Chapter Outline

Unreal Engine (UE): From Object State to Images and Interaction

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.2.1 Framework and Rendering Pipeline
Textbook: Sec. 3.1.4.1

3.2.2 3D Asset Management
Textbook: Sec. 3.1.4.2

3.2.3 Lighting and Rendering
Textbook: Sec. 3.1.4.3

3.2.4 Physics and Animation
Textbook: Sec. 3.1.4.4

3.2.5 Simulation Interaction Framework
Textbook: Sec. 3.1.4.5

3.2.1 Framework and Rendering Pipeline: Three Tiers

UE Turns Object States into Images Through Three Tiers

Simulation Interaction Framework: Actors, controllers, events, and Tick determine each object's state for this frame.

Core Engine Modules: Unreal Engine (UE) manages models, materials, animation, and physics resources. Its rendering pipeline converts 3D scenes into 2D images in real time.

Platform interface: the Render Hardware Interface (RHI) converts common rendering commands into calls to graphics application programming interfaces (APIs), such as DirectX, Vulkan, and Metal.

The interaction framework sets states, the engine renders them, and the RHI passes commands to the graphics card.

3.2.1 Framework and Rendering Pipeline: Scene Objects

Start with Levels, Actors, and Components

Level: organizes scene objects and their placement, such as ground, buildings, lights, and a box.

Actor: an object instance that can be placed in a Level. A box Actor can move, rotate, and run its own logic.

Component: a functional module owned by an Actor, such as a static mesh, camera, or collision shape;
Components have different roles.

In this example, the box Actor references the Cube mesh asset through a StaticMesh Component. Moving the instance does not modify the mesh file.

A Level organizes entities; an Actor combines Components; each Component provides a capability.

3.2.1 Framework and Rendering Pipeline: Ownership and Attachment

The Root Component Supplies the Actor's Transform

Box Actor: owns Components

Root：SceneComponent

Attachment

StaticMesh Component
References the Cube mesh

Root Component: a SceneComponent provides position,
rotation, and scale; the Actor uses its transform.

Child Component: stores a transform relative to its parent.
When the parent moves, the attached child updates.

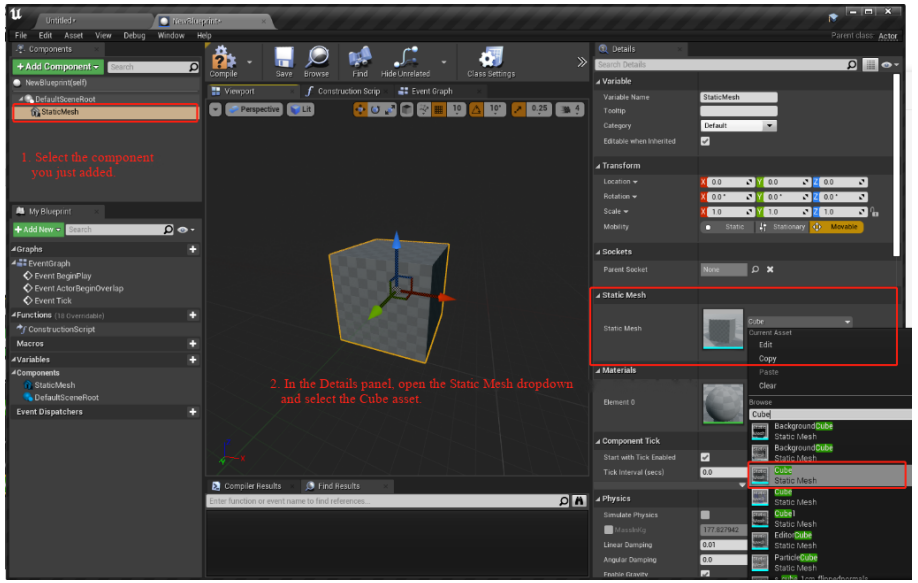
General Component: an ActorComponent may hold only logic,
such as state management, without a spatial position.

Spatial Components can form parent–child relationships; logic Components may have no position.

3.2.1 Framework and Rendering Pipeline: Objects in the Editor

A Real Box: Its Component References the Cube Mesh

Box Blueprint components (UE4.27.2)



Component tree: DefaultSceneRoot→StaticMesh



Properties: choose Cube in the Static Mesh field; set the Component transform under Transform.

Selected mesh →



Run: compile and save the Blueprint, drag it into the Level, then click Play to observe the instance.

Component settings configure the instance; the chosen mesh asset determines its visible shape.

3.2.1 Framework and Rendering Pipeline: Objects and Exchanged Data

Use a Drone to Distinguish Objects from Exchanged Data

Object / data	Drone example	Role
Level	Flight-test scene	Organizes terrain, buildings, lights, and multiple entities
Actor	One drone	Placeable, movable, identifiable scene entity
Component	Mesh, collision shape, camera	Appearance, contact boundaries, or an observation source
Interaction data	Position, attitude, control input	External models or Blueprints read, update, and verify entity behavior

Position, attitude, and control inputs are data, not an additional object tier.

3.2.1 Framework and Rendering Pipeline: Events and Tick

Events Set State; Tick Updates It over Time

BeginPlay: initialize at object startup, e.g., set the initial angle and target speed.

Input event: update the speed variable on a new command.
Unchanged commands need no repeated setup.

Tick: runs when enabled, following its schedule. DeltaSeconds is this update's interval in seconds.

Time scaling: speed × time gives displacement; angular speed × time gives an angular increment. Fixed steps per frame make speed depend on frame rate.

Example: $\omega = 90°/\mathrm{s}$ and $\Delta t = 1/60\ \mathrm{s}$, this frame rotates by $\Delta\theta = \omega\,\Delta t = 1.5°$. Doubling the interval gives 3° per frame at the same 90°/s.

Store a new target when triggered; use the elapsed time for continuous changes.

3.2.1 Framework and Rendering Pipeline: Pipeline Overview

The CPU Prepares Each Frame; the GPU Generates the Image

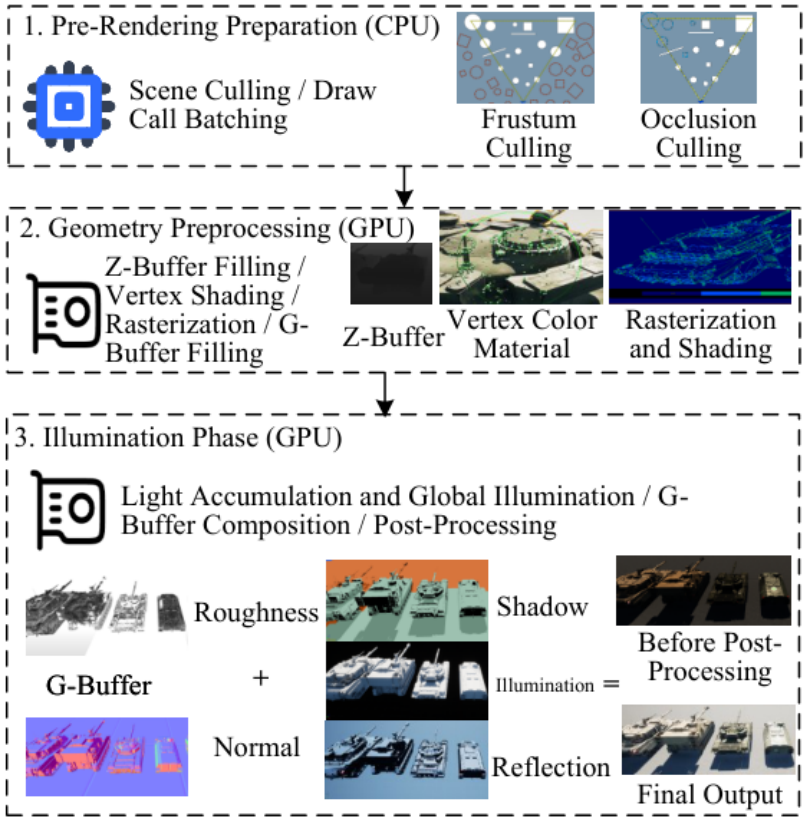


Fig. 3.4 Rendering Pipeline Process

Central processing unit (CPU): updates object states, selects what to draw, and submits rendering work.

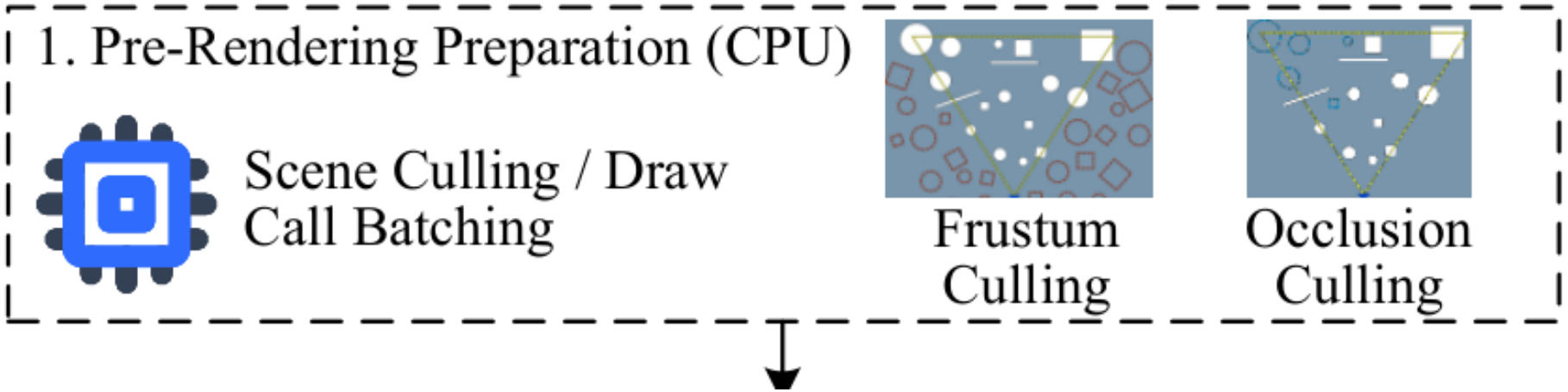
Graphics processing unit (GPU): processes geometry and surfaces, computes lighting and post-processing, and outputs the image.

Read in order: pre-render preparation → geometry preprocessing → lighting. Each stage is enlarged next.

3.2.1 Framework and Rendering Pipeline: Culling

Distance, Frustum, and Occlusion Tests Reduce Drawing Work

Culling: use distance, view-frustum, and occlusion tests to exclude objects that need not be drawn.



Occlusion tests: run on the CPU or GPU, depending on the method. Objects outside the view can still update their motion.

Fig. 3.4 (detail) Visibility Preparation and Object Selection

Exclude invisible objects so the rendering budget serves visible surfaces.

3.2.1 Framework and Rendering Pipeline: Culling

Which Objects Enter the Image as the Camera Moves?

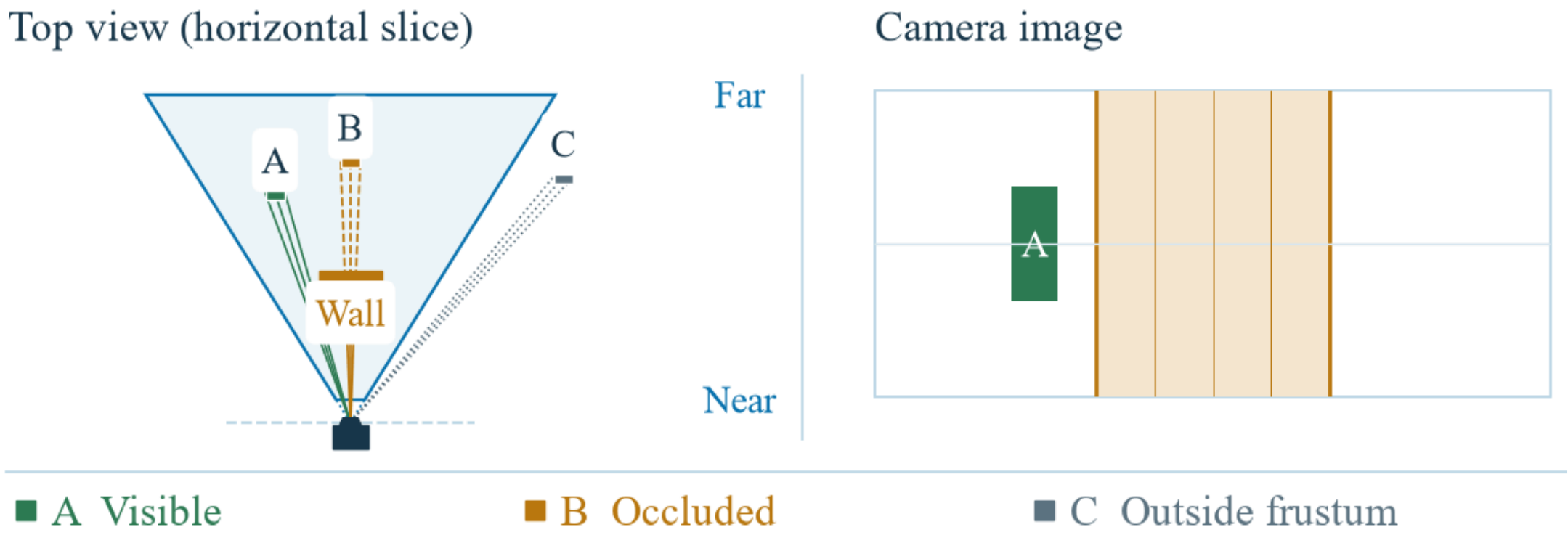


Figure: horizontal frustum slice (left) and the same camera's image (right): outside, occluded, and visible objects.

3.2.1 Framework and Rendering Pipeline: Draw Calls and Batching

Batching Cuts Draw Calls; PSOs Cut State-Change Overhead

Draw call: each CPU drawing command specifies vertex buffers, index buffers, and material state. More calls increase CPU work.

Batching and instancing: submit compatible objects sharing a mesh and material together. If 120 objects remain after culling but use only 4 mesh–material combinations, calls could ideally fall from about 120 to about 4.

Pipeline State Object (PSO): precompile and cache a complete configuration, including shader bindings, input assembly, and blending, for direct use at runtime with less state-change overhead.

Batching conditions: different materials or translucent objects requiring individual sorting are harder to batch. Actual call counts depend on scene organization.

Culling reduces objects to draw; batching and PSOs reduce submission and state-change costs.

3.2.1 Framework and Rendering Pipeline: Rasterization and Depth

Geometry Processing Finds the Surface at Each Pixel

Rasterization: cover pixels with projected triangles and interpolate each pixel's normal and texture coordinates.

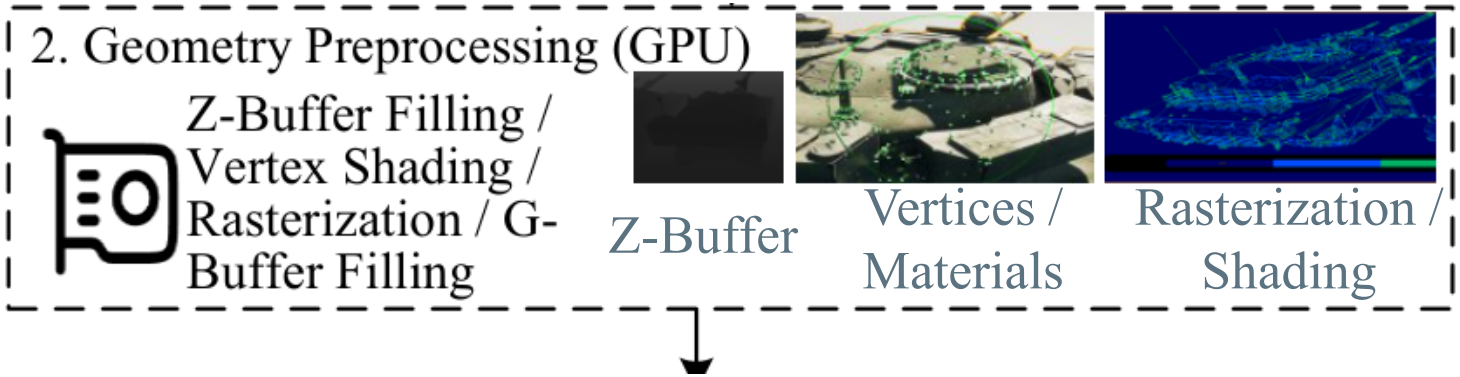


Fig. 3.4 (detail) Geometry Preprocessing (GPU)

Depth buffer (Z-Buffer): stores the nearest surface's encoded depth at each pixel to determine occlusion.

Early depth testing (Early-Z): compare depth first; reject farther surfaces without shading.

Depth determines visibility; normals and material parameters support lighting.

3.2.1 Framework and Rendering Pipeline: Rasterization and Depth

Which Surface Occludes the Other at the Same Pixel?

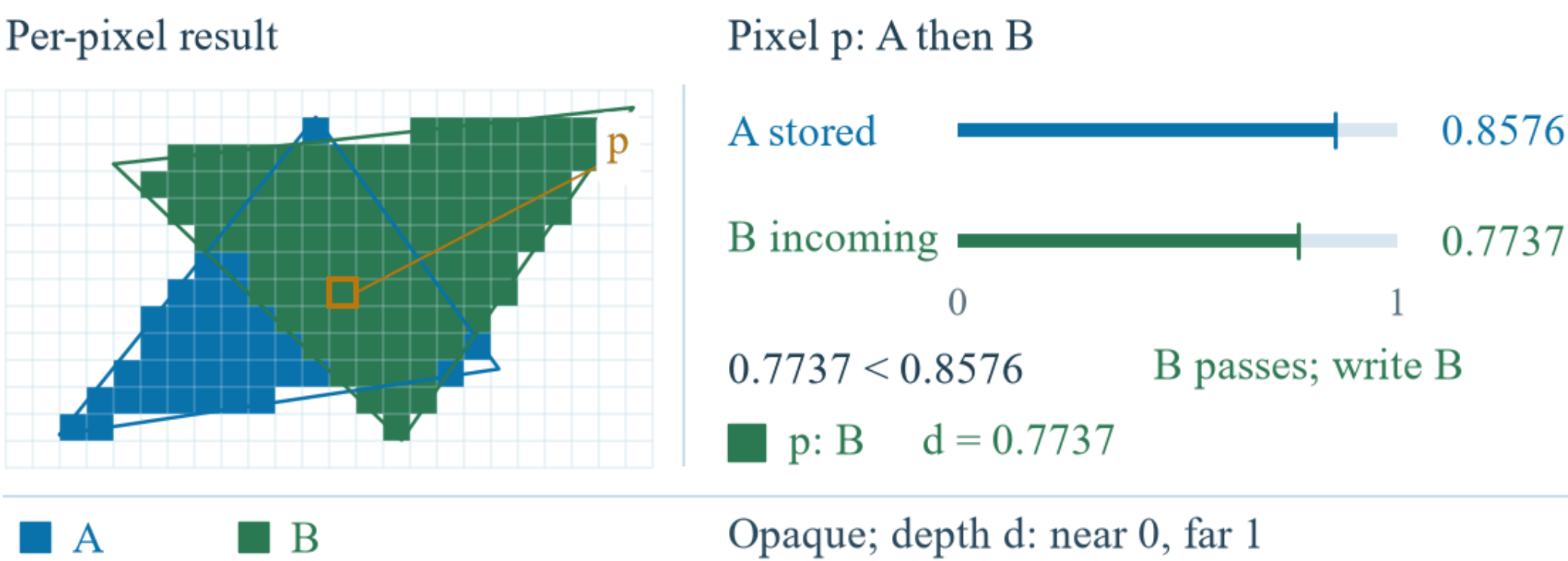


Figure: opaque surfaces compare depth per pixel; the nearer surface's color is retained.

3.2.1 Framework and Rendering Pipeline: Deferred Rendering

Deferred Rendering Stores Surfaces Before Computing Lighting

Geometry and materials
Identify visible surfaces
➡
G-Buffer
Stores surface properties
➡
Lighting
Combines surfaces and lights

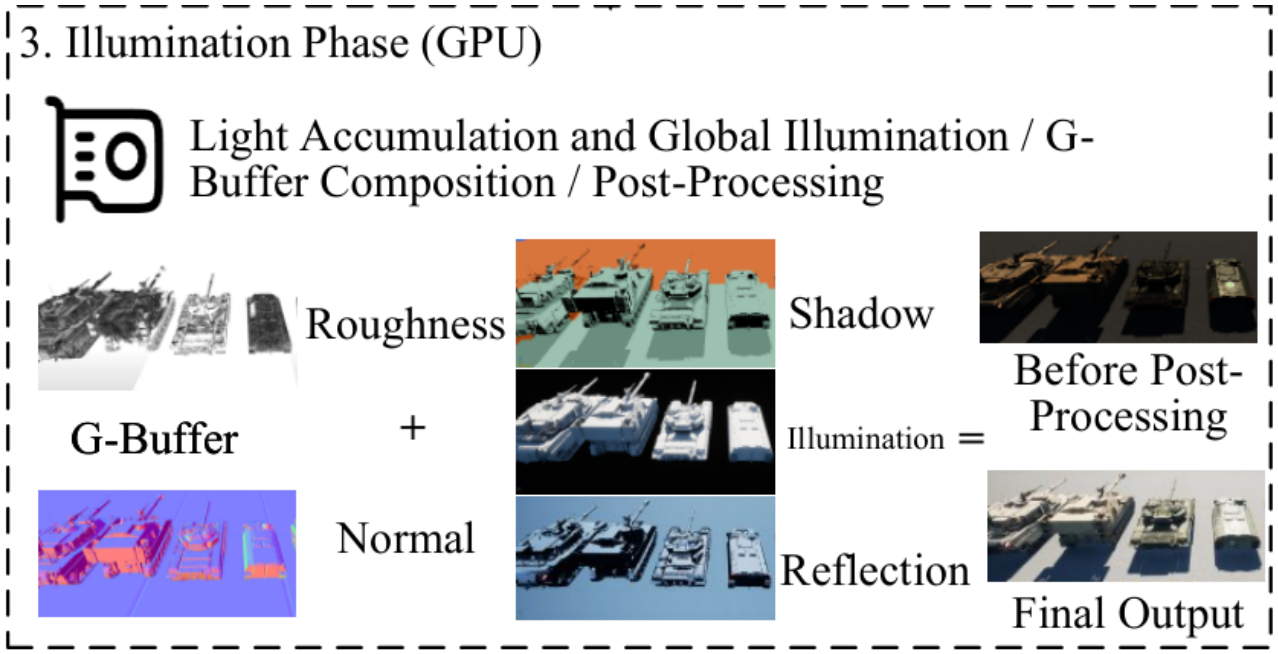
Geometry Buffer (G-Buffer): buffers store normals, base color, roughness, and other surface attributes for deferred rendering. Depth usually has a separate buffer.

Why defer lighting: evaluate each light only at visible screen pixels for efficient many-light scenes. Forward rendering computes lighting as each surface is drawn.

Surface buffers hold intermediate attributes; lighting and post-processing still determine the final color.

3.2.1 Framework and Rendering Pipeline: Lighting and Post-Processing

After Lighting, Post-Processing Changes the Final Pixels



Surface data: normals, roughness, and other properties come from the G-Buffer.

Lighting: combine surface properties, lights, and shadows; add lighting and reflections.

Post-processing: exposure, tone mapping, and other effects change final pixels without moving geometry.

Fig. 3.4 (detail) Lighting Stage (GPU)

Fixed geometry can still change brightness; exposure and post-processing also change the image.

3.2.1 Framework and Rendering Pipeline: Frame-Time Example

Use Frame Time to Choose What to Optimize

Frame rate and frame time: f is frames per second (FPS); the time available per frame is $T = 1000/f$ ms. At 30 FPS it is about 33.3 ms; at 60 FPS, about 16.7 ms.

Read metrics: run the Level in the UE editor, press ~ to open the console, and enter stat unit. Read Frame, Game (game thread), Draw (render thread), and GPU times in milliseconds.

Example assumptions: Game ≈9 ms, Draw ≈6 ms, GPU ≈28 ms. These stages pipeline different frames. Frame time is near the slowest stage, ≈28 ms (≈36 FPS), rather than 43 ms. Check GPU work first.

Frame-rate cap: t.MaxFPS 30 limits the maximum to 30 FPS; actual performance may be lower. It does not set camera sampling or control frequency.

The CPU and GPU can work in parallel pipelines; their frame times do not simply add.

3.2.1 Framework and Rendering Pipeline: Finding Bottlenecks

When Frame Time Rises, Identify the Work Responsible

High Game time: check Actor Tick updates, repeated logic, or communication handling. Put one-time work in the appropriate event.

High Draw time: check drawing requests and object organization. Batching or instancing requires compatible assets and materials.

High GPU time: check pixel count, materials, shadows, and post-processing. Level of detail (LOD) can reduce geometry rendering costs.

Comparison: fix the camera and scene, then change one setting. Record Frame time while checking whether target edges remain clear.

Change one main condition at a time; use frame time and image quality to explain the benefit.

Chapter Outline

Unreal Engine (UE): From Object State to Images and Interaction

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.2.1 Framework and Rendering Pipeline
Textbook: Sec. 3.1.4.1

3.2.2 3D Asset Management
Textbook: Sec. 3.1.4.2

3.2.3 Lighting and Rendering
Textbook: Sec. 3.1.4.3

3.2.4 Physics and Animation
Textbook: Sec. 3.1.4.4

3.2.5 Simulation Interaction Framework
Textbook: Sec. 3.1.4.5

3.2.2 3D Asset Management: Import Pipeline

Import External Models as UE Assets, Then Reference Them

Digital content creation (DCC) tools: model, unwrap texture coordinates, and create skeletal animations in Blender, Maya, 3ds Max, or similar software.

Import retention: format and importer support determine what survives. Verify materials, texture connections, and animation individually.

Assets and instances: imported .uasset files are managed in the Content Browser. Components reference assets; one asset can supply many instances.

Imported models become assets; instances in a Level reference assets rather than copying them.

3.2.2 3D Asset Management: Geometry Detail

LOD and Nanite Match Geometry Detail to Screen Size

LOD: prepare several meshes from fine to coarse for one model. Switch according to screen size, using fewer triangles at a distance.

Worked example: projected height is approximately proportional to $1/d$, and pixel count to $1/d^2$. When d rises from 10 m to 100 m, pixel count falls to about 1/100.

Nanite (virtualized geometry): UE5 automatically selects detail according to screen-pixel demand. It can reduce manual LOD work for detailed scanned models.

Check: follow a fixed camera path, inspect door and window edges at LOD transitions, and compare frame time. Assess clarity and cost separately.

LOD and UE5's Nanite control geometric detail and rendering work according to screen demand.

3.2.2 3D Asset Management: Geometry Detail

Distant Objects Need Less Detail on Screen

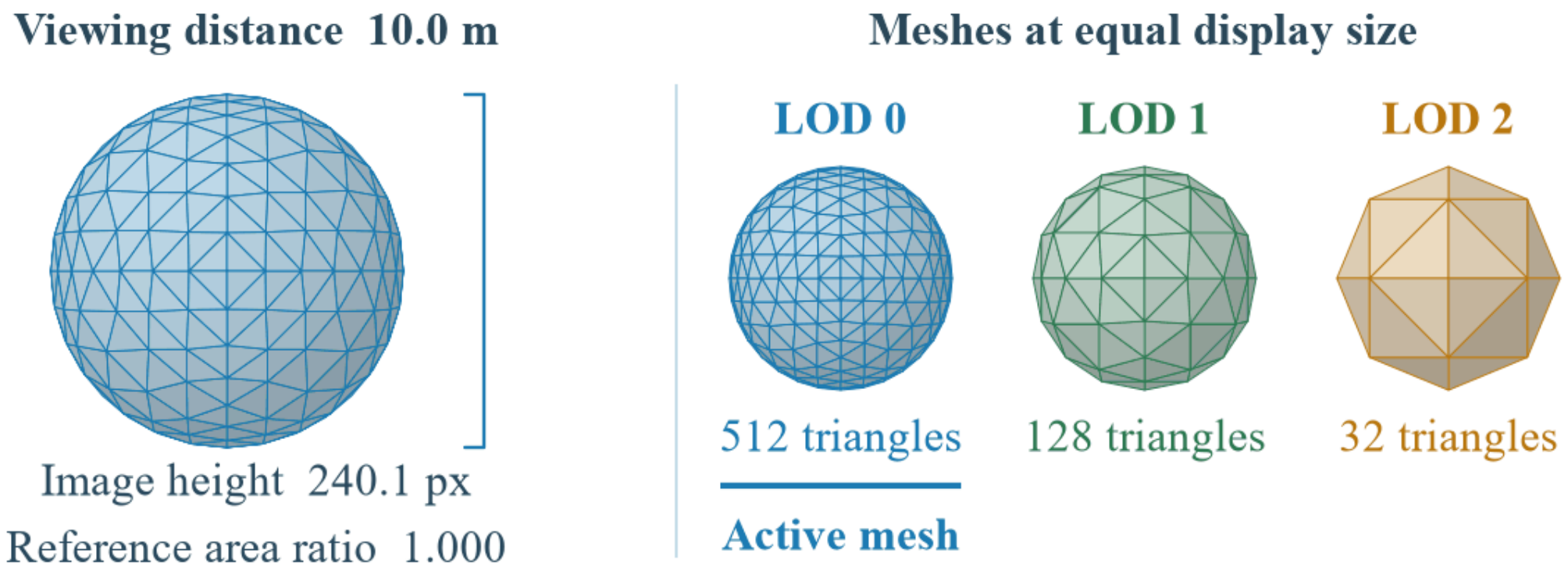


Figure: switch meshes by projected size at a fixed field of view; equal-size views at right reveal mesh structure.

3.2.2 3D Asset Management: UVs and Mipmaps

UV Places Textures; Mipmaps Reduce Distant Flicker

Texture coordinates (UV): map a surface to a 2D texture, with U horizontal and V vertical. One tile here spans 0–1; ordinary UVs can extend outside it for tiling.

Worked example: in a 1024×1024 texture, $(u, v) = (0.25,\ 0.5)$ maps to pixel coordinates $(uW,\ vH) = (256,\ 512)$.

Mipmaps: precompute progressively smaller textures. If one 1024-texel span covers 128 screen pixels, estimate mip level $\log_2(1024/128) = 3$.

Texture filtering: trilinear filtering interpolates between adjacent mip levels; anisotropic filtering keeps oblique ground and runways clear.

Match texture sampling to viewing scale to reduce flicker on distant and oblique surfaces.

3.2.2 3D Asset Management: Material Inputs

How Material Parameters Change Surface Lighting

Node-based Material Editor: connect texture samples and constants to Base Color, Metallic, Specular, Roughness, Ambient Occlusion, Normal, and other inputs.

Reflection distribution: higher Roughness broadens highlights and spreads reflections; Metallic changes the reflection components.

Specular and Ambient Occlusion (AO): the default Specular value of 0.5 gives about 4% specular reflectance. AO darkens crevices and corners.

Normal map: changes local shading normals to suggest fine relief, while mesh silhouettes and collision boundaries stay the same.

At runtime: material instances can change parameters; virtual textures load large textures on demand.

Material parameters change reflection; the relevant models determine geometry and collision boundaries.

3.2.2 3D Asset Management: PBR Parameters

Which Lighting Features Do PBR Parameters Change?

Physically based rendering (PBR): use material parameters to describe how surfaces respond to light.

Parameter	What it changes	What to inspect
Base Color	The surface's base color	Overall brightness and color under the same lighting
Metallic	Whether the surface reflects as a metal	Highlight and environment-reflection characteristics
Roughness	Whether reflections are concentrated or spread	Sharpness of specular highlight edges
Normal	Local apparent relief without changing the silhouette	How fine texture changes local lighting

3.2.2 3D Asset Management: Roughness Comparison

Official Comparison: Roughness Spreads Reflections

Each row, left to right: Roughness increases from 0 to 1

Nonmetal

Metal

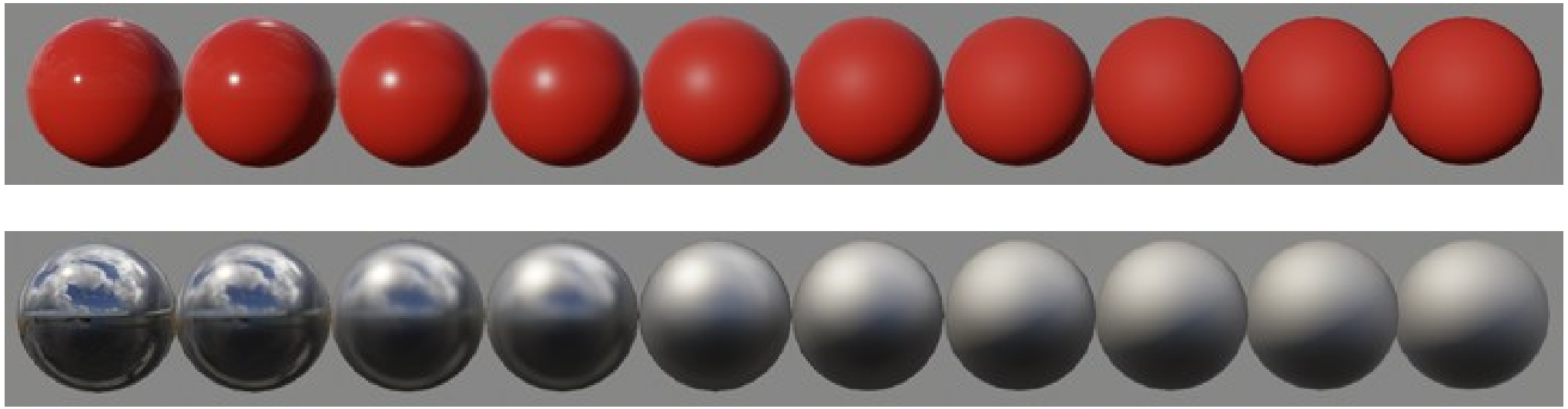


Figure: Epic's official material examples. Compare highlights and environment reflections within each row.

The silhouette stays the same; higher Roughness broadens highlights and reflections.

3.2.2 3D Asset Management: Terrain

Landscape Builds Editable Terrain from Grayscale Heightmaps

Landscape: generate 3D terrain from a grayscale heightmap, add material layers, create roads and rivers, and adjust detail automatically with viewing distance.

Heightmap: each pixel stores a 16-bit height (0–65535), with 32768 at zero height. At Z scale 100, the range is approximately −256 m to +256 m.

$$h = \frac{v - 32768}{128} \cdot \frac{s_z}{100}\ \text{m}$$

Worked example: $v = 40960,\ s_z = 100 \Rightarrow h = (40960 - 32768)/128 = 64\ \text{m}$

Data source: geographic elevation data, such as a digital elevation model (DEM), can be converted into an importable heightmap.

A mismatch in elevation zero or Z scale distorts the imported terrain's height differences.

3.2.2 3D Asset Management: Large-World Streaming

World Partition Loads Cells Around a Streaming Source

Load cells that intersect the circle

16 cells loaded

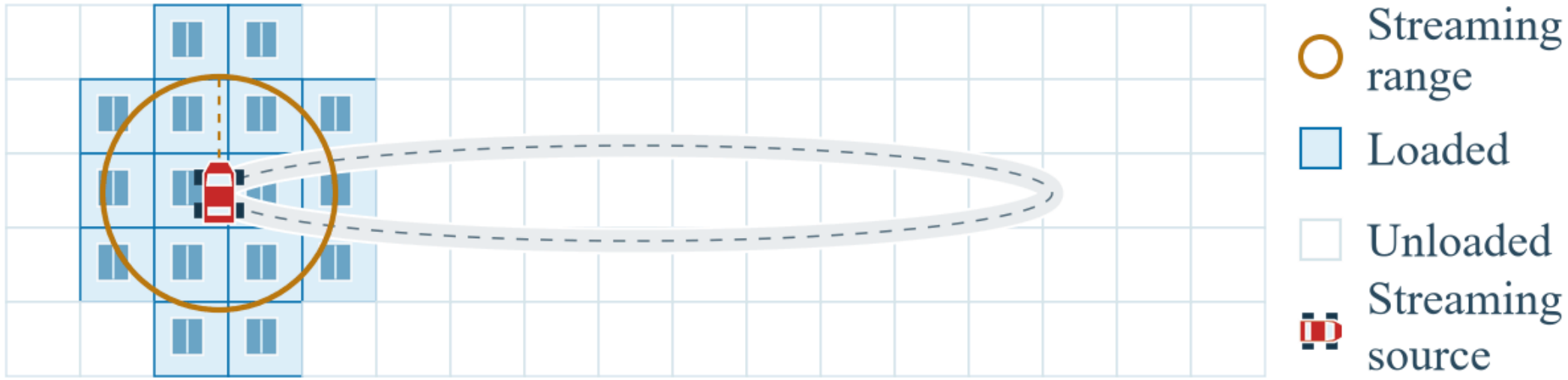


Figure: grid-cell loading and unloading as a streaming source moves (teaching illustration)

Load nearby cells and unload distant ones; an object's own settings also determine whether it stays loaded.

Chapter Outline

Unreal Engine (UE): From Object State to Images and Interaction

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.2.1 Framework and Rendering Pipeline
Textbook: Sec. 3.1.4.1

3.2.2 3D Asset Management
Textbook: Sec. 3.1.4.2

3.2.3 Lighting and Rendering
Textbook: Sec. 3.1.4.3

3.2.4 Physics and Animation
Textbook: Sec. 3.1.4.4

3.2.5 Simulation Interaction Framework
Textbook: Sec. 3.1.4.5

3.2.3 Lighting and Rendering: Direct and Indirect Light

Direct and Indirect Light Follow Different Paths

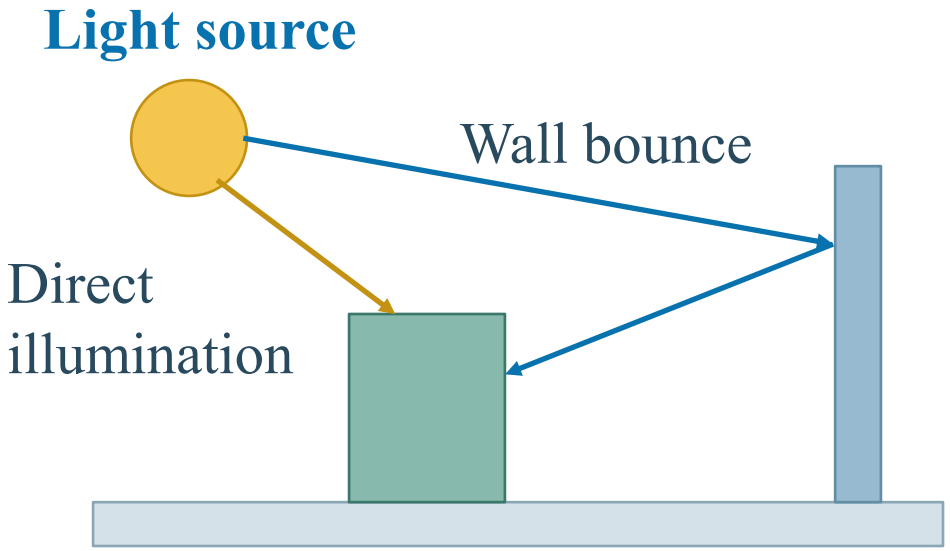


Figure: direct and bounced light paths (schematic)

Direct light: travels from source to target surface; an obstruction may put the target in shadow.

Indirect light: reaches the target after hitting another surface; the wall's color affects the bounce.

Role of indirect light: brightens shadowed areas and adds depth. It is costly; common options are baked lightmaps or UE5's real-time Lumen.

Lights, occlusion, materials, and exposure jointly change camera pixel values.

3.2.3 Lighting and Rendering: Direct and Indirect Light

Wall Reflections Can Light an Object in Direct Shadow

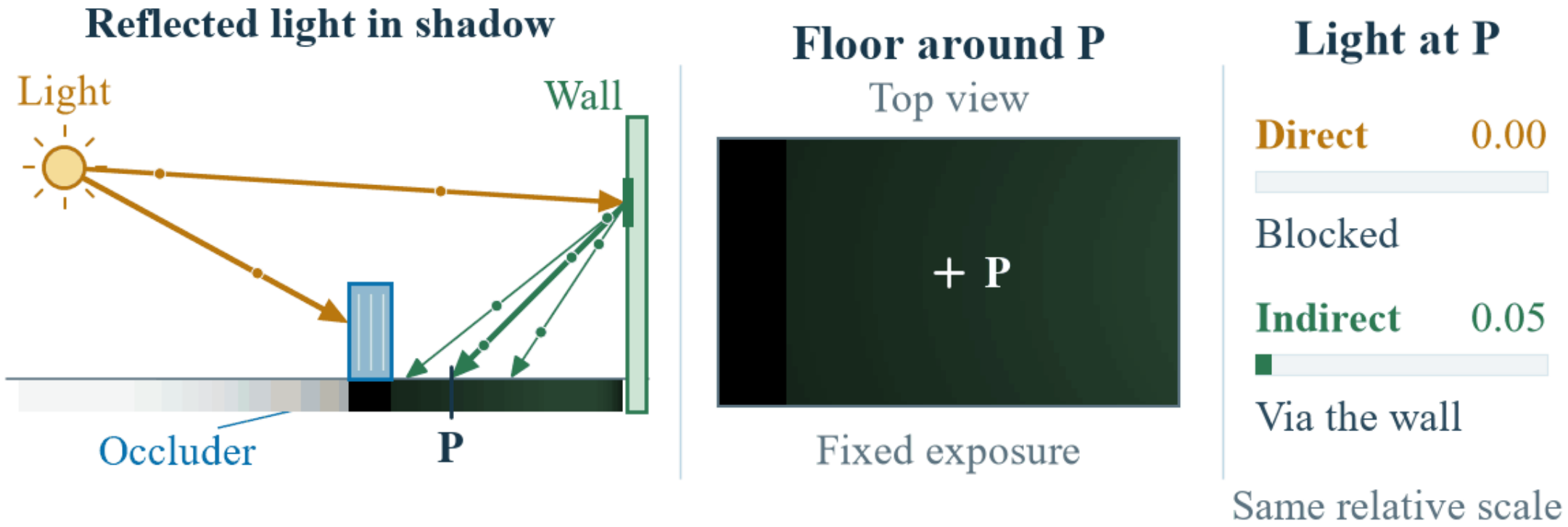


Figure: a moving occluder changes the direct path; one wall bounce can still illuminate point P.

3.2.3 Lighting and Rendering: Light Types and Mobility

Three Light Mobility Types Define Runtime Changes

Light types: Directional Lights model the sun; Sky Lights model the sky. Other types include Point Lights, Spotlights, and Rect Lights.

Choose Static for fixed scenes, Stationary for brightness changes, and Movable for a moving sun or vehicle headlights.

Static: fully fixed; precomputed in the editor; fastest

Stationary: adjustable brightness and color; partly precomputed; intermediate cost

Movable: adjustable position and direction; fully real time; highest cost

More freedom to change at runtime requires more real-time computation and higher rendering cost.

3.2.3 Lighting and Rendering: Lightmaps

Lightmaps Store Precomputed Static Lighting as Textures

Lightmap: a texture encoding static-light results for direct runtime sampling, avoiding real-time computation. It does not update automatically when a light or object moves.

Lightmass: UE's precomputation system. Each static mesh needs an additional, non-overlapping UV set to store lighting.

Worked example: if a 4 m wall fills a 64×64 lightmap, each texel (texture pixel) spans about $400/64 = 6.25$ cm. Finer shadow edges are blurred.

Platform example: in [installation directory]\RflySim3D\RflySim3D\Content, Changsha_BuiltData beside Changsha.umap stores the map's build data, mainly lighting.

Baked static lighting is cheapest at runtime, but becomes inaccurate when lights or objects move.

3.2.3 Lighting and Rendering: Lumen

Lumen Computes Indirect Light and Reflections in UE5

Lumen: UE5's real-time global illumination system. It updates light bounces as scenes change, without baking lightmaps or manually placing reflection captures.

Tracing: Lumen starts with screen information, then uses distance-field software ray tracing or hardware ray tracing according to project settings to estimate indirect light and reflections.

Reflections: update with the scene. Glass reflection and refraction also depend on materials and rendering settings.

Post-processing and fallback: combine Lumen with depth of field, color correction, or bloom. On constrained hardware, disable it and use lightmaps.

Use Lumen for changing scenes and lights; use lightmaps for fixed scenes when frame rate takes priority.

3.2.3 Lighting and Rendering: Shadows

Cascaded Shadow Maps Keep Nearby Shadows Sharp

Why shadows matter: without shadows, objects appear to float. Shadows show their relation to the ground.

Cascaded shadow maps: split shadows by camera distance, using higher spatial resolution near the camera and lower resolution farther away.

Worked example: with 2048-pixel shadow maps, a near cascade spanning 20 m gives $20/2048\ \text{m} \approx 0.98\ \text{cm}$ per pixel; a far cascade spanning 200 m gives about 9.8 cm.

Settings: a Directional Light's Dynamic Shadow Distance and cascade count set coverage and precision.

Tradeoff: coarse distant shadows are less noticeable. Near-cascade clarity matters most for a downward-looking camera flying near the ground.

Near cascades preserve detail at high spatial resolution; far cascades save resources at lower resolution.

3.2.3 Lighting and Rendering: Reflections

Choose Reflections for Scene Dynamics and Platform

Reflection: smooth surfaces can mirror the environment. Rough concrete still reflects light but usually forms no clear mirror image.

Lumen reflections: update with the scene in real time; dynamic scenes, UE5

Screen Space Reflections: use only pixels already on screen; dynamic, but cannot show off-screen objects

Reflection captures: pre-capture the environment at a location; relatively static scenes

Ray tracing: trace virtual rays in reflection directions; quality and cost depend on implementation and sampling

Choose a reflection method by scene changes, off-screen information, and available computing power.

Chapter Outline

Unreal Engine (UE): From Object State to Images and Interaction

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.2.1 Framework and Rendering Pipeline
Textbook: Sec. 3.1.4.1

3.2.2 3D Asset Management
Textbook: Sec. 3.1.4.2

3.2.3 Lighting and Rendering
Textbook: Sec. 3.1.4.3

3.2.4 Physics and Animation
Textbook: Sec. 3.1.4.4

3.2.5 Simulation Interaction Framework
Textbook: Sec. 3.1.4.5

3.2.4 Physics and Animation: Sources of Motion

Motion Solving, Collision Queries, and Animation Have Different Roles

Physics solving: when enabled, mass, forces, constraints, and collision shapes enter UE's motion and contact calculations.

Collision queries: test hits or overlaps by shape and channel. They may return results without driving rigid-body motion.

Component animation: update rotor or landing-gear transforms for continuous visible motion, using rules from Blueprints or the animation system.

Course platform: CopterSim can compute vehicle motion and send states to the 3D application. Visible rotor rotation alone does not calculate thrust.

Animation changes visible components; the chosen motion model computes dynamics.

3.2.4 Physics and Animation: Animation System

Skeletons, Animation Blueprints, and Control Rig Drive Motion

Skeleton: defines joint hierarchy and positions for a character or mechanical vehicle, providing the basis for animation deformation.

Animation Blueprint: a visual state machine that blends and switches animation sequences using speed, direction, state, and other parameters.

Control Rig: generate or modify motion in the editor or at runtime, adding constraints and inverse kinematics to infer joint angles from an end-effector position.

Landing-gear state machine: retract when more than 2 m above ground; extend below 1 m. Different thresholds prevent repeated switching near the boundary.

Separate the joint hierarchy from state transitions to make complex motion easier to modify.

3.2.4 Physics and Animation: Fixed Per-Frame Increments

Original Box Example: Add 1 Degree per Tick

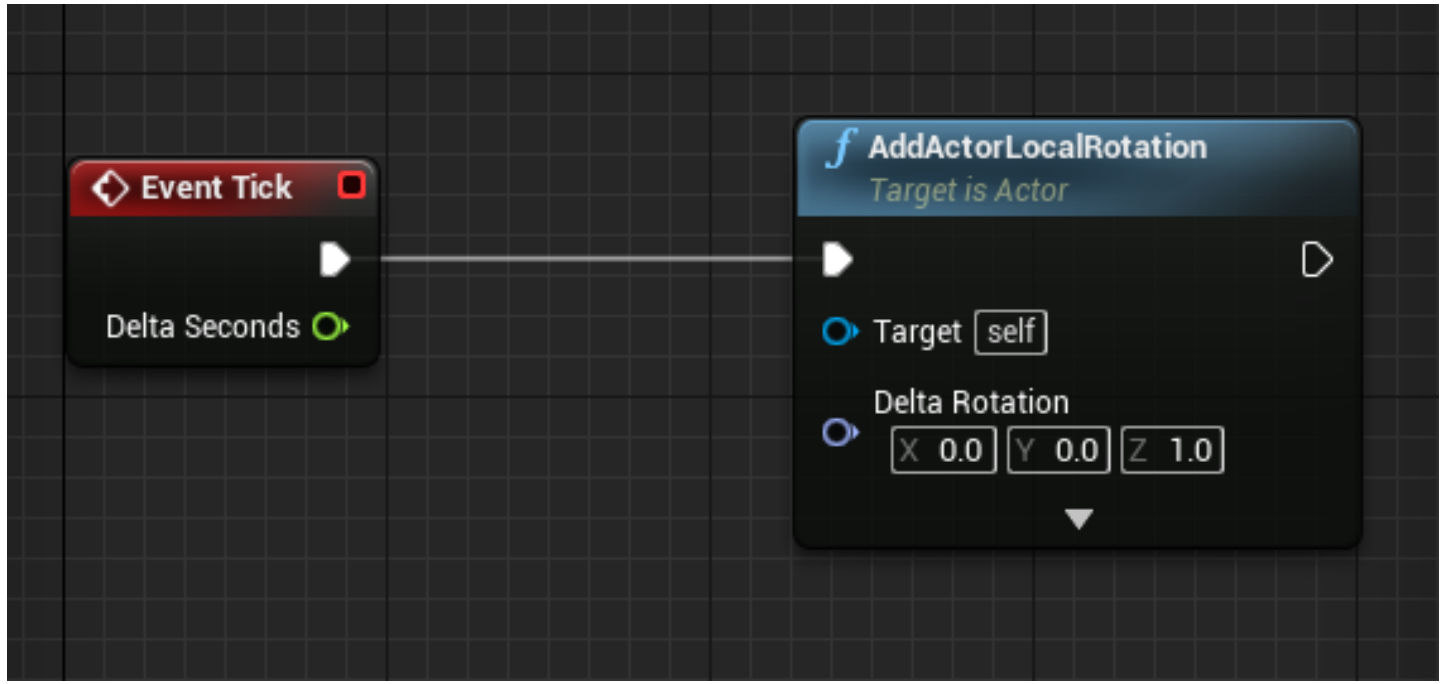


Figure: box Tick and local-rotation nodes

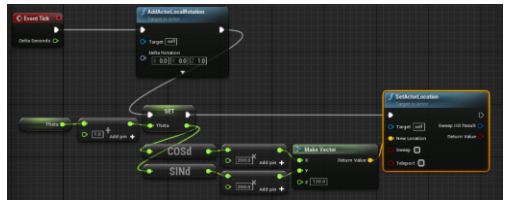
Execution wire: Event Tick calls AddActorLocalRotation, with Self as Target.

Angular increment: Delta Rotation Z is 1, adding 1° about local Z.

Time input: DeltaSeconds is unconnected. 30 FPS gives ≈30°/s; 60 FPS gives ≈60°/s.

3.2.4 Physics and Animation: Sine, Cosine, and Position

Original Box Example: Angle Also Drives Circular Motion



Full graph / position detail

State: add 1 to Theta each execution; COSd/SINd use degrees.

Position: $X = 200\cos\theta$, $Y = 200\sin\theta$, $Z = 120$ (θ = Theta).

Geometry: units are cm; at $\theta = 90°$, position is (0, 200, 120).



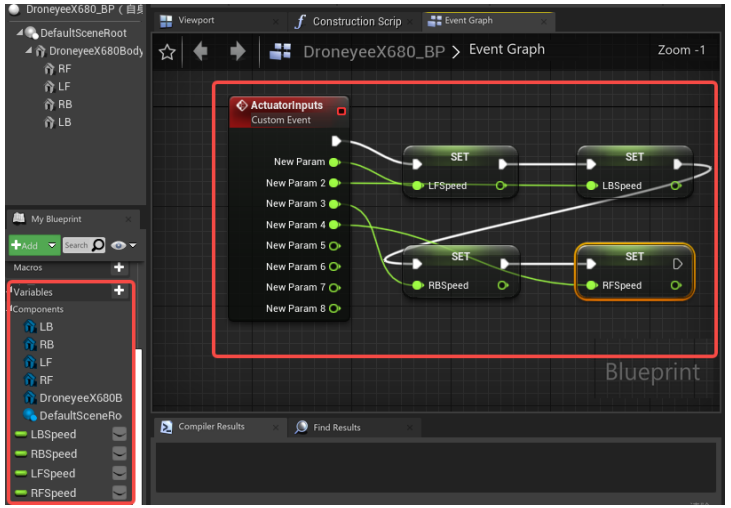
Update Theta each frame, then use sine and cosine to calculate world position.

3.2.4 Physics and Animation: Events to Variables

The X680 Input Event First Stores Four Rotor Speeds

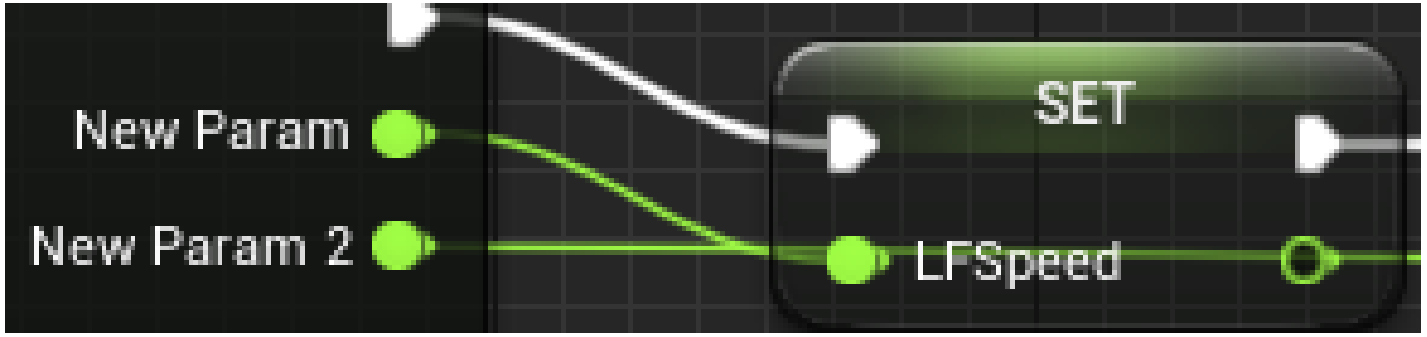
Event contract: ActuatorInputs takes 8 floats; this example's first 4 channels only store speeds.

Targets: 4 rotors attached to the body



Component tree and event overview

Detail: input 1 → LFSpeed



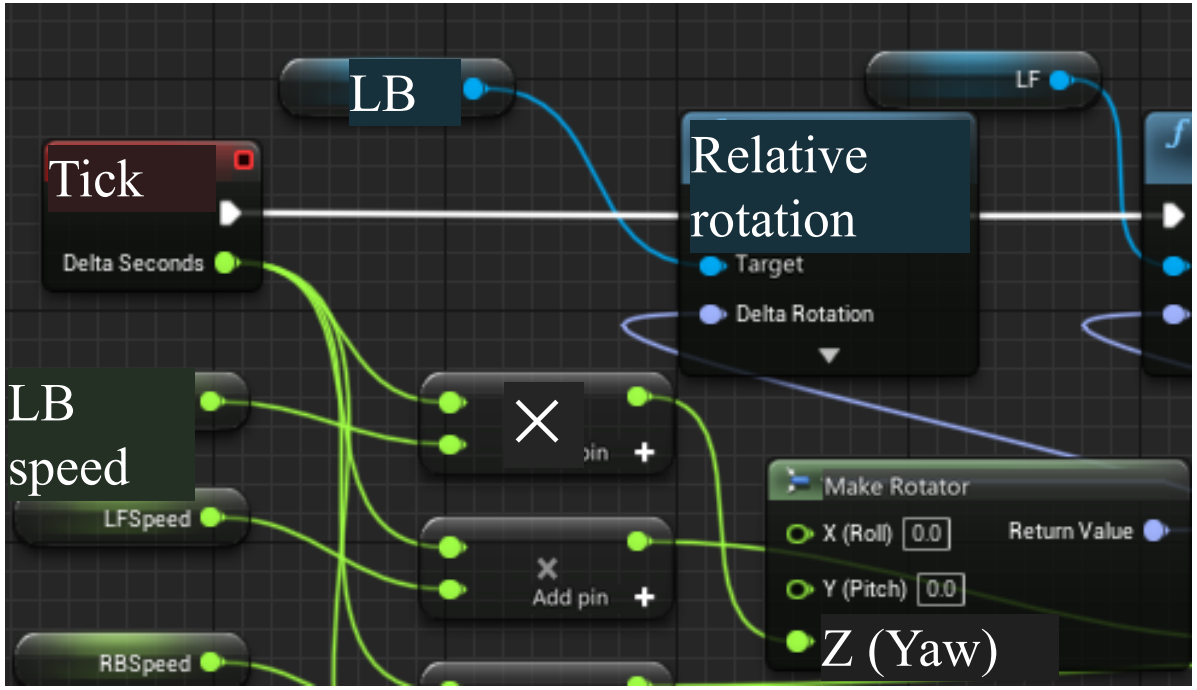
1→LFSpeed / LF front left 2→LBSpeed / LB rear left
3→RBSpeed / RB rear right 4→RFSpeed / RF front right

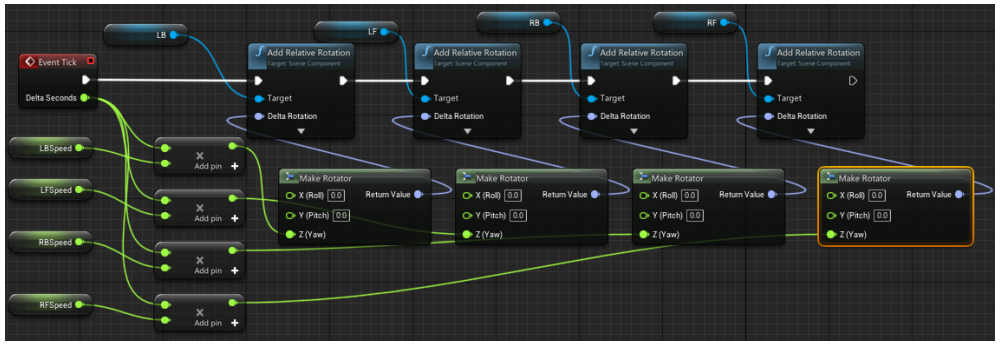
ActuatorInputs receives parameters; speed variables are read by later Tick updates.

3.2.4 Physics and Animation: Time to Component Rotation

X680 Multiplies Speed by Time for Each Frame's Rotation

Figure: enlarged LB path at left; all four paths at right





LBSpeed × DeltaSeconds
→ Make Rotator Yaw
→ LB: AddRelativeRotation

Yaw increment: $\Delta\psi = \omega\,\Delta t$, with ω in degrees/second and Δt in seconds.

Multiply angular speed by DeltaSeconds, then add the resulting angle to the corresponding rotor.

3.2.4 Physics and Animation: Frame-Rate Comparison

Two Update Rules Give Different Results over One Second

Teaching assumptions: one Tick per frame at 30 or 60 FPS (not performance measurements).

Update rule	30 FPS	60 FPS
Fixed 1°/frame	30°/s	60°/s
Specified 60°/s	Add 2°/frame	Add 1°/frame
Time-integrated over 1 second	60°	60°

Calculation: $60°/\mathrm{s} \times \frac{1}{30}\ \mathrm{s} = 2°$; $60°/\mathrm{s} \times \frac{1}{60}\ \mathrm{s} = 1°$.

Fixed angles per frame change speed with frame rate; speed × time preserves the specified angular speed.

3.2.4 Physics and Animation: Comparing Per-Frame Rules

Over One Second, Only the Fixed Increment Depends on Frame Rate

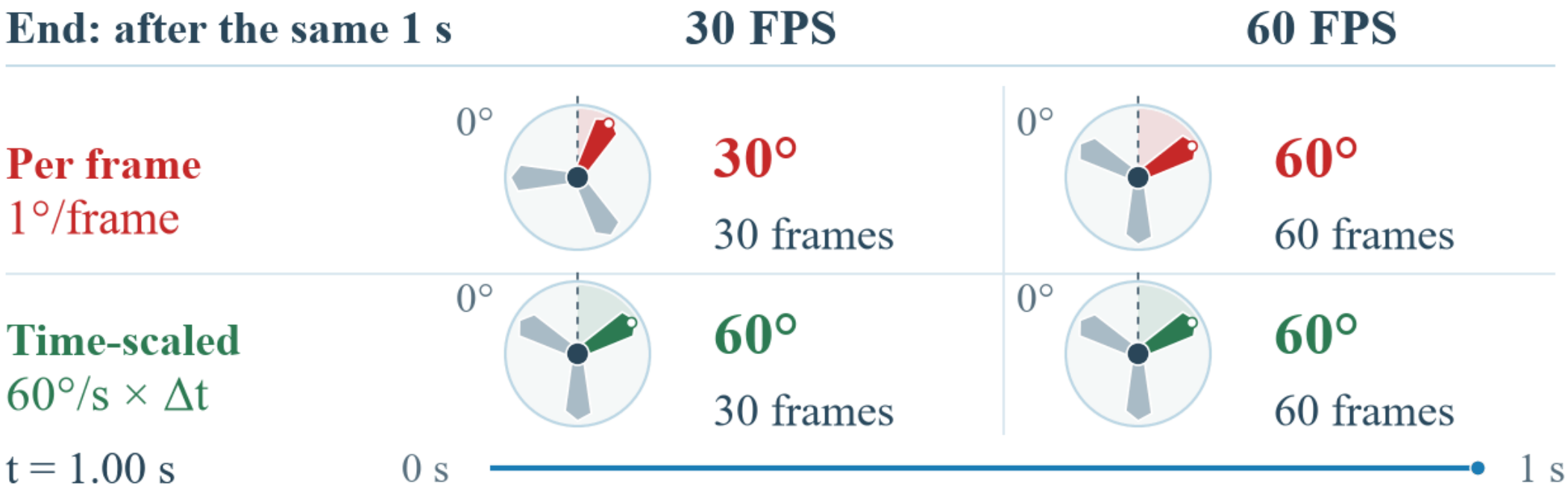


Figure: different Tick rules produce different rotations over the same model time.

Adding 1° per frame changes speed with frame rate; angular speed × interval still gives 60° per second.

3.2.4 Physics and Animation: Blueprints and C++

Blueprint Capabilities Depend on the Project's Exposed Interfaces

This Blueprint: ActuatorInputs stores speed, Tick calculates angle increments, and AddRelativeRotation updates the specified rotor Component.

C++ extensions: implement protocol reception, device access, or reusable calculations, then expose Blueprint-callable functions as needed.

Interface requirements: a blank UE project does not automatically include RflySim-specific events.
It needs the matching plugins, models, and asset-registration workflow.

Cooking (Cook): convert editor assets into runtime resources. Slide section 3.5 covers cooking for the platform's UE version and registering assets.

Blueprints organize events and states; underlying interfaces expose capabilities as nodes.

3.2.4 Physics and Animation: Collision Detection

Collision Detection Filters Pairs Before Computing Contact

Broad phase: use axis-aligned bounding boxes (AABBs) or bounding spheres to quickly reject pairs that cannot collide.

Narrow phase: compute contact from candidate collision shapes, obtaining contact points, normals, and penetration depth.

Worked example: AABBs intersect only if $a_{\min} \le b_{\max}$ and $b_{\min} \le a_{\max}$ hold on all three axes. X intervals [0, 2] and [1.5, 3] overlap, but Y intervals [0, 2] and [2.5, 4] do not: reject the pair.

Why two phases: 1000 objects give $n(n-1)/2 = 499500$ pairs, too many for detailed tests. A bounding volume hierarchy (BVH) or octree organizes space to compare nearby pairs.

The broad phase quickly filters bounding volumes; the narrow phase computes contact only for candidates.

3.2.4 Physics and Animation: Collision Settings

Collision Settings Enable Queries, Physics, or Both

Collision channels: WorldStatic is for stationary objects; WorldDynamic is for objects moved by code or animation.

No Collision: neither queries nor physics; decoration and distant scenery

Query Only: ray, sweep, and overlap queries; trigger regions and ranging rays

Physics Only: rigid-body and constraint simulation; debris needing simulated motion only

Query and Physics: both; objects that must respond physically and remain detectable

Use Query Only to ask whether something was hit; enable physics when objects must be moved by impacts.

3.2.4 Physics and Animation: Collision Responses

Both Objects' Responses Determine Block, Overlap, or Ignore

Response: each object sets Block, Overlap, or Ignore for each collision channel.

A and B denote each object's response to the other object's channel.

A Block + B Block: mutual blocking; hit events may occur

A Block + B Overlap: overlap only; no blocking

A Overlap + B Overlap: overlap; both must enable overlap events

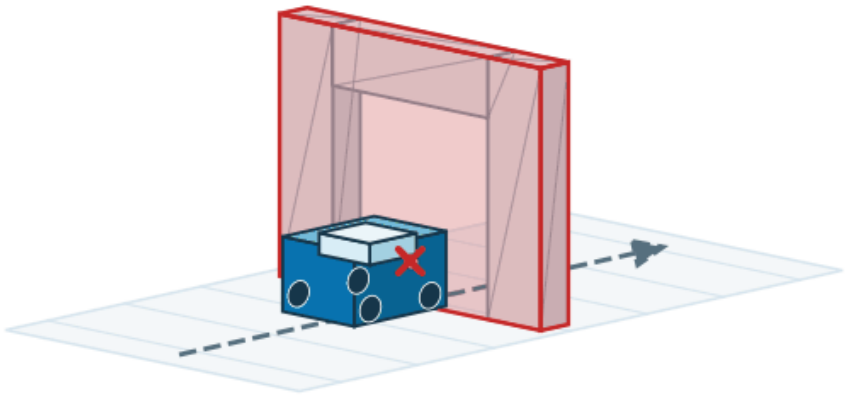
Either Ignore: neither blocking nor overlap events

Both must Block to block each other; if either Ignores, there is no interaction.

3.2.4 Physics and Animation: Rendering and Collision

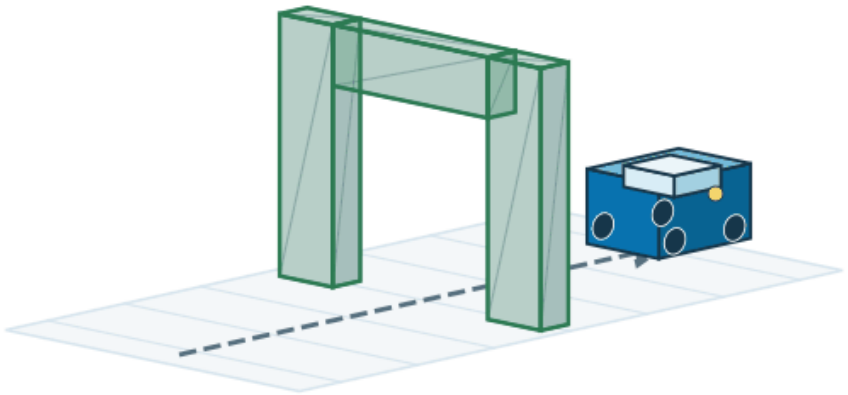
A Visible Doorway Can Still Be Blocked by Collision Geometry

One solid box: doorway blocked



Query hit: stop

Door-frame boxes: opening kept



No query hit: pass

Render mesh Collision shapes Path query; no contact forces

Figure: the same appearance with different collision shapes gives different passage-query results.

Render meshes, LOD, and collision shapes are configured separately; collision queries determine passage.

Chapter Outline

Unreal Engine (UE): From Object State to Images and Interaction

Chapter 3 3D Environment Modeling and Simulation

3.1 Simulation Fundamentals and Tools

3.2 UE Scenes, Rendering, and Interaction

3.3 Multimodal Sensor Data Generation

3.4 Distributed Architecture and Interfaces

3.5 Scene and Vehicle Assets

3.6 Examples and Integrated Practice

3.7 Exercises and Course Projects

5 Units in This Section

3.2.1 Framework and Rendering Pipeline
Textbook: Sec. 3.1.4.1

3.2.2 3D Asset Management
Textbook: Sec. 3.1.4.2

3.2.3 Lighting and Rendering
Textbook: Sec. 3.1.4.3

3.2.4 Physics and Animation
Textbook: Sec. 3.1.4.4

3.2.5 Simulation Interaction Framework
Textbook: Sec. 3.1.4.5

3.2.5 Simulation Interaction Framework: Input System

Map Device Signals into Actions a Controller Can Use

Input Mapping: map buttons and sticks from different devices to in-game events or commands, abstracting hardware differences.

Event-driven input: changes such as press and release trigger events; continuous inputs such as sticks have their current state read each frame.

Platform example: in RflySim3D, F1 opens help, B changes the followed aircraft, V switches onboard cameras, and a ground double-click displays hit information.

Input mappings handle device differences; control logic receives consistent action intents.

3.2.5 Simulation Interaction Framework: Control Relationships

Possession Connects a Controller to Its Pawn

Pawn: an Actor that a Controller can possess; the textbook also describes it as a game piece or character body.

Character: a Pawn subclass with common character-movement capabilities, suitable for walking, jumping, and related behavior.

Controller: a PlayerController handles keyboard or gamepad input;
an artificial intelligence (AI) controller makes rule-based decisions. The Controller itself displays no mesh.

Possess: establishes a Controller's control of a Pawn;
Unpossess releases it. The default is one-to-one: release before switching.
The entity and its Components still implement motion.

The Controller handles intent; the Pawn and its Components implement motion.

3.2.5 Simulation Interaction Framework: Rules and Global State

Game Mode Sets Rules; Game State Tracks Progress; HUD Gives Feedback

The theater analogy: Actors are performers, Pawns are controlled characters, and Game Mode is the common script.

Interface building: Unreal Motion Graphics (UMG) combines widgets into interactive menus, instruments, and other interfaces.

Game Mode: defines rules and flow; takeoff conditions and mission completion

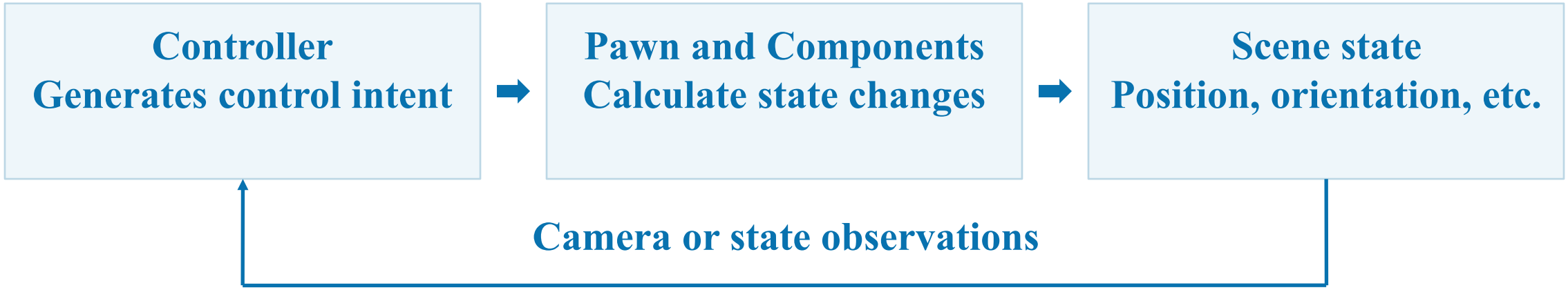
Game State: records global state; mission progress and score

Head-Up Display (HUD): shows key live information; altitude, speed, and alerts

Game Mode sets rules, Game State tracks progress, and HUD/UMG display information to people.

3.2.5 Simulation Interaction Framework: Input and Feedback

Trace Control Intent into Scene State



Forward path: send control intent to the Pawn and movement Components to compute displacement and update scene state.

Feedback: camera or state interfaces return results for control logic to compute the next input.

Control intent changes state; camera or state observations return to the control logic.

3.2.5 Simulation Interaction Framework: AI Behavior

Behavior Trees, Blackboards, and Perception Form the Decision Loop

Behavior Tree: a visual decision flow with Selector, Sequence, and Task nodes; actions include movement and search, according to conditions.

Blackboard: the Behavior Tree's data store for target positions, visibility, and other information used in condition checks.

Perception system: sight, hearing, and other senses detect targets; project handlers write them to the Blackboard, triggering behavior changes.

Navigation Mesh (NavMesh): builds traversable scene regions for path planning and obstacle avoidance, with dynamic updates.

Mass Entity: UE5's lightweight architecture can control thousands of crowd, traffic, and formation agents when configured for the workload.

Perception handlers write the Blackboard; the Behavior Tree reads it to decide; the NavMesh routes to the target.

3.2.5 Simulation Interaction Framework: Networking and Multiple Agents

Server Authority, Replication, and RPC Keep Networked States Consistent

NetDriver: manages client–server connections and data transfer for state synchronization.

Property replication (Replicated): when Actor replication is enabled and relevancy and replication conditions hold, the server schedules the latest state for relevant clients.

Remote Procedure Call (RPC): invoke locally and execute remotely, for example when a client requests a mission start.

Worked scenario: the server moves drone 2 to (100, 0, 50) m. Eligible clients display it on receipt; arrival times may differ, and intermediate changes need not all arrive.

Iris: opt-in UE5 replication for large multi-agent simulations with hundreds of clients, given sufficient server and network capacity.

The server holds authoritative state; replication sends updates to clients; RPC carries individual requests.

Learning Resources

Object Resources: Entities, Components, and Control

Actor: root Components, transforms, and instance lifecycles

https://dev.epicgames.com/documentation/en-us/unreal-engine/actors-in-unreal-engine

Component: logic Components and spatial attachment

https://dev.epicgames.com/documentation/en-us/unreal-engine/components-in-unreal-engine

Controller: Possess and the Pawn control relationship

https://dev.epicgames.com/documentation/en-us/unreal-engine/controllers-in-unreal-engine

Learning Resources

Node Practice: Time Updates from the Box to X680

Tick: time intervals, tick groups, and update dependencies

https://dev.epicgames.com/documentation/en-us/unreal-engine/actor-ticking-in-unreal-engine

Box Readme: Components, 1° per frame, and circular position

https://rflysim.com/doc/zh/px4psp/RflySimAPIs/3.RflySim3DUE/0.ApiExps/e0_DevToolsUsage/1.UEBlueprintUsage/Readme.pdf

X680 Readme: ActuatorInputs and speed × time

https://rflysim.com/doc/zh/px4psp/RflySimAPIs/3.RflySim3DUE/2.AdvExps/e2_BlueprintModel/1.BlueprintBuild/Readme.pdf

Learning Resources

Imaging Resources: Materials, Lighting, and Rendering Paths

PBR: comparative images for Base Color, Metallic, and Roughness

https://dev.epicgames.com/documentation/en-us/unreal-engine/physically-based-materials-in-unreal-engine

Forward rendering: capability tradeoffs against the deferred path

https://dev.epicgames.com/documentation/en-us/unreal-engine/forward-shading-renderer-in-unreal-engine

Lumen: conditions for UE5 dynamic indirect lighting and reflections

https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine

Learning Resources

Performance and Interaction: Metrics and Collision

Stat commands: Frame, Game, Draw, and GPU metrics

https://dev.epicgames.com/documentation/en-us/unreal-engine/stat-commands-in-unreal-engine

Collision complexity: when to use simple shapes or triangle meshes

https://dev.epicgames.com/documentation/en-us/unreal-engine/simple-versus-complex-collision-in-unreal-engine

Console variables: frame-rate caps and forced LOD settings

https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-console-variables-reference

Classroom Exercises

Analyze a Drone's Simulation Image and Interaction

Frame time: Game 12 ms, Draw 8 ms, GPU 21 ms. What is the approximate frame time? Which stage should be optimized first?

Rotor angle: X680's RF command is 120°/s. If frame rate drops from 60 to 40 FPS, how far does it rotate per frame and per second?

Collision: a doorway's render mesh has an opening, but its collision shape is one solid box. What happens to a drone trying to pass? Which setting should change?

Lighting: for an inspection from morning to evening, which light mobility should the sun use? What is the cost?

Terrain: with heightmap value 30720 and Z scale 100, what is the elevation in meters?

State the rule or formula, substitute the values, then explain what the result means.

Reference Answers

Exercise Answers: State the Rule, Then Substitute Values

Frame time: the stages pipeline in parallel, so the frame is near the slowest stage, GPU ≈21 ms (≈48 FPS). Reduce GPU work first.

Rotor angle: $120/40 = 3°$ per frame (2° at 60 FPS); still 120° over one second. Time-based accumulation is independent of frame rate.

Collision: the query hits the solid collision box and blocks the drone. Fit collision shapes to the doorframe or use complex collision.

Lighting: use a Movable light for the sun, allowing shadows to change with time. Real-time computation increases frame time.

Terrain: $(30720 - 32768)/128 = -16$ gives −16 m, or 16 meters below zero elevation.

Use the slowest pipeline stage, angular speed, collision shape, and light mobility to reason through the answers.

Section Summary

Five Threads from Object State to Images and Interaction

Framework and rendering: update state in the interaction framework; the CPU culls and submits; the GPU performs vertex transformation, rasterization, deferred lighting, and post-processing.

Assets and lighting: Components reference assets. LOD, mipmaps, and World Partition control detail and loading; lighting trades precomputation against real-time work.

Physics and interaction: animation changes only the display; queries and both objects' responses determine collision. Mapped input goes to Controllers; dedicated objects manage rules and global state.

Common errors: adding frame times; fixed per-frame increments changing speed with frame rate; assuming a visible opening is passable; overlooking that property replication runs only from server to client.

Each frame sets state before imaging; timings, lighting, and collision explain the simulation result.