals, depths).
- Calculates both diffuse and specular reflections for direct lighting on surfaces.

Lumen — Indirect Lighting

- Traces light in real time using surface cache tracing — no baked lightmaps needed.
- Supports color bleeding — indirect light carries color from one surface to nearby objects.
- Updates dynamically when objects move or light sources change.

Lumen — Reflections

- Screen-Space Reflections (SSR): For reflective materials visible on screen (water, glass).
- Ray-Traced Reflections: For accurate off-screen reflections using hardware ray tracing.
- Reflections integrate with GI — they reflect indirect light, not just direct sources.
- Dynamic updates — moving objects or changing materials update reflections in real time.

Advanced Lumen Features

- Global Illumination Tracing: Multiple techniques — surface cache + voxel ray tracing.
- Voxelization and Cache Management: Dynamic cache updated as scene changes.
- Lighting Volume Optimization: Scene divided into lighting volumes for efficient calculations.
- Temporal and Spatial Filtering: Smooths out noise in indirect lighting and reflections.
- Real-Time Adjustments: Light sources, materials, and geometry changes reflected instantly.

Lumen Performance Considerations

- Real-time GI and reflections are computationally heavy — especially in complex scenes.
- Uses clustering, caching, and spatial filtering to improve efficiency.
- Quality settings available to balance visual fidelity vs. performance.
- Designed for next-gen hardware (PS5, Xbox Series X, high-end PCs).

Shadow Casting in Unreal Engine

Shadow Maps

Unreal generates a depth texture from the light's perspective. During shading, the fragment's depth is compared to the shadow map to determine if it's in shadow.

- Used for directional lights, spotlights, and point lights.
- Limitation: Shadow acne and peter-panning artifacts at low resolutions.

Distance Field Shadows

Uses signed distance fields for soft, accurate dynamic shadows. Supports shadow casting for distant objects and complex assets.

- Higher memory cost due to distance field storage.
- Lumen can leverage distance fields to enhance indirect lighting accuracy.

Screen-Space Shadows (SSSS)

- Computes shadows in screen space using the G-buffer.
- Efficient for small-scale details near the camera.
- Limitation: Cannot account for off-screen geometry.

Ray-Traced Shadows

- Most realistic — traces rays to simulate accurate light/shadow interactions.
- Supports soft shadows based on light size and distance.
- Highest computational cost — used for cinematic-quality rendering.

Advanced Shadow Optimization

- Cascaded Shadow Maps (CSM): Splits shadow map into cascades per distance range. Higher quality near camera.
- Virtual Shadow Maps (VSM): UE5 solution for Nanite. High-quality per-pixel shadows without CSM artifacts.
- Shadow Bias Adjustments: Fine-tuning to reduce shadow acne and peter-panning.

Screen-Space Techniques

Screen-space techniques operate within the screen's viewport, leveraging the already-rendered image. Fast because they require no information outside the current frame.

Technique	Purpose	Limitation
SSR (Reflections)	Simulates reflections using current frame pixels.	Cannot reflect off-screen geometry.
SSAO	Approximates soft shadows in crevices/corners.	Approximate — can create artifacts.
SSGI	Cheaper GI approximation in screen space.	Limited to visible content only.
Screen-Space Shadows	Shadow effects via depth buffer analysis.	Cannot handle off-screen occluders.
SSDOF (Depth of Field)	Simulates camera focus/blur effects.	Can look incorrect near occlusion boundaries.
Motion Blur	Blends pixels between frames to simulate motion.	Can look wrong with fast rotations.
Bloom	Glows around bright pixels above threshold.	Can over-saturate if threshold is low.
Lens Flares	Optical lens artifact simulation.	Purely cosmetic — no physical basis.

Chapter 8: Advanced Deferred Rendering Techniques

This chapter covers advanced optimizations and techniques within the deferred rendering pipeline, focusing on how to get the most out of modern GPU hardware in Unreal Engine.

Multiple Render Targets (MRTs)

MRTs allow the GPU to output data to several textures simultaneously during G-buffer creation. A single draw call writes normals, depth, albedo, and specular all at once.

- Benefits: Reduces the need for multiple rendering passes, saving GPU processing and bandwidth.
- Limit MRTs to essential attributes only — each extra target costs memory and bandwidth.
- Use lower bit-depth textures for non-critical data (e.g., 8-bit for some buffers).
- Avoid overdraw and redundant writes to the same render target.

Clustered and Tiled Lighting

Divides the screen into tiles or 3D clusters in view space. Each tile/cluster processes only lights within its bounds — far more efficient than processing all lights for all pixels.

- Benefits: Handles large numbers of dynamic lights more efficiently than classic deferred.
- Ensure light volumes fit tightly to reduce unnecessary processing per tile.
- Use sparse clusters for less densely lit areas.
- Combine with shadow light culling to exclude lights not visible in the scene.

Shader Complexity Reduction

Simplified and optimized material shaders reduce GPU load. Use Unreal's Shader Complexity view to visualize expensive shaders in the viewport.

- Minimize texture lookups in shaders — each sample costs TMU bandwidth.
- Use static switches to avoid dynamic branching (branch divergence kills warp efficiency).
- Utilize shared samplers to reuse texture fetches across shader stages.
- Optimize math-heavy calculations with precomputed values or simpler approximations.
- Replace high-cost procedural noise with a pre-baked texture.

Temporal Techniques

Temporal Anti-Aliasing (TAA)

Smoothens jagged edges by reusing data from previous frames — accumulating samples over time. Also denoises effects like SSR and shadows.

- Adjust jitter patterns and sample accumulation to avoid ghosting on fast-moving objects.
- Proper motion vectors are critical for correct temporal reprojection.

Temporal Super Resolution (TSR)

Reconstructs high-quality frames from lower-resolution inputs. Allows rendering at 50–75% resolution and upscaling to native with minimal quality loss.

- Ensure proper motion vector generation for accurate temporal data.
- UE5's TSR is Unreal's native implementation — similar to DLSS/FSR conceptually.

Nanite Virtualized Geometry

Nanite replaces traditional polygon rendering for complex static meshes. It handles billions of triangles with automatic LOD generation, streaming, and culling.

- Eliminates manual LOD creation and reduces draw call overhead for high-detail assets.
- Use for static meshes with high geometric complexity (hero props, architecture, rocks).
- Avoid enabling Nanite for simple objects or frequently animated meshes.
- Optimize cluster sizes and distances for desired performance/visual trade-offs.

Virtual Shadow Maps (VSM)

A shadowing solution tailored for Nanite meshes, providing high detail at low overhead. Avoids the need for cascading shadow maps on Nanite objects.

- Handles fine shadow detail at near distances efficiently.
- Use shadow filtering to improve visual quality.
- Limit shadow draw distances for non-critical distant objects.

Post-Processing Pipeline Optimization

Bloom

- Adjust thresholds and intensity to avoid over-saturating the scene.
- Use resolution-scaled bloom to save performance — bloom can run at half resolution.

Motion Blur

- Apply selectively (per-object motion blur) to reduce GPU costs.
- Full-screen motion blur is expensive — consider turning it off on mobile.

Tone Mapping

- ACES tone mapping for cinematic visuals without impacting real-time performance.
- Avoid excessive exposure adjustments that wash out the GI.

Ray Tracing Integration

A hybrid approach combining ray tracing for selective high-quality effects with deferred rasterization for the majority of rendering.

- Use denoising techniques to reduce sample counts (NVIDIA DLSS RR, AMD FSR).

- Limit ray-traced effects to high-priority surfaces or camera-proximity objects.
- Combine with screen-space techniques to balance quality and performance.
- Ray-traced effects to prioritize: reflections, shadows, GI. Avoid RT for everything.

Compute Shaders for Custom Passes

Offload heavy calculations — particle simulations, custom effects, preprocessing — to compute shaders for fully parallel GPU execution.

- Compute-based fluid simulations.
- G-buffer preprocessing or custom lighting passes.
- Minimize memory read/write operations — prefer shared memory within thread groups.
- Use thread groups optimized for the target GPU architecture's warp/wavefront size.