3D GIS

Gaming Engines

This module introduces game engines as a platform for presenting three-dimensional geospatial data.

Module Topics

1. Unity vs. Unreal
2. Benefits and uses of gaming engines
3. Challenges
4. Types of assets
5. Mesh attributes
6. Cesium for getting geospatial 3D tiles into game engines
7. ESRI integration with game engines
8. Blender → Unreal
9. ArcGIS Pro → Unreal
10. Using assets generated by others
11. Unreal Engine overview
12. Unreal Engine installation
13. Blueprints
14. Software/user interface layout
15. Project structure
16. Projects vs. levels
17. Assets in Unreal Engine
18. Types of actors and components
19. Nanite
20. Brushes
21. Materials
22. Lighting, sky, and atmosphere
23. Physics
24. Collisions
25. Players and controls
26. Landscapes and foliage

2

These are the topics covered in this module.

Software

We will look briefly at the two engines that dominate this space. There are others, but in practice the decision for a geospatial project comes down to Unity or Unreal, and both have first-party geospatial integrations that make them viable.

Unity

- Cross-platform game engine
- Released in 2005
- Indie game development
- 2D and 3D game development
- Interactive simulation development

https://unity.com/

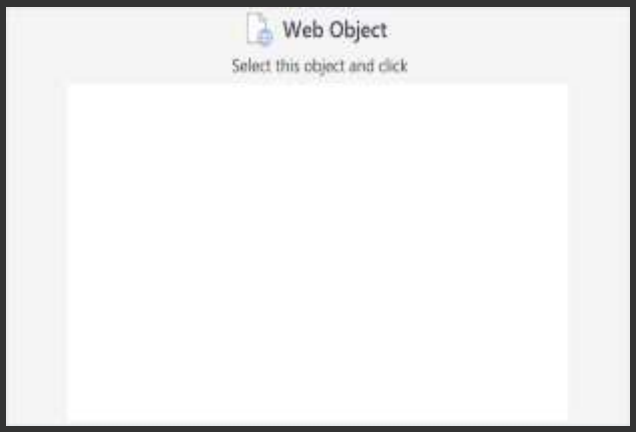
4

Unity is one of the two commonly used engines. It was released in 2005 and built its reputation on being approachable, which is why it dominates indie and mobile development. Unity uses C# for scripting, which many people find gentler than C++, and its footprint is lighter, so it runs comfortably on modest hardware. It handles both 2D and 3D work, and it is widely used well outside games for simulation, training, architectural visualization, and industrial applications. Cesium and Esri both provide Unity integrations, so it is entirely viable for geospatial work. If you already know C# or expect to deploy to mobile devices, Unity is a reasonable choice.

Unreal Engine

- Game engine
- Developed by Epic Games
- Used for 2D and 3D game development, film and cinema, and VR/AR development

UNREAL ENGINE



https://www.unrealengine.com/

5

Unreal Engine is the other commonly used gaming engine, developed by Epic Games. In this course we will discuss Unreal rather than Unity. The reasoning is worth stating. Unreal's renderer produces more photorealistic output with less effort, which matters when the goal is to represent a real place convincingly. Its Nanite and Lumen systems handle dense scanned geometry and realistic lighting in ways that are directly useful to us. Blueprints let you build interactive behavior without writing code, which lowers the barrier considerably. And Unreal has become the default in film, television, and architectural visualization, so the workflows and tutorials you will find align well with what we are doing. Note also that Unreal is not just a game engine anymore; a great deal of its development is now driven by film and simulation users.

Why use gaming engines?

Before we go further it is worth being honest about the trade. A game engine is a genuinely different kind of tool from the GIS software you have been using, and it is good at a specific set of things and bad at others. The next two slides lay out both sides.

Benefits

- Fast rending of 3D scenes/GPU accelerated
- Styling, materials, and textures applied to meshes
- Modeling of atmosphere and lighting
- Modeling of physics (e.g., gravity)
- Assign physics and interactions to objects/players
- Interactive viewing in 3D (first person, third person, virtual reality)
- Supports augmented reality (AR) and virtual reality (VR) development
- Supports development for different platforms (PC, mobile device, gaming consoles, VR headsets)
- Interact with environment using an avatar and game controls
- Build other user interface components

These are the reasons to reach for an engine. The first is speed: rendering is GPU accelerated and happens in milliseconds, so you navigate the scene rather than waiting on a render queue. The second cluster is about realism. Materials, textures, atmosphere, and lighting are modeled to a standard that traditional GIS rendering simply does not attempt. The third cluster is about interaction. You can assign physics to objects, walk through the scene as an avatar, and build interface elements that respond to the user. That is what makes an engine appropriate for public presentation, stakeholder engagement, and site communication. And finally, deployment: the same project can be built for a desktop, a phone, a console, or a VR headset. For a planning meeting or a public consultation, letting people walk through a proposal is a fundamentally different kind of communication than showing them a map.

Challenges

- Coordinate systems
- Preparation of spatial data and assets
- Complex workflows/pipelines
- Still actively being developed
- Not originally developed for spatial data
- Different terminology

And here is the other side. The first item is the one that will cost you the most time. Engines have no native concept of a coordinate reference system; a level is local space with an arbitrary origin, and reconciling that with projected coordinates is real work, which we will return to at the end of this module. Data preparation is substantial: geospatial formats are not engine formats, and conversion is rarely a single step. The pipelines are long, and a change upstream often means re-running several stages. The engines are also under constant development, so a tutorial from two years ago may describe an interface that no longer exists. Perhaps the most persistent friction is simply vocabulary. These tools were built by and for game developers, and the words they use for familiar concepts are different from ours. Much of this module is devoted to translating that vocabulary.

Getting data into a gaming engine

This section covers the practical question of what an engine will accept, how to prepare data from the tools you already know, and where to find content you do not want to or can't build yourself.

Types of Assets

- Polygon mesh
- Point cloud
- Gaussian splats
- Voxels
- Material = shader + parameters
- Textures = image data for material input

These are the forms that three-dimensional content takes. A polygon mesh is the dominant representation and what most of this pipeline is built around. Point clouds are your native product from lidar and dense matching, and both engines can display them, though support is less mature than for meshes. Gaussian splats are a newer capture representation that renders photorealistically but is not a surface, so it does not support measurement or collision. Voxels appear in volumetric and procedural workflows. The last two items are the ones students most often conflate, so it is worth separating them clearly. A material is a shader with parameters; it describes how a surface responds to light. A texture is image data feeding one of those parameters. A material is not a picture, and a texture is not a material.

Mesh Attributes

- Normals = surface direction per vertex or face; controls shading smoothness; flipped normals are the single most common cause of "why is my model black.“
- UV Coordinates = the 2D mapping from texture space to surface; textures cannot be applied without them; photogrammetric meshes arrive with machine-generated UVs that are efficient but unreadable; hand-modeled assets get deliberate unwraps
- Associated Materials/Textures

Geometry alone is not enough. A mesh carries attributes, and two of them cause most of the trouble when moving content between applications. Normals define which way a surface faces and control how smoothly it shades. If normals are flipped, the surface renders black or invisible, and this is far and away the most common complaint when an imported model looks wrong. UV coordinates map the two-dimensional texture onto the three-dimensional surface, and without them you simply cannot apply a texture. Photogrammetric meshes come with machine-generated UVs that are efficient but unreadable to a human, while hand-modeled assets have deliberate unwraps you can work with. The third item is a reminder that materials and textures travel separately from geometry, which is why export format and resource packing matter on the next few slides.

Cesium

- Cesium Ion: store 3D tiles
- Cesium for Unity: integration with Unity engine
- Cesium for Unreal: integration with Unreal engine
- CesiumJS: JavaScript library for creating 3D web scenes

https://cesium.com/

12

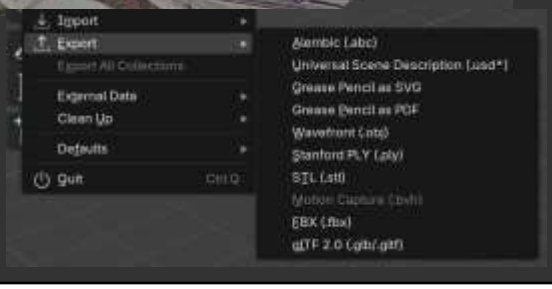
The Cesium products are used to represent 3D data on the web. Cesium Ion allows you to store 3D tiles on the web for use in applications. Cesium has been integrated into Unity as Cesium for Unity and into Unreal Engine as Cesium for Unreal. CesiumJS supports the creation of 3D web scenes displayed in a viewport on a web page. For our purposes, Cesium for Unreal is the single most important tool on this slide. It solves the coordinate system problem we flagged earlier: it anchors the scene to a real latitude and longitude, handles the precision issues that arise with large coordinates, and streams 3D Tiles into the engine. It also gives access to global terrain and imagery, and to Google's photorealistic 3D tiles, so you can have a plausible world around your own data without modeling any of it. It is free and open source, which makes it an easy recommendation.

ESRI Tools

- ArcGIS Maps SDK for Unity (https://developers.arcgis.com/unity/)
- ArcGIS Maps SDK for Unreal Engine (https://developers.arcgis.com/unreal-engine/)

Esri provides its own software development kits for both engines. These take a different approach from Cesium: rather than streaming a general tile format, they connect directly to ArcGIS services, so scene layers, imagery, elevation, and feature layers you already publish can be consumed in the engine. If your organization is already invested in ArcGIS Online or Enterprise, this is the shorter path, because your existing services become the data source with no export step. The trade is that you are tied to the Esri ecosystem and to its licensing. Cesium and the Esri SDKs solve overlapping problems, and which one fits depends more on where your data already lives than on the technical merits.

Blender



- If using textures: pack resources

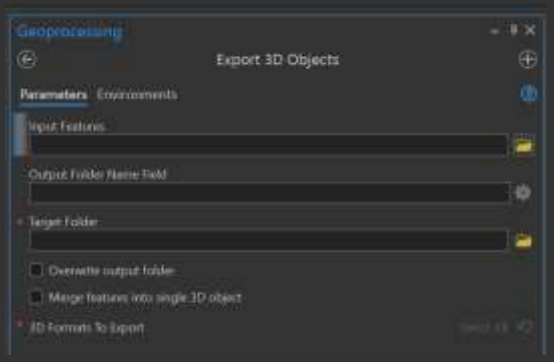
- Export to: OBJ or FBX

14

This slide describes how to create geospatial 3D objects in Blender for import into a gaming engine. The process is straightforward: draw or create your mesh over a base map in Blender using the GIS plugin. If you are using textures, pack the resources so all files are self-contained, otherwise the export will reference images that do not travel with it and your model will arrive untextured. Export to OBJ or FBX. FBX is generally the better choice because it handles materials and hierarchies more reliably. Two things to check before exporting: apply your transforms, because unapplied scale and rotation will produce surprises on import, and confirm the axis convention, since Blender is Z-up and several other applications are Y-up. Getting this wrong produces a model lying on its side, which is easy to fix but confusing the first time.

ArcGIS Pro

- Export 3D Objects tool
- Must be stored as 3D object layer or multipatch
- Can export to FBX or glTF
- Can include textures



ArcGIS Pro is the other export path and it is the one you will use for content that already exists as GIS data. The Export 3D Objects tool is the mechanism, and the requirement is that your data is stored as a 3D object layer or a multipatch feature class. That is worth stressing: a polygon with an extrusion applied for display is not a 3D object, and the tool will not accept it. You may need to convert first. FBX and glTF are both supported, and both carry textures if the source has them. This path is how extruded building footprints, procedurally generated structures, or multipatch models from a city dataset move into the engine.

3rd Party Assets

- Fully Free
  - Poly Haven
  - Smithsonian Open Access 3D
  - NASA 3D Resources
  - Scan the World
  - Blender Studio
- Mixed-License
  - Sketchfab
  - BlenderKit
  - SketchUp 3D Warehouse
  - Objaverse/Objaverse-XL
- Commercial
  - Fab (integrated with Unreal)
  - TurboSquid
  - CG Trader
  - Poliigon
  - Textures.com
  - Kitbash3D
  - Adobe Substance 3D Assets

You will not model everything yourself, and you should not try. The licensing tier at the left is as important as the quality, especially in a course context. The fully free sources are all effectively public domain, which means you can redistribute them in a lab packet without tracking attribution. Poly Haven in particular is excellent and is where I would send you first for textures and environment lighting. Smithsonian and NASA are useful for demonstrating what real scanned artifacts look like. The mixed-license group requires you to read the license on each individual asset; Sketchfab is the largest repository of downloadable scans anywhere, and SketchUp 3D Warehouse is unbeatable for quickly populating a scene with street furniture and vehicles, though the geometry is often loose. The commercial marketplaces are where you go when you need something specific and polished.

3rd Party (Geospatial)

- Cesium Ion
- Google Photorealistic 3D Tiles API
- OpenStreetMap Building Footprints
- OpenTopography
- USGS 3DEP
- Municipal open data portals (e.g., Helsinki, Zurich, and New York)
- ESRI Living Atlas

These are the sources specific to our discipline, and several of them will already be familiar. Cesium Ion and the Google photorealistic tiles give you global context with essentially no preparation, which is the fastest way to put your own data in a recognizable setting. OpenStreetMap footprints with height attributes extrude into serviceable low-detail city models. OpenTopography and the USGS 3D Elevation Program (3DEP) are your terrain sources. Municipal open data portals are worth watching: a growing number of cities publish proper 3D building models with documented levels of detail (LODs), and the European cities listed are among the best examples. Esri's Living Atlas rounds this out if you are working inside that ecosystem.

Procedural Generation

- Blender Geometry Nodes
- Gaea (synthetic terrain)
- World Machine (synthetic terrain)
- SpeedTree (trees)
- Substance Designer (materials)

Asset libraries have a ceiling, and procedural generation is how you get past it. Blender's geometry nodes are the right tool for scattering and instancing, and they are free and already in your workflow from earlier in this course. Gaea and World Machine generate synthetic terrain, which is useful when you need plausible landscape beyond the edge of your real data. SpeedTree is the industry standard for vegetation and integrates directly with Unreal. Substance Designer authors materials parametrically rather than as fixed images, which means resolution is not baked in and variations are trivial. The general principle is that anything you need many slightly different copies of is a candidate for procedural generation rather than a library download.

Unreal Engine

From here the module narrows to Unreal Engine specifically. The goal for the rest of this lecture is vocabulary and mental model. Most of the difficulty people have with Unreal is not that the software is hard; it is that the words are unfamiliar, and the documentation assumes you already know them.

Software Overview

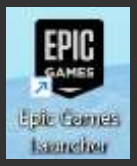
- Real-time 3D engine and editor; not a modeling package
- Current generation: Unreal Engine 5
- Developed by Epic Games; source code available on GitHub
- Free to download and use for learning, education, and non-commercial work
- Royalty applies only to commercial products past a revenue threshold
- Runs on Windows, macOS, and Linux
- Builds for PC, mobile, consoles, and VR headsets
- Fab asset marketplace integrated into the editor
- Extensive official documentation, tutorials, and community forums

https://www.unrealengine.com/

A few orienting facts before we open the software. Unreal is a real-time engine with an editor attached; it is not a modeling package, and although it now has modeling tools, you should still build content in Blender or ArcGIS Pro and bring it in. The licensing is generous for our purposes: it is free to download and use for learning and non-commercial work, and the royalty only applies to commercial products past a revenue threshold. Nothing we do in this course triggers a payment. The source code being on GitHub is unusual for software of this scale and it means the engine is genuinely inspectable, though you will not need to do that. The final point matters practically: because Unreal is used across games, film, and architecture, the tutorial ecosystem is large, and you will often find that the architectural visualization community's material is closer to our needs than the game development material is.

Installation

- Install the Epic Games Launcher first
- Engine versions are installed and managed from the launcher's Unreal Engine tab
- Multiple engine versions can coexist; each project is tied to a version
- Hardware
  - Dedicated GPU required; 8 GB VRAM or more recommended
  - 16 GB RAM minimum, 32 GB preferred
  - 50+ GB disk per engine version; SSD strongly recommended
- Optional components at install time
  - Starter content and templates
  - Platform support (Android, iOS, Linux) — install only what you need
  - Engine source and debug symbols — large, and rarely needed
- First launch compiles shaders; expect a long initial wait



Installation trips people up more than it should, so a few specifics. You do not download Unreal directly; you install the Epic Games Launcher, and the engine is installed from its Unreal Engine tab. Multiple versions can coexist, and each project is bound to the version that created it, which matters because opening a project in a newer version is a one-way conversion. A dedicated GPU is not optional, and integrated graphics will not give a usable experience. Disk space is the other surprise, since each engine version is around fifty gigabytes before you add any projects, and an SSD makes a very large difference to load times. At install time you will be offered optional components. Take the starter content and templates. Skip the platform support packages unless you actually intend to build for those devices, and skip the debug symbols entirely. The engine compiles thousands of shaders, and on a modest machine that can take a long time. It is not frozen, and it only happens once.

Blueprints and C++

- Two ways to define behavior in a project
- Blueprints = visual scripting using node graphs
  - No compiler or IDE setup required
  - Immediate iteration; changes are visible right away
  - Readable without programming experience
  - Slower at runtime than compiled code
- C++ = full access to the engine
  - Best performance; required for engine-level modification
  - Requires Visual Studio or Xcode and longer compile cycles
- Both can be used in the same project
- Common pattern: C++ for framework, Blueprints for iteration
- Blueprints alone are sufficient for most geospatial visualization work

Unreal gives you two ways to make things happen, and for this course the answer is Blueprints. Blueprints are visual scripting: you wire nodes together, with execution flowing left to right along a white wire and data flowing along colored wires. The advantages are real. There is no compiler or development environment to configure, changes take effect immediately, and the graph is readable by someone who has never programmed. The cost is runtime performance, which is slower than compiled code, and that is essentially irrelevant at the scale we are working. C++ gives you full access to the engine and the best performance, but it requires Visual Studio, and every change means a compile cycle. In professional projects the two are mixed, with programmers building systems in C++ and designers assembling behavior from them in Blueprints. For everything in this course, Blueprints alone are sufficient.

Software Layout

- Viewport = the 3D view of the current level
- Outliner = list of every Actor in the level
- Details = properties of the current selection
- Content Browser = every asset in the project
- Toolbar = save, settings, build, and Play
- Place Actors = palette of lights, shapes, and volumes
- Modes = Select, Landscape, Foliage, Modeling, Mesh Paint
- Navigation
  - Right mouse button held + WASD to fly
  - F to focus on the selected object
  - Alt + left drag to orbit
- Panels are dockable; layouts can be saved and reset

This is the orientation slide for the interface. There are six main panels, and they map onto obvious functions. The viewport is your 3D view. The Outliner is the scene graph presented as a searchable list, and it is how you find something when the viewport is crowded. The Details panel shows properties for whatever is selected and is where nearly all configuration happens. The Content Browser holds the project's assets, which is a different thing from the level's contents, and that distinction is the subject of the next several slides. The Modes dropdown is worth pointing out early because Landscape, Foliage, and Modeling all live there. The navigation controls at the bottom are the single most useful thing to memorize on day one. Right mouse held down plus WASD flies the camera; F focuses on the selection. I would encourage spending twenty minutes just flying around a template level before trying to build anything.

Terminology Overview

- Containers
  - Project, Level (Map), World Partition
- Content
  - Asset, Actor, Component, Blueprint
- Geometry
  - Static Mesh, Skeletal Mesh, Brush (BSP), Nanite, Landscape
- Appearance
  - Material, Material Instance, Texture, Mobility, Lumen
- Behavior
  - Collision, Overlap, Physics, Pawn, Player Controller, Game Mode
- Presentation
  - Widget (UMG), Post Process Volume, Attenuation

This slide exists as a reference and as a warning. Almost all of the difficulty in learning Unreal is vocabulary rather than concept. I have grouped the terms by what kind of thing they describe, because the grouping itself is part of the mental model. The container group tells you how work is organized. The content group is the core object model, and if you understand only one thing from this lecture, make it that a project contains levels, levels contain Actors, and Actors are built from components. The remaining groups are systems layered on top of that model. Each of these terms gets its own treatment in the slides that follow.

Project Structure

- .uproject = the entry point file you open
- Content/ = all assets: meshes, materials, textures, levels
- Config/ = project-wide settings stored as text
- Plugins/ = extensions such as Cesium for Unreal or the ArcGIS Maps SDK
- Source/ = C++ code, if the project uses any
- Saved/ and Intermediate/ = caches; safe to delete and will regenerate
- Templates set the starting point: Blank, First Person, Third Person, VR
- Project Settings hold global configuration that applies to every level
- Everything a project uses lives inside the project folder

A project is the largest unit of organization: a folder on disk with a .uproject file at the top that you double-click to open. Everything the project uses lives inside that folder, because Unreal has no concept of linking to an asset stored elsewhere on your drive. The Content folder is where every asset ends up. Config holds project-wide settings as plain text, which is occasionally useful when something is misconfigured and the interface will not let you fix it. The Plugins folder is where Cesium for Unreal or the ArcGIS Maps SDK will install. Two practical notes. The Saved and Intermediate folders are caches, they can grow to many gigabytes, and they are safe to delete; if you are zipping a project to submit, delete them first or your submission will be enormous. And the template you choose when creating the project matters more than it looks, because it determines whether you get a working character and camera out of the box.

Levels

- Level = a single scene, also called a Map
- Saved as a .umap asset in the project
- One project can contain many levels
- Only one level is open in the editor at a time
- Persistent level plus sublevels for large scenes
- World Partition streams large worlds automatically
- Startup level is set in Project Settings

A level is a single scene. The terminology is slightly confusing because Unreal's interface and documentation use level and map interchangeably; they are the same thing, and the file extension is .umap for historical reasons. A project can hold as many levels as you want, but only one is open at a time in the editor. The persistent level plus sublevels arrangement is the older approach to breaking up a large scene, and you will still see it in tutorials. World Partition is the modern replacement, and it is largely automatic: it divides the world into a grid and loads only the cells near the camera. For geospatial work this is not optional. A landscape at any useful resolution covering more than a few square kilometers will exceed what you can hold in memory, and streaming is what makes it tractable. If your editor becomes unusably slow on a large scene, streaming configuration is the first thing to check.

Project vs. Levels

- Project = the whole application
  - Opened by the .uproject file
  - Owns all assets, settings, and plugins
  - Contains many levels
  - Packaged into the final build
- Level = one scene inside the project
  - Saved as a .umap asset
  - Holds placed Actors, lighting, and terrain
  - References assets rather than containing them
- Analogy: the project is like an ArcGIS Pro project; a level is like one map or scene inside it
- Deleting an Actor from a level does not delete the asset from the project

This is the distinction I most want you to hold onto, because nearly every early confusion traces back to it. The project is the container for everything. A level is one scene inside that container. The analogy in the middle is usually what makes it click for people coming from GIS: the project is the ArcGIS Pro project, and a level is one of the maps or scenes inside it. The consequence at the bottom is the practical payoff. When you drag a mesh from the Content Browser into the viewport, you are not moving it; you are creating a reference to it. Delete that object from the level and the asset is still in the project, ready to be placed again. Conversely, deleting the asset from the Content Browser will break every level that used it, and Unreal will warn you before it does.

Assets and the Content Browser

- Asset = a definition stored in the project
- Imported files are converted to .uasset on import
  - FBX, OBJ, glTF, PNG, WAV and similar formats
  - The original file is not linked; re-import to update
- Assets reference each other: mesh to material to texture
  - Reference Viewer displays the dependency graph
- Rename and move assets inside the editor only
  - Moving files in the operating system breaks references permanently
- Migrate copies an asset and its dependencies to another project
- Establish a consistent folder structure early

Two rules about assets will save a great deal of pain. First, importing is a conversion, not a link. When you import an FBX from Blender, Unreal reads it and writes its own .uasset file. Changing the FBX on disk afterwards does nothing until you explicitly re-import, and the re-import command is on the asset's right-click menu. Second, and this is the one that causes real damage: never move or rename asset files using Windows Explorer or the Finder. Unreal tracks references by path. Rename inside the editor and it updates every reference automatically. Rename outside the editor and you have silently broken every level that used that asset, often in a way that does not surface until much later. The Reference Viewer is worth demonstrating, because seeing the dependency graph makes the relationships concrete. The Migrate command is the safe way to move work between projects, since it copies an asset along with everything it depends on.

Actors

- Actor = anything that can be placed in a level
- Every Actor has a Transform: location, rotation, and scale
- Units are centimeters; a 100-unit cube is one meter
- Examples
  - Static Mesh Actor, light, camera, volume
  - Player Start, trigger, Blueprint Actor
- The asset is the definition; the Actor is an instance in a level
- Many Actors can share a single asset at almost no memory cost

Actor is the word Unreal uses for anything that can be placed in a level. It is deliberately broad: a rock, a light, a camera, a trigger region, and the player's own character are all Actors. The one thing they share is a Transform, which is a location, rotation, and scale. Pay attention to the units. Unreal works in centimeters, so the default cube that reads as one hundred units is one meter on a side. This matters enormously when importing geospatial data, because if your source is in meters you will import something a hundred times too small, and the error is easy to miss until you try to walk around in it. The final two points restate the asset and instance distinction in Unreal's own vocabulary. Fifty placed copies of a tree cost you one mesh in memory and fifty transforms, which is why populating a scene is cheaper than expected.

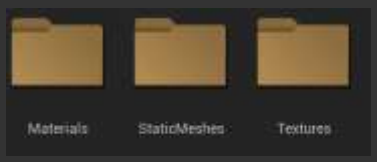
Actor Components

- Components add capability to an Actor
- Common components
  - Static Mesh Component = geometry
  - Light Component = illumination
  - Audio Component = sound
  - Collision Component = an interaction volume
  - Widget Component = user interface placed in the world
- One component is the root; others attach beneath it
- Each component has a transform relative to the Actor
- The Details panel changes with the selected component, not just the Actor
- Capabilities are composed rather than inherited

If Actor is the noun, components are the parts that give it capability. This is a compositional design: rather than having a dedicated class for a lamp post, you build an Actor and attach a mesh component for the geometry, a light component for the illumination, an audio component for a faint hum, and a collision component if you want it to detect the visitor walking past. The practical implication is in the Details panel, and it is a common source of confusion. When a property you expect is not there, you have very likely selected the Actor when you needed the component, or the reverse. The component list sits at the top of the Details panel, and clicking between entries changes everything below it. Note also that each component carries its own transform relative to the Actor, which is how you position a light above a pole without moving the whole object.

Static vs. Skeletal Meshes

- Static Mesh
  - Rigid geometry; vertices do not move relative to one another
  - Terrain, buildings, props, photogrammetric meshes
  - Cheapest to render; supports instancing
- Skeletal Mesh
  - Bound to a skeleton of bones with skin weights
  - Characters, animals, animated machinery
  - Requires an Animation Blueprint to drive it
- Nearly all geospatial content is static
- Collision is stored on the mesh asset, not on the placed Actor



Unreal divides meshes into two categories, and the distinction is about deformation rather than complexity. A Static Mesh is rigid: it can move, rotate, and scale as a whole, but its vertices do not move relative to one another. That covers essentially all geospatial content, including terrain, buildings, photogrammetric meshes, vegetation, and props. The word static refers to the geometry, not to whether the object can move, which is a point worth making explicitly because the name misleads people. A Skeletal Mesh is bound to a hierarchy of bones with skin weights, so its surface deforms as the bones move, and you need one for a character with an animated body. In a typical walkthrough project the only skeletal mesh is the avatar, and it will come from the template. The last point is a forward reference to the collision slide: collision is a property of the mesh asset, which surprises people who expect to configure it on the placed object.

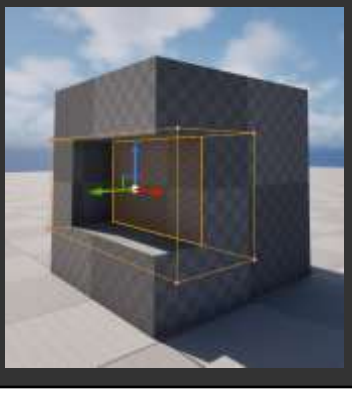
Nanite

- Virtualized geometry system introduced in Unreal Engine 5
- Streams and renders only the triangles that contribute to the view
- Continuous level of detail at roughly pixel scale
  - No manually authored LODs and no visible popping
- Makes dense photogrammetric and scanned meshes practical
- Enabled per mesh asset in the Static Mesh editor
- What it does not solve
  - Texture memory, which is often the real bottleneck
  - Collision, which is still generated separately
  - Limited support for transparency and skeletal meshes

Nanite is the single most important recent change for anyone bringing scanned or photogrammetric geometry into a game engine. Historically you had to decimate aggressively and hand-author levels of detail (LODs), which meant a reality capture mesh was either unusable or degraded past the point of being worth capturing. Nanite virtualizes geometry: it streams and draws only the triangle clusters that actually contribute to what you are seeing, at roughly one triangle per pixel, and it does this continuously, so there is no visible popping as you move. Enabling it is a checkbox on the mesh asset. The limitations at the bottom deserve emphasis: Nanite does not make performance problems disappear. It does not address texture memory, and on a photogrammetric mesh the texture atlas is very often what is actually filling your video memory. It also does not give you collision, which is generated separately, and that is the subject of a later slide.

Brushes (BSP Geometry)

- Primitive geometry built inside the editor rather than imported
- Additive brushes add solid volume to the world
- Subtractive brushes carve volume away
  - How a doorway or window opening is cut
- Shapes: box, cylinder, cone, sphere, and stairs
- Geometry Editing mode adjusts vertices and faces directly
- A finished arrangement can be converted to a Static Mesh
- Legacy technology, retained for fast blockouts
- Modeling Mode is the modern alternative and outputs Static Meshes



Brushes, or BSP geometry, are Unreal's original in-editor modeling system. They work additively and subtractively, which is closer to constructive solid geometry than to polygon modeling: you place a solid box to add volume, then place a subtractive box to carve a doorway out of it. The appeal is speed. You can rough out the shape and scale of a building in a few minutes without leaving the editor, which is exactly what you want when you are testing whether a space feels right before committing to modeling it properly. This is legacy technology. It is still present and still useful, but Epic's direction is the Modeling Mode toolset, which does similar things and produces real Static Meshes. I include brushes here mainly because you will encounter them in older tutorials and need to recognize what you are looking at.

Brushes vs. Meshes

- Brushes
  - Fast blockout and testing of spatial layout
  - Simple architectural volumes
  - Poor UV control; difficult to texture well
  - No instancing, no Nanite, inefficient collision
- Meshes
  - Anything imported from Blender, ArcGIS Pro, or a scanner
  - Custom materials and deliberate UVs
  - Instancing, Nanite, and efficient collision
  - All terrain and photogrammetric content
- Rule of thumb: brushes are scaffolding, meshes are construction material

The short version of this slide is the last line: brushes are scaffolding and meshes are construction material. Use a brush when you are trying to answer a question about space. Is this room the right size? Does this corridor feel too narrow? Where should the building sit relative to the terrain? Those questions are worth answering before you spend hours modeling. Once you have the answer, build the real thing as a mesh. The performance points are the practical reason. Brushes are recomputed by the editor, they do not get instanced, they do not benefit from Nanite, and their collision is inefficient. A scene built entirely from BSP will run measurably worse than the equivalent built from static meshes, and it will be harder to texture properly because the UV control is poor.

Materials

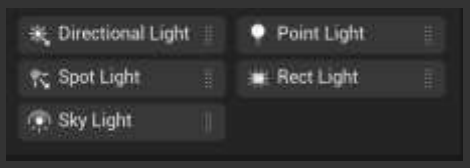
- Material = a shader graph defining how a surface responds to light
- Physically based rendering (PBR) inputs
  - Base Color = surface color with lighting removed
  - Metallic = conductor or dielectric
  - Roughness = how scattered the reflection is
  - Normal = fine surface detail without added geometry
  - Emissive, Opacity, and Specular
- Textures supply image data to these inputs
- Material Instance = a child exposing selected parameters
  - Updates instantly; editing a material recompiles the shader
- Master material pattern: one parameterized parent, many instances

35

Materials in Unreal are node graphs that compile into shaders. The physically based rendering inputs will look familiar from the Blender portion of this course, because PBR is a shared standard rather than an Unreal invention, and the same base color, metallic, roughness, and normal inputs appear in both. The second half of the slide is what is specific to working efficiently in Unreal. Editing a material means recompiling a shader, which is slow and gets slower as the project grows. A Material Instance is a lightweight child that exposes chosen parameters as simple controls, and changing those is instantaneous. The professional workflow, which I would encourage from the start, is to build one flexible master material with parameters exposed, then create instances for every variation you need. Users who skip this and author fifty separate materials will spend a great deal of time watching a compilation progress bar.

Lighting

- Light types
  - Directional = the sun; parallel rays, only rotation matters
  - Sky Light = captures the sky as ambient illumination
  - Point, Spot, and Rect = local sources
- Emissive materials also contribute light through Lumen
- Mobility
  - Static = baked into lightmaps; cheapest; cannot change
  - Stationary = fixed position; color and intensity can vary
  - Movable = fully dynamic; no lighting build required
- Lumen = real-time global illumination and reflections
- Baked lighting needs lightmap UVs, which imported meshes rarely have
- Recommended for geospatial scenes: Movable lights with Lumen



Five light types cover everything you will need. The Directional Light is the one to understand first, because it is your sun: its rays are parallel and arrive from infinitely far away, which means only its rotation matters. Moving a directional light does nothing at all, which confuses people the first time. The Sky Light is the quiet hero of a good-looking outdoor scene, because it captures the sky and surroundings and feeds that back as ambient illumination, which is what stops your shadows from being solid black. Mobility is the biggest decision on this slide. Historically, you baked as much lighting as possible, because dynamic lighting was expensive. Lumen has changed that calculation substantially by providing real-time global illumination, so light bounces between surfaces and you get color bleeding and soft indirect light without baking. My recommendation for this course is at the bottom, and the reason is practical: baked lighting requires a second UV channel laid out for lightmaps, and imported meshes, especially photogrammetric ones, almost never have usable lightmap UVs. Using movable lights with Lumen avoids the problem entirely.

Sky and Atmosphere

- Sky Atmosphere = physically based atmospheric scattering
  - Driven by the angle of the Directional Light
- Volumetric Cloud = three-dimensional clouds that cast shadows
  - Visually convincing but expensive
- Exponential Height Fog = aerial perspective and depth cues
- Sky Light = feeds the resulting sky back as ambient light
  - Set to real-time capture if the sun moves
- Post Process Volume = exposure, color grading, and lens effects
  - Set Exposure to Manual and enable Infinite Extent

37

A convincing outdoor scene is not one Actor but several working together, and the order matters. Sky Atmosphere performs physically based atmospheric scattering, which is what makes the sky blue overhead and orange near the horizon, and it responds automatically to the angle of your directional light, so moving the sun changes the sky without any further work. Volumetric clouds are genuine three-dimensional volumes that the sun lights and that cast shadows on the terrain below. They look superb and they are expensive, so for a large walkthrough a simple sky texture may be the better trade. Exponential Height Fog is the one people skip and should not. Aerial perspective, the way distant terrain washes out and goes slightly blue, is a major depth cue, and without it a large landscape reads as flat and toy-like. The post process settings at the bottom are worth doing in the first ten minutes of any new project. Auto-exposure continuously rebalances brightness the way an eye does; it is realistic and it makes it impossible to judge your lighting, and any recorded walkthrough will visibly pulse as you turn.

Geographically Accurate Sunlight

- Sun Position Calculator plugin drives the Directional Light
- Inputs
  - Latitude and longitude
  - Date, time of day, time zone, and daylight saving
  - North offset to align the scene to true north
- Produces a sun angle correct for that place and moment
- Enables
  - Shadow studies for a proposed structure
  - Solar access and overshadowing assessment on real terrain
  - Seasonal comparison, such as solstice against equinox
  - Animated time-of-day sequences
- Same astronomical basis as solar analysis tools in GIS

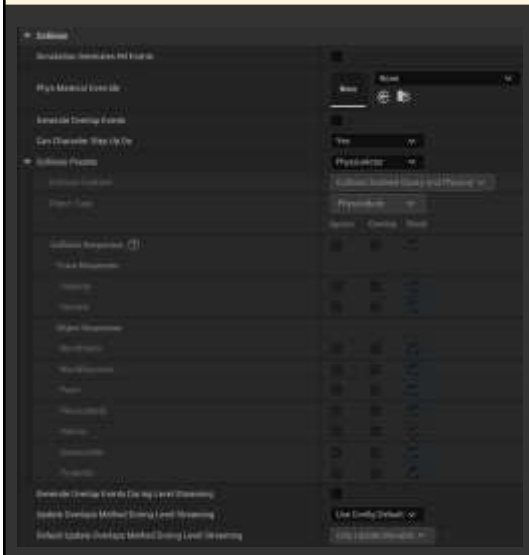
The Sun Position Calculator is a plugin that ships with the engine, and it drives the directional light from latitude, longitude, date, time, and time zone. That is the same astronomical calculation sitting behind a hillshade or a solar radiation tool in GIS, but here it is driving a real-time render you can walk around inside. The consequence is that shadows in your scene are not artistic choices; they are correct for that place at that moment. That makes shadow studies, solar access assessments, and seasonal comparisons defensible outputs rather than illustrations. The one thing you must get right is the north offset. If your imported model is not aligned to true north, every shadow will be wrong by exactly that angle, and the result will look entirely plausible while being wrong, which is the most dangerous kind of error.

Physics

- Chaos is the physics solver in Unreal Engine 5
- Simulate Physics enables rigid body motion; it is opt-in per component
- Requirements
  - Mobility set to Movable
  - The mesh must have collision geometry
  - Gravity enabled separately
- Mass is derived from volume and density, or overridden directly
- Physics Constraints link two bodies: hinge, slider, or ball joint
- A geospatial walkthrough needs very little simulation

Physics in Unreal is handled by the Chaos solver, and it is opt-in. An Actor does nothing physical until you tick Simulate Physics on its component, and there are three prerequisites people routinely miss: the mobility has to be Movable, the mesh needs collision geometry, and gravity is a separate toggle. Physical Materials deserve a moment because the name is misleading. They have nothing to do with appearance; they carry friction and restitution, and they also carry a surface type that you can use elsewhere, most usefully to select the right footstep sound for grass as opposed to gravel. The point at the bottom is the one I want to land for this course. For the kind of scene we are building you need very little simulation. Terrain and buildings should remain static. What you actually need is collision so the avatar does not fall through the ground, and that is a separate system, which is next.

Collisions



- Collision geometry is separate from the visible mesh
- Simple collision = boxes, spheres, capsules, convex hulls; fast
- Complex collision = per-triangle against the render mesh; accurate
- Responses
  - Block = physically stops movement
  - Overlap = passes through but fires an event
  - Ignore = no interaction and no event
- Both objects must agree before an interaction occurs
- Collision presets bundle a whole response matrix under one name
- Imported meshes often arrive with no collision, or a single box
  - Use Complex Collision As Simple suits static terrain and scanned meshes
- Player Collision view mode makes collision geometry visible

Collision is one of the most important concepts for a walkthrough project and one of the least intuitive, because the collision geometry is invisible and is not the mesh you can see. Every static mesh carries a separate, simplified representation used for collision tests, and it has to be simplified, because testing a character's movement against several million triangles every frame is not viable. Note the two distinct behaviors. Block physically stops movement; overlap detects entry without stopping anything, and overlap is how you build a trigger that plays a sound or displays information when a visitor walks into a room. The rule that both objects must agree is the source of the most confused debugging. If your trigger is set to overlap but your character is set to ignore that channel, nothing happens and there is no warning. You press play and immediately fall through the terrain, or you cannot move because the whole model has a box around it. The cause is almost always that the imported mesh has no collision or a default box, and for static terrain the fix is to use complex collision as simple. Demonstrate the Player Collision view mode; seeing the invisible geometry makes the whole concept concrete.

Players

- Game Mode = the rules and class list for a level
  - Declares which Pawn, Controller, and HUD are used
- Pawn = the physical body in the world
- Character = a Pawn with a capsule collider and movement component
- Player Controller = receives input and possesses a Pawn
  - Persists even if the Pawn is replaced
- Player Start = an Actor marking where the avatar spawns
  - Without one, the player spawns at the world origin
- First Person and Third Person templates configure all of these

41

For an avatar to walk around your scene, four pieces have to line up, and each has a specific job. The Game Mode holds the rules and, critically, the class list: it declares which Pawn class the player uses and which Controller. The Pawn is the physical body, and the Character subclass is what you almost always want, because it arrives with a capsule collider and a movement component that already knows how to walk, jump, and climb steps. The Player Controller represents the user rather than the body: it receives input and possesses a Pawn. That separation matters if you ever want to switch between a walking avatar and a flying camera, because the controller persists while the pawn is swapped. Player Start marks the spawn point, and forgetting to place one is a rite of passage; you spawn at the world origin, which in a georeferenced scene may be a very long way from your terrain. My advice is the last line: start from a template and modify it rather than assembling this from a blank project.

Game Control

- Enhanced Input system
  - Input Actions define what can happen
  - Input Mapping Contexts bind them to keys, buttons, or axes
- Supports keyboard, mouse, gamepad, and VR controllers
- Character Movement Component handles walking, jumping, and step height
  - Walk speed, jump velocity, gravity scale, and maximum slope
  - Movement modes: walking, falling, flying, swimming
- A flying or spectator mode is useful for reviewing large landscapes
- VR: teleport locomotion causes less motion sickness than smooth movement
- Input Mode determines whether input goes to the game or the interface

Input in current versions of Unreal goes through the Enhanced Input system, which separates what can happen from how it is triggered. An Input Action is an abstract event such as move or jump; an Input Mapping Context binds those actions to specific keys, buttons, or axes. The benefit of the indirection is that supporting a gamepad or a VR controller becomes a matter of adding a mapping rather than rewriting logic. The Character Movement Component is where the feel of walking lives, and the settings there are worth adjusting for a geospatial scene. Default game movement speeds are tuned for entertainment and are considerably faster than a person actually walks, which distorts the sense of scale in a site visualization. Maximum slope angle and step height also matter on real terrain, since default settings may prevent the avatar from climbing ground a person could walk up. A flying or spectator mode is worth adding for reviewing a large landscape quickly. And for VR, teleport locomotion is the safer default, because smooth movement in a headset causes motion sickness in a significant fraction of users.

Audio

- Sound Wave = the imported audio file
- Sound Cue = a node graph adding randomization, looping, and modulation
- MetaSound = the modern procedural audio graph
- Ambient Sound Actor places a sound at a position in the level
- Attenuation
  - Inner radius at full volume and a falloff distance beyond it
  - Shape: sphere, capsule, box, or cone
- Spatialization positions sound in three dimensions for the listener
- Occlusion muffles sound blocked by geometry
- Audio Volumes apply reverb within a region
- Sound Classes and Submixes handle grouping and mixing

Audio is consistently undervalued by people building visualization scenes, and it is the cheapest realism you can buy. A silent walkthrough of a beautiful landscape feels dead; add wind, distant water, and birdsong and the same scene feels like a place. The asset hierarchy is straightforward. A Sound Wave is the imported file. A Sound Cue is a graph that lets you randomize between several waves, modulate pitch, and loop without an obvious seam, and that randomization is what stops a repeated sound from becoming irritating. MetaSounds are the modern and more powerful replacement, and are where Epic is investing. In the level you place an Ambient Sound Actor, and its attenuation settings define how it fades with distance and what shape its audible region takes. Getting the inner radius and falloff distance wrong is the usual mistake: a river is either audible across the entire county or vanishes the moment you step back from it. Occlusion muffles sound blocked by geometry and is convincing, but it runs a trace, so use it selectively.

User Interface Components (Unreal Motion Graphics)

- Widget Blueprint = a user interface asset
  - Designer tab for visual layout
  - Graph tab for the logic behind it
- Common widgets: Text, Button, Image, Slider, Check Box, Progress Bar
- Panels control arrangement
  - Canvas Panel for free positioning
  - Vertical Box, Horizontal Box, and Scroll Box stack children automatically
- Anchors keep widgets positioned when the resolution changes
- Displaying a widget
  - Screen space: Add to Viewport, for menus and readouts
  - World space: Widget Component attached to an Actor
- VR requires world-space widgets
- World-space widgets work well as feature information panels

UMG is Unreal's interface system and it is genuinely pleasant to work with. A Widget Blueprint has two tabs: a Designer tab where you lay elements out visually, and a Graph tab where you wire up what they do. Two things are worth calling out. First, the panel you place widgets inside determines how they are arranged. A Canvas Panel lets you position things anywhere, which is tempting and usually a mistake; vertical and horizontal boxes lay their children out automatically and adapt when the window changes size. Second, anchors. An anchor ties a widget to a point or region of the screen, and if you build an interface at your monitor's resolution without setting anchors, everything will be in the wrong place on a projector. The display section at the bottom is the part most relevant to us. Screen space is the conventional heads-up display approach, but it does not work in VR, because there is no flat screen and anything locked to the view is uncomfortable to look at. A Widget Component places the interface on a surface inside the world, which is required for VR and is often better for our purposes anyway: attach an information panel to a building and the attribute data appears next to the feature it describes, which is much closer to how a map popup behaves.

Geospatial Data in Unreal Engine

The last section brings this back to spatial data specifically: how to get terrain and vegetation into the engine, and how to reconcile Unreal's local coordinate space with real-world coordinates.

Landscape and Foliage

- Landscape = a specialized heightfield terrain, not a Static Mesh
  - Imports a 16-bit greyscale heightmap; export a DEM to that format
  - Horizontal and vertical scaling must be set to match the source
  - Sculpt and smooth interactively in the editor
  - Paint material layers such as rock, grass, soil, and snow
  - Built-in level of detail, streaming, and collision
- Foliage paints instanced meshes across a surface
  - Instancing makes tens of thousands of plants viable
  - Density, scale, and rotation variation per brush
  - Can align to the surface normal or remain upright
  - Procedural Foliage volumes generate distributions from species rules
  - Culling distance is the main performance control

Landscape is Unreal's terrain system, and it is not a static mesh; it is a specialized heightfield with its own level of detail, streaming, and collision handling. The route in from our world is straightforward. Export your DEM as a sixteen-bit greyscale raster and import it as a heightmap. You will need to work out the horizontal and vertical scaling to match the source, and that is arithmetic worth doing carefully rather than adjusting by eye, because a landscape with the wrong vertical exaggeration will look plausible and be wrong. Once it is in, you can sculpt it and paint material layers so that steep slopes show rock and flatter ground shows grass. Foliage is the companion system for vegetation, and the important word is instancing. Rather than placing thirty thousand individual tree Actors, the foliage system draws many copies of a small number of meshes very efficiently, and that is the difference between a forested landscape that runs and one that does not. Culling distance is your main performance lever: beyond a set range, instances simply stop drawing.

Using Real Coordinates

- Unreal has no native coordinate reference system
  - A level is local space measured in centimeters from an arbitrary origin
- Transforms are single precision
  - Large projected coordinates exceed usable precision
  - Symptoms: geometry jitter and flickering surfaces
- Origin rebasing shifts the world to keep the view near zero
- Cesium for Unreal
  - Georeference Actor anchors the scene to latitude, longitude, and height
  - Handles origin rebasing automatically as the user moves
  - Streams 3D Tiles, including your own photogrammetric tilesets
- ArcGIS Maps SDK provides a comparable georeferenced workflow
- Always confirm scale, vertical datum, and north alignment on import

This is the bridge between everything in this lecture and the geospatial data the course is actually about, and it is the challenge we flagged at the very beginning. Unreal has no native notion of a coordinate reference system. A level is local space measured in centimeters from an arbitrary origin. That is fine until you try to place content at real projected coordinates, because Unreal transforms are single precision, and a few hundred thousand meters from the origin the smallest representable step becomes larger than the detail in your model. The symptom is unmistakable: geometry visibly jitters as the camera moves and surfaces flicker against each other. The solution is origin rebasing, which means keeping the region you are looking at near zero and shifting the world as the user travels. You can implement that yourself, but Cesium for Unreal does it automatically, and it also streams 3D Tiles, so your own photogrammetric tilesets can sit alongside global terrain and imagery. The Esri SDK offers a comparable path if your data already lives in ArcGIS. The final line is a checklist item worth repeating every time: confirm scale, vertical datum, and north alignment on import, because all three can be wrong in ways that still look convincing.

This is the end of this lecture module.

Please return to the West Virginia View Webpage for additional content.

Thanks! Hope you found this useful.