# Glossary

## A

**Actor**

An **Actor** is any object that can be placed into a level. Actors are a generic Class that support 3D transformations such as translation, rotation, and scale. Actors can be created (spawned) and destroyed through gameplay code (C++ or Blueprints). In C++, AActor is the base class of all Actors.

There are several different types of Actors, some examples include: StaticMeshActor, CameraActor, and PlayerStartActor.

**AI**

**AI**, or Artificial intelligence, is behavior controlled by a set of programmed instructions, instead of by human input. For example, non-player characters (NPCs) are controlled with AI.

**AIController**

Just as the PlayerController possesses a Pawn as a representation of the player in a game, an **AIController**possesses a Pawn to represent a non-player character (NPC) in a game. By default, Pawns and Characters will end up with a base AIController unless they are specifically possessed by a PlayerController or told not to create an AIController for themselves.

**animation**

**Animation** is a change in a property value over time. Examples include changes in position, rotation, and/or scale, but can also extend to any other property value as well.

"Animation" can refer specifically to sequences created in Unreal Matinee, or to sequences created for Skeletal Meshes in external 3D modeling and animation applications, and then imported into the Unreal Editor. It can even refer to a change over time in an Actor's property (or properties) created by C++ code or Blueprints.

**Animation Sequence**

An **Animation Sequence**, or **AnimSequence**, is an animation asset created in a 3D modeling and animation application for a Skeletal Mesh, and then imported into the Unreal Editor. You can work with Animation Sequences in Persona.

**asset**

An **asset** is an element used to build a game. Examples include content imported into or created in the **Content Browser**, such as Materials, Static Meshes, Textures, Particle Systems, Blueprints, and Sound Cues. Each asset is saved in an individual .uasset file.

## B

**Blueprints**

The **Blueprints Visual Scripting** system (or **Blueprints** for short) is a visual scripting system based on the concept of adding and connecting nodes together to create gameplay elements all from within the Unreal Editor. Flexible and powerful enough to create an entire game without ever having to touch a single line of code, Blueprints provide the ability to use virtually the full range of concepts and tools generally only available to programmers.

**Blueprint Editor**

[INCLUDE#Glossary\_BlueprintEditor]

**Blueprint Interface**

A **Blueprint Interface** is a type of asset in which undefined functions are created. Any Blueprint can implement this Interface and then create its own definition of those functions. This serves as one way for Blueprints to communicate to one another, particularly when one Blueprint interacts with many other Blueprints.

**Blueprint Macro Library**

A **Blueprint Macro Library** is a Blueprint container that holds a collection of macros or self-contained graphs that can be placed as nodes in other Blueprints. These can be time-savers, because they can store commonly used sequences of nodes, complete with inputs and outputs for both execution and data transfer. Macros are shared among all graphs that reference them, but they are auto-expanded into the graph as if they were a collapsed node during compiling. This means that Blueprint Macro Libraries do not need to be compiled, but changes to a macro are only reflected in graphs that reference that macro when the Blueprint containing those graphs is recompiled.

**Brush**

A **Brush** is an Actor that describes a 3D volume that is placed in a level in order to define level geometry (referred to as BSPs) and gameplay volumes. Typically you will use BSP Brushes to prototype or block-out your levels for gameplay testing.

Volumes on the other hand have several uses depending upon the effects attached to them such as: Blocking Volumes (which are invisible and used to prevent Actors from passing through them), Pain Causing Volumes (which causes damage over time to any Actor that overlaps it) or Trigger Volumes (which are used as a way to cause events when an Actor enters or exits them).

## C

**Cascade**

**Cascade** is the particle-system editor in the Unreal Editor. You can access Cascade by creating or opening a particle system in the **Content Browser**. In Cascade, you can add, modify, and remove Particle Emitters, add modules to Particle Emitters, and modify module properties.

**Character**

A **Character** is a subclass of a Pawn Actor that is intended to be used as a player character. The Character subclass includes a collision setup, input bindings for bipedal movement, and additional code for movement controlled by the player.

**cinematic sequence**

A **cinematic sequence**, also just called a cinematic, is an animation that is played in the game. Some cinematic sequences halt gameplay and involve one or more changes in camera viewpoint; these are called **cutscenes**. A cinematic sequence can either be created and controlled by Matinee or imported as a pre-rendered video created in an external application.

**class**

A **Class** defines the behaviors and properties of a particular Actor or Object used in the creation of an Unreal Engine game. Classes are hierarchical, meaning a Class inherits information from its parent Classes (the Classes it was derived or "sub-classed" from) and passes that information to its children. Classes can be created in C++ code or in Blueprints.

**Blueprint Class**

A **Blueprint Class** is an asset that content creators can use to easily add functionality on top of existing gameplay classes. Often shortened as just Blueprint. Blueprints are created inside of Unreal Editor visually, instead of by typing code, and saved as assets in a content package. They essentially define a new class or type of Actor, which can then be placed into maps as instances that behave like any other type of Actor.

**collision**

**Collision** is a programmatic way of preventing objects from overlapping during a physics simulation, giving the illusion of solidity in an object. May also refer to a Collision mesh, which is a simplified version of the geometry, generally existing as an invisible shell or hull around the mesh. This simplification aids in performance, allowing complex objects with high polygon counts to calculate their collisions with simpler geometry, which is faster to process.

"Collision" can also be non-blocking (overlapping). In this case, instead of preventing objects from interpenetrating each other, the collision can simply register that the objects are interpenetrating (overlapping).

A Collision surface is required for an object to be simulated via physics calculations.

**component**

A **Component** is a piece of functionality that can be added to an Actor. Components cannot exist by themselves, however when added to an Actor, the Actor will have access to and can use functionality provided by the Component.

For example, a Spot Light Component will allow your Actor to emit light like a spot light, a Rotating Movement Component will make your Actor spin around, or an Audio Component will make your Actor able to play sounds.

**Components Mode**

In **Components Mode**, you can add components to your Blueprint, manage the details of existing components, and arrange components. The list of components in Components Mode is hierarchical.

Components Mode is one of three modes; the others are Default Mode and Graph Mode.

**Construction Script**

The **Construction Script** runs following the Components list when an instance of a Blueprint Class is created. It contains a node graph that is executed allowing the Blueprint Class instance to perform initialization operations. This can be extremely powerful as actions like performing traces into the world, setting meshes and materials, and so on can be used to achieve context-specific setup. For instance, a light Blueprint could determine what type of ground it is placed upon and choose the correct mesh to use from a set of meshes or a fence Blueprint could perform traces extending out in each direction to determine how long of a fence is needed to span the distance.

The Construction Script controls functionality while working with the object in the Editor, and the Event Graph controls functionality during gameplay; Construction Scripts do not execute during gameplay.

**Content Browser**

The **Content Browser** is the primary area of the Unreal Editor for creating, importing, organizing, viewing, and modifying content assets within Unreal Editor. It also provides the ability to manage content folders and perform other useful operations on assets, such as renaming, moving, copying, and viewing references. The **Content Browser** can search for and interact with all assets in the game.

In the **Content Browser**, you can search and sort assets by type, and you can filter your view of the assets either by choosing specific types of assets in the **Filters** drop-down list, or by typing text in the **Search Assets** box.

**Controller Class**

**Controllers** are non-physical Actors that can possess a Pawn (or Pawn-derived class like Character) to control its actions. A PlayerController is used by human players to control Pawns, while an AIController implements the artificial intelligence for the Pawns they control. Controllers take control of a Pawn with the Possess function, and give up control of the Pawn with the Unpossess function.

## D

**data pin**

**Data pins** are used for taking data into a node or outputting data from a node. Data pins are type-specific and can be wired to variables of the same type (which have data pins of their own) or a data pin of the same type on another node. Like execution pins, data pins are displayed as an outline when not wired to anything, and solid when wired.

Nodes can have any number of input or output data pins. The data pins of a Function Call node correspond to the parameters and return value of the corresponding function.

**Defaults Mode**

The **Class Defaults** or **Defaults** mode displays all of the properties of the Blueprint, including variables contained within the Blueprint and components defined in native code as well as replication, navigation, and collision properties. The values set for these properties in the Defaults mode will be used when new instances of the Blueprintare created.

Defaults Mode is one of three Blueprint Editor modes; the others are Components Mode and Graph Mode.

**Details panel**

The **Details panel** is a panel in the Level Editor that contains information about the properties for various elements. Many of the tools and editors in UE4 contain Details panels: the Level Editor, Matinee, Persona, and so on.

**Director Group**

The **Director Group** is a specific type of Matinee group that controls camera viewpoints. You can use the Director Group to switch camera viewpoints between shots. Each Matinee sequence can have only one Director Group.

## E

**event**

In Blueprints, an **event** is an entry point, or first executed node of visual scripting. This entry point is often triggered by an in-game occurrence. For instance, a player entering a volume can trigger an Overlap event, which can then be used to cause any functionality to happen within a Blueprint, such as opening a door, turning on a light, and so on.

**EventGraph**

The **EventGraph** of a Blueprint contains a node graph that uses events and function calls to perform actions in response to gameplay events associated with the Blueprint. This is used to add functionality that is common to all instances of a Blueprint. This is where interactivity and dynamic responses are setup. For example, a light Blueprint could respond to a damage event by turning off its LightComponent and changing the material used by its mesh. This would automatically provide this behavior to all instances of the light Blueprint.

**execution pin**

**Execution pins** are used to connect nodes together to create a flow of execution. When an input execution pin is activated, the node is executed. Once execution of the node completes, it activates an output execution pin to continue the flow of execution. Execution pins are displayed as an outline when not wired, and solid when wired to another execution pin. Function Call nodes always have only a single input execution pin and a single output execution pin as functions only have one entry point and one exit point. Other types of nodes can have multiple input and output execution pins, allowing for different behavior dependent on which pin is activated.

## F

**FBX file**

**FBX** is a file format developed by Autodesk. This file format is used to import Static Meshes, Skeletal Meshes, and Animation Sequences into the Unreal Editor.

**Flow Control**

**Flow Control** operations allow the flow of execution to be controlled explicitly in Blueprints. This control can be in the form of choosing one branch of the graph to execute based on whether some condition is true, executing a specific branch multiple times, executing multiple branches in a specific order, and so on. The default flow control operations include branches (if statements), loops (for and while), gates, and sequences.

**framework**

The **framework** systems form the foundation of gameplay in the Unreal Engine. The gameplay framework can be divided into three categories: gametype, player, and interface. Each of these is responsible for a particular aspect of the gameplay and each is comprised of one or more systems and classes that define the different parts and how they work together.

## G

**GameMode**

The **GameMode** Class is responsible for setting the rules of the game that is being played. The rules can include how players join the game, whether or not a game can be paused, and level transitions, as well as any game-specific behavior such as win conditions.

You can set the default GameMode in the **Project Settings**, but can override it on a per-Level basis. Regardless of how you choose to implement the GameMode, there is always only one GameMode present per-level. In a multiplayer game, the GameMode only exists on the server and the rules are replicated (sent) to each of the connected clients.

**GameState**

The **GameState** contains the information that you want replicated to every client in a game, or more simply it is 'The State of the Game' for everyone connected.

It often contains information about game score, whether a match has started or not, how many AI to spawn based upon the number of players in the world, and other game specific information.

For multiplayer games, there is one instance of the GameState on each player's machine with the server's instance being the authoritative one (or the one that clients get their updated information from).

**Graph**

The **Graph** is a visual representation of a particular graph of nodes as it shows all of the nodes contained in the graph as well as the connections between them. It provides editing capabilities for adding and removing nodes, arranging nodes, and creating links between

1. Breakpoints can also be set in the Graph to aid in

[debugging](Engine/Blueprints\UserGuide\Debugging)

 Blueprints.

The Blueprint Editor Modes were removed in 4.7. Please see

[Global Variables](Engine/Blueprints/Editor "Global Variables")

 for information about the **Blueprint Editor Interface**.

**group**

In Matinee, a **group** is a container for animation and visual effects for a specific Actor. In the Level Editor, a **group** is an organizational object in which selected Actors can be combined together so they can be transformed as a single unit.

## K

**key, keyframe**

In animation, a **key** or **keyframe** is a record of a given value or values at a specific time. Curves are then used to interpolate between these keyframes, resulting in animated change over time.

**keybindings**

**Keybindings** are custom-defined keyboard shortcuts. In the Unreal Engine, you can bind specific keys or key combinations to execute specific commands or actions.

**Kill Z**

The **Kill Z** is the coordinate along the Z axis at which an Actor is automatically destroyed or players are killed. Setting a Kill Z helps prevent player characters from getting stuck and being unable to respawn when they fall off an edge.

## L

**Level**

A **Level** is a user defined area of gameplay. Levels are created, viewed, and modified mainly by placing, transforming, and editing the properties of the Actors it contains. In the Unreal Editor, each Level is saved as a separate .umap file, which is also why you will sometimes see them referred to as Maps.

## M

**Material**

A **Material** is an asset that can be applied to a mesh to control the visual look of the scene. At a high level, it is probably easiest to think of a Material as the "paint" that is applied to an object. But even that can be a little misleading, since a Material literally defines the type of surface from which your object appears to be made. You can define its color, how shiny it is, whether you can see through the object, and much more.

In more technical terms, when light from the scene hits the surface, a Material is used to calculate how that light interacts with that surface. These calculations are done using incoming data that is input to the Material from a variety of images (textures) and math expressions, as well as from various property settings inherent to the Material itself.

Unreal Engine 4 utilizes a physically-based shading model. This means that rather than defining a Material using arbitrary properties (such as Diffuse Color and Specular Power), you instead use properties more easily relatable to the real world. These include Base Color, Metallic, Specular, and Roughness.

**Matinee**

The **Matinee** animation tool provides the ability to animate the properties of Actors over time, to create either dynamic gameplay or cinematic in-game sequences. The system is based on the use of specialized animation tracks in which you can place keyframes to set the values of certain properties of the Actors in the level. The Matinee Editor is similar to the non-linear editors used for video editing, making it familiar to video professionals.

**Matinee Actor**

A **Matinee Actor** is a specific instance of a Matinee animation, created as an Actor and placed in your scene.

**Migrate Tool**

The **Content Browser** contains the **Migrate Tool**, which allows assets to be copied from one project to another, along with any dependent assets. For instance, you can migrate a Material, and any Texture assets that define that Material will automatically be copied along with it. This is useful when you need to merge or fork projects, or when transitioning from a test environment into a production project.

**Mobility**

**Mobility** is a property of Static Mesh and Light Actors. The Mobility property, which is set in the **Details** panel, determines whether an Actor can move during gameplay. In addition, for Light Actors, the Mobility property determines the types of light and shadows cast. Mobility property settings include Static, Stationary (Light Actors only), and Movable.

In the World Outliner, an Actor's Mobility setting is indicated by a small colored dot on the left: orange for Movable, yellow for Stationary, none for Static.

Actors with a Movable setting have the most versatility but the highest performance hit; Actors with a Static setting have the least versatility but the lowest performance hit.

**Modes panel**

**Modes** change the primary behavior of the Level Editor for a specialized task, such as moving and transforming assets in the world, sculpting landscapes, generating foliage, creating geometry brushes and volumes, and painting on meshes. Modes panels contain a selection of tools tailored to the selected editing mode.

**Movable**

**Movable** is one possible Mobility property settings. This setting can be applied to both Static Mesh and Light Actors.

**Movable Static Mesh Actors** can be moved during gameplay, and can interact with other Actors.

**Movable Lights** are lights that can be moved and changed in game. Typical uses would be headlights on a car, flashlights, light fixtures that sway in the wind, and so on. Movable Lights are completely dynamic, and do not support indirect lighting at the moment. Movable lights use dynamic shadowing for the entire scene. As a result, shadow casting movable lights can cost 20x more than a Stationary light.

Of the three light mobilities, Movable lights tend to have medium quality, highest mutability, and the highest performance cost.

Movable Lights can change position, rotation, color, brightness, falloff, radius, and just about every other property they have.

**Move tool**

The **Move tool** is a transformation widget that you can use to move (translate) Actors in your scene along the X, Y, and Z axes.

## N

**node**

A **node** is a single block of visual scripting code, generally seen as a shape into which various data and/or execution wires are connected.

## O

**Object**

The base building blocks in the Unreal Engine are called **Objects** and contain a lot of the essential "under the hood" functionality for your game assets. Just about everything in Unreal Engine 4 inherits (or gets some functionality) from an Object. In C++, UObject is the base class of all objects; it implements features such as garbage collections, metadata (UProperty) support for exposing variables to the Unreal Editor, and serialization for loading and saving.

## P

**panel**

A **panel** is a part of the UI that contains information about your level and can be moved, docked, or dragged into a separate window. Examples include the **Details** panel and the **Content Browser**.

**Pawn**

**Pawns** are a subclass of Actor and serve as an in-game avatar or persona, for example the characters in a game. Pawns can be controlled by a player or by the game's AI, in the form of non-player characters (NPCs).

When a Pawn is controlled by a human or AI player, it is considered as Possessed. Conversely, when a Pawn is not controlled by a human or AI player it is considered as Unpossessed.

**Persona**

**Persona** is the animation editing toolset within Unreal. It is a robust system for editing Skeletons, Skeletal Meshes, Animation Blueprints, and various other animation assets. Most (if not all) of your animation work within Unreal will take place in this editor. This is a multi-purpose tool that contains panels for editing skeletal

[Sockets](Engine/Content\Types\SkeletalMeshes\Sockets)

, previewing [Animation Sequences](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/Sequences/index.html) , setting up animation [Blend Spaces](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/Blendspaces/index.html) and [Montages](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/AnimMontage/index.html) , editing [Animation Blueprints](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/AnimBlueprints/index.html) , and much more.

You can open Persona by **double-clicking** any animation-related asset, such as Skeletal Meshes, Blend Spaces, Animation Blueprints, and so on.

**PhAT**

**PhAT**, the Physics Asset editing tool in the Unreal Editor, is used to modify physics and collision properties for Skeletal Mesh Actors.

**Physical Material**

**Physical Materials** are used to define the response of a physical object when interacting dynamically with the world. Physical Materials are fairly easy to use. Creating one will give you a set of default values, identical to the default physical material that is applied to all physics objects. Examples of this would be a character's dead body (ragdoll), a movable crate, and so on.

**pin**

A **pin** is part of a visual scripting node. Pins connect to other nodes via wires. Execution pins must be hooked up in order for a node to execute. Data pins are color-coded according to type, and are used to input data from or output data to other nodes.

**pivot point**

The **pivot point** (or origin point) indicates the location on a Static Mesh Actor or a Skeletal Mesh Actor that has been defined in the 3D modeling program in which it was created as the object's base for alignment purposes. The pivot point can be in the center of an object, at its base, or even outside it.

**Play In Editor (PIE)**

**Play In Editor (PIE)** is a mode in which you can play your level directly in the Unreal Editor window, starting in a location you specify, for testing purposes. In this mode, you can play the current level exactly as it would play in-game, so that you can test gameplay functionality, including player controls and level events triggered by players' actions.

**PlayerController**

The **PlayerController** Class is used to take player input and translate that into interactions in the game and every game has at least one PlayerController in it. A PlayerController often possesses a Pawn or Character as a representation of the player in a game.

The PlayerController is also the primary network interaction point for multiplayer games. During multiplayer play, the server has one instance of a PlayerController for every player in the game since it must be able to make network function calls to each player. Each client only has the PlayerController that corresponds to their player and can only use their PlayerController to communicate with the server.

**PlayerState**

A **PlayerState** is the state of a participant in the game, such as a human player or a bot that is simulating a player. Non-player AI that exists as part of the game world would not have a PlayerState.

Example data that would be appropriate in a PlayerState include player name or score, their current level or health, or whether they are currently carrying the flag in a Capture the Flag game.

For multiplayer games, PlayerStates for all players exist on all machines (unlike PlayerControllers) and can replicate data from the server to the client to keep things in sync.

**project**

A **Project** is a self-contained unit that holds all the content and code that make up an individual game and coincides with a set of directories on your disk. For example, in the image below the Hierarchy Tree of the **Content Browser**contains the same directory structure found inside your Project folder on your disk.

Although a Project is often referenced by the .uproject file associated with it, they are two separate files that exist alongside each other. The .uproject is a reference file used to create, open, or save a file, whereas the Project contains all of the files and folders associated with it.

You can create any number of different Projects which can all be maintained and developed in parallel. Both the Engine (and Editor) can easily switch between them which will allow you to work on multiple games at once or have several test projects in addition to your main game Project.

## R

**Rotate tool**

The **Rotate tool** is a transformation widget that you can use to rotate Actors in your scene along the X, Y, and Z axes.

## S

**safe frame**

In Matinee, the **safe frame** is the view, defined by the viewpoint camera's field of view, that will be seen during your cinematic.

**Scale tool**

The **Scale tool** is a transformation widget that you can use to scale Actors in your scene along the X, Y, and Z axes.

**Simulate In Editor (SIE)**

Using the **Simulate** button begins a **Simulate In Editor** session in the currently active viewport. While simulating, gameplay begins, including the execution of Blueprints and C++ code that do not rely on a player's interaction with the game. While simulating, you have full access to the Editor's tools, so you can modify the scene and its contents, or even place new Actors. You can also select and inspect Pawns controlled by AI right as they are performing actions, and quickly debug and tweak gameplay behaviors. However, because you are not using a PlayerController while simulating, you cannot enter game controls. You can save certain changes made in a Simulate In Editor session using [Keep Simulation Changes](https://docs.unrealengine.com/en-US/BuildingWorlds/LevelEditor/InEditorTesting/KeepSimulationChanges/index.html) .

**Skeletal Mesh Actor**

A **Skeletal Mesh Actor** is a 3D model asset that can be animated in an external 3D modeling program and then imported into the Unreal Editor. A Skeletal Mesh Actor consists of two parts: the mesh itself and a hierarchical skeleton whose bones can be used to create animation by deforming the mesh according to rules defined in the 3D modeling program. You can import Skeletal Mesh Actors as assets in the **Content Browser**.

**socket**

Commonly in games, you will want to attach an object to the bone of a character. This might be a weapon attached to a hand, or a hat attached to the head. For this purpose, Unreal Engine allows you to create **Sockets** in its animation toolset - [Persona](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/Persona/index.html) - which are offset from a bone within the Skeletal Mesh. Sockets can then be translated, rotated, and scaled relative to the bone. Static Meshes and/or Skeletal Meshes can also be previewed attached to sockets. This allows content creators to easily set up sockets for a Skeletal Mesh, and then tell the programmer the names of the sockets to attach objects to.

**sound cue**

A **sound cue** is an audio asset that contains a reference to an audio file created in an external program that has been imported into a project. You can modify the audio output based on the imported audio file in the Sound Cue Editor.

**Static**

**Static** is one of three possible Mobility property settings. This setting can be applied to both Static Mesh and Light Components.

**Static Lights** are lights that cannot be changed or moved in any way at runtime. They are calculated only within Lightmaps, and once processed, have no further impact on performance. Movable objects cannot integrate with static lights, so the usefulness of static lights is limited.

Of the three light mobilities, Static lights tend to have medium quality, lowest mutability, and the lowest performance cost.

**Static Mesh**

INCLUDE:#Glossary\_Static\_Mesh

**Stationary**

**Stationary** is one of three possible Mobility property settings. This setting can be applied only to Light Components.

**Stationary Lights** are lights that are intended to stay in one position, but are able to change in other ways, such as their brightness and color. This is the primary way in which they differ from Static Lights, which cannot change in any way during gameplay. However, it should be noted that runtime changes to brightness only affect the direct lighting. Indirect (bounced) lighting, since it is pre-calculated by Lightmass, will not change.

Of the three light mobilities, Stationary lights tend to have the highest quality, medium mutability, and medium performance cost.

## T

**tangent handle**

A **tangent handle** is a tool used in keyframe editing to adjust the shape and curve of the lines connecting two keyframes. You can find tangent handles attached to the keys in Curve Editors.

**Texture**

**Textures** are images that are used in Materials. They are mapped to the surfaces the Material is applied to. Either Textures are applied directly - for example, for Base Color textures - or the values of the Texture's pixels (or texels) are used within the Material as masks or for other calculations. In some instances, Textures may also be used directly, outside of materials, such as for drawing to the HUD. For the most part, Textures are created externally within an image-editing application, such as Photoshop, and then imported into Unreal Editor through the **Content Browser**. However, some Textures are generated within Unreal, such as Render Textures. These generally take some information from the scene and render it to a Texture to be used elsewhere.

**time cursor**

The **time cursor** is the black vertical bar at the bottom of the timeline that indicates the position in time of the animation's playback. You can drag the time cursor back and forth to preview the animation in the 3D viewport.

**timeline**

A **timeline** is a visual representation of time in Matinee, which you can use to set and modify keys for keyframe animation, and to preview your cinematic. Different from the Timeline node in Blueprints.

**Timeline node**

**Timeline nodes** are special nodes within Blueprints that allow for simple time-based animation to be quickly designed and played back based on in-game events. Timelines are somewhat like simple Matinee sequences in that they allow simple values to be animated and events to be fired off over time. These can be edited directly inside the Blueprint editor by **Double-clicking** on the Timeline in the Graph tab or in the My Blueprint tab. They are specifically built for handling simple, non-cinematic tasks such as opening doors, altering lights, or performing other time-centric manipulations to Actors within a scene.

## U

**Unreal Frontend**

**UnrealFrontend** (UFE) is a tool intended to simplify and speed up daily video game development and testing tasks, such as preparing game builds, deploying them to a device, and launching them. UFE is designed to be the central interface for all game deployment, profiling, and testing tasks.

## V

**viewport**

The **viewport** is the visual representation of your level in the Level Editor. While working on your level, you can use a 3D viewport, 2D viewports, or a combination. Viewports contain a variety of tools to help you see exactly the data you need.

## W

**widget**

A **widget** is a visual tool used in the Level Editor. Widgets typically have a visual representation in the viewport. For example, the Move tool has red, green, and blue arrows indicating the X, Y, and Z axes.

**wire**

A **wire** is a line connecting two nodes in the graph section of a visual script, such as a Blueprint. You create wires by dragging from the pins on nodes. You can drag a wire from a pin on one node to a pin on another node, or you can drag from a pin to an empty space, and use the context menu that is created to select another node to add to your graph. Like pins, wires are color-coded.

**World**

A **World** contains a list of Levels that are loaded. It handles the streaming of Levels and the spawning (creation) of dynamic Actors.

# **Levels**

|  |  |
| --- | --- |
| **Level** **Editor**  The Level Editor is the primary Editor used to construct your gameplay levels. Generally speaking it is where you define the play space for your game by adding different types of [Actors and Geometry](https://docs.unrealengine.com/en-US/Basics/Actors/index.html) , [Blueprints](https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html) , [Cascade Particle Systems](https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/index.html) or anything else that you want to add to your level. By default, when you create or open a project it will open up to the Level Editor. | |
| **Material Editor** |  |
| A picture containing text, indoor, electronics, computer  Description automatically generated | The Material Editor is where you can create (or edit existing) Materials which are assets that can be applied to a mesh to control its visual look. For example, you can create a "dirt" Material and apply it to floors in your level or terrain to create a dirt looking surface. |
| **Blueprint** **Editor** |  |
|  | The Blueprint Editor is where you can work with and modify Blueprints, which are special assets that can be used to create new types of Actors and script level events, without needing to write any form of C++ code. |
| **Behavior Tree Editor** |  |
|  | Inside the Behavior Tree Editor you can script Artificial Intelligence through a visual node-based system (similar to Blueprints) for Actors in your levels (this could be any number of different behaviors for enemies, NPC characters, vehicles, etc.) |
| **Persona Editor** |  |
|  | The Persona Editor is the animation editing toolset within Unreal Engine 4 and is used for editing [Skeleton Assets](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/Skeleton/index.html) , [Skeletal Meshes](https://docs.unrealengine.com/en-US/WorkingWithContent/Types/SkeletalMeshes/index.html) , [Animation Blueprints](https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/AnimBlueprints/index.html) and various other animation assets. Most (if not all) of your animation work within Unreal Engine 4 will take place in this Editor. |
| **Cascade Editor** |  |
|  | [Cascade Particle Systems](https://docs.unrealengine.com/en-US/RenderingAndGraphics/ParticleSystems/index.html) in Unreal Engine 4 are edited inside the Cascade Editor, which is a fully integrated and modular particle effects editor. Cascade offers real-time feedback and modular effects editing, allowing fast and easy creation of even the most complex effects. |
| **Niagara Editor** |  |
|  | Create special effects in the Niagara Editor using a fully modular particle effects system comprised of separate emitters for each effect. Emitters can be saved in the Content Browser for future use and used as the basis of new emitters in your current or future projects. |
| **UMG UI Editor** |  |
|  | The UMG UI Editor is a visual UI authoring tool which can be used to create UI elements such as in-game HUDs, menus or other interface related graphics you wish to present to your users. |
| **Matinee Editor** |  |
|  | The Matinee Editor is based on the use of specialized animation tracks in which you can place keyframes to set the values of certain properties of the Actors in your level. This allows you to create in-game cinematics, dynamic gameplay events or even animate the properties of Actors over time (such as driving the brightness of a light). |
| **Sound Cue Editor** |  |
|  | The behavior of audio playback in Unreal Engine 4 is defined within Sound Cues which can be edited using the Sound Cue Editor. Inside the Sound Cue Editor, you can combine and mix several sound assets to produce a single mixed "output" saved as a Sound Cue. |
| **Paper2D Sprite Editor** |  |
|  | The Paper2D Sprite Editor enables setup and editing of individual [Paper 2D Sprites](https://docs.unrealengine.com/en-US/AnimatingObjects/Paper2D/Sprites/index.html) (which are in essence a quick and easy way to draw 2D images in Unreal Engine 4). |
| **Paper2D Flipbook Editor** |  |
|  | With the Paper2D Flipbook Editor you can create 2D animations called Flipbooks. By specifying a series of Sprites along certain key frames inside the Paper2D Flipbook Editor, those frames are then "flipped" through to create an animation. The best way to think of Flipbooks is of the old hand-drawn style of animation. |
| **Physics Asset Tool Editor** |  |
|  | The Physics Asset Tool (or PhAT for short) is used for creating Physics Assets for use with [Skeletal Meshes](https://docs.unrealengine.com/en-US/WorkingWithContent/Types/SkeletalMeshes/index.html) . You can start from nothing and build to a full ragdoll setup or use the automation tools to create a basic set of Physics Bodies and Physics Constraints. |
| **Static Mesh Editor** |  |
|  | The Static Mesh Editor is used to preview the look, collision, and UVs as well as set and manipulate the properties of [Static Meshes](https://docs.unrealengine.com/en-US/WorkingWithContent/Types/StaticMeshes/index.html) . Inside the Static Mesh Editor you can also setup [LODs](https://docs.unrealengine.com/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/LODs/index.html) (or Level of Detail settings) for your Static Mesh assets. |
| **Media Player Editor** |  |
|  | The Media Player Editor allows you to define media files or URLs to use as source media for playback inside Unreal Engine 4.  Although not necessarily an "Editor" in that you cannot edit media files in this Editor, you define settings to how your source media will playback such as auto-playing, play rate or looping for example.  You can also see information about your media inside this editor as well as use standard playback controls to view the media. |
| **Font Editor** |  |
|  | The Font Editor allows you to add, organize and preview Font assets. In this editor, you can also define Font parameters, including the hinting and loading policies, and the layout method of the asset. |

