An open-source project

The Terminology Of

Game Development

For game developers, artists and designers

Published on GitHub

Contents

1   Preface

There were a lot of words that I was unfamiliar with, as I started to learn game development.

So I decided to create my own "dictionary" to use as a reference for learning and making this path easier.

Then I thought, why not make this open-source so anyone out there, who has similar issues would benefit from having this?

Therefore I decided to make this "terminology" open for everyone.

For this terminology, I'm using different sources. These sources may include:

- Unreal engine 4 official documentation
- Unreal engine 4 official forums
- Wikipedia
- Autodesk
- Unity engine documentation
- Stackoverflow

2   Contributors

Here's a list of all people who helped me gather this terminology book:

Aien Saidi
aien.saidi@gmail.com
github: @aientech
twitter: @i_in_dev

A

Action Mapping • Action mappings are inputs that only output execution pins. Triggering these events can run new lines of blueprint code. In contrast to Axis Mappings (*See Axis Mapping*) they are executed only once and thus are simpler in the context of the key definition.

Actions that are bound only to a pressed or released event will fire every time any key that is mapped to it is pressed/released. However, in the case of Paired Actions (actions that have both a pressed and a released function bound to them) we consider the first key to be pressed to have captured the action. Once a key has captured the action the other bound keys' press and release events will be ignored until the capturing key has been released.

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

AI • Artificial intelligence (AI), is intelligence demonstrated by machines, unlike the natural intelligence displayed by humans and animals.

Creating Artificial Intelligence (AI) for characters (*See Character*) or other entities in your projects (*See Project*) in Unreal Engine 4 (UE4) is accomplished through multiple systems working together. From a Behavior Tree (*See Behavior Tree*) that is branching between different decisions or actions, running a query to get information about the environment through the Environment Query System (EQS) (*See Environment Query System (EQS)*), to using the AI Perception system (*See AI Perception System*) to retrieve sensory information such as sight, sound, or damage information.

all of these systems play a key role in creating believable AI in your projects. Additionally, all of these tools can be debugged with the AI Debugging tools, giving you insight into what the AI is thinking or doing at any given moment.

AI Perception System • In addition to Behavior Trees (*See Behavior Tree*) which can be used to make decisions on which logic to execute, and the Environmental Query System (EQS) (*See Environment Query System (EQS)*) used to retrieve information about the environment; another tool you can use within the AI framework which provides sensory data for an AI (*See AI*) is the AI Perception System. This provides a way for Pawns (*See Pawn*) to receive data from the environment, such as where noises are coming from, if the AI was damaged by something, or if the AI

sees something. This is accomplished with the AI Perception Component that acts as a stimuli listener and gathers registered Stimuli Sources.

Axis Mapping • In general, Axis Mappings allow us to map keyboard, mouse, and controller inputs to a "friendly name" that can later be bound to game behavior, such as movement. Axis Mappings are continuously polled, allowing for seamless movement transitions and smooth game behavior. Hardware axes (such as controller joysticks) provide degrees of input, rather than discrete input (1 - pressed vs. 0 - not pressed). While controller joystick input methods are effective at providing scalable movement input, Axis Mappings can also map common movement keys, like WASD to continuously-polled game behavior.

B

Behavior Tree ● Behavior Trees assets in Unreal Engine 4 (UE4) can be used to create artificial intelligence (AI) (*See AI*) for non-player characters (*See NPC*) in your projects. While the Behavior Tree asset is used to execute branches containing logic, to determine which branches should be executed, the Behavior Tree relies on another asset called a Blackboard (*See Blackboard*) which serves as the "brain" for a Behavior Tree.

Blackboard ● A blackboard is a simple place where data can be written and read for decision making purposes. A blackboard can be used by a single AI pawn (*See Pawn*), shared by a squad, or used for any other purpose where it's convenient to have a central place to look up relevant data. Blackboards are commonly used with behavior trees, but you could use them separately from behavior trees for convenience or you could make a behavior tree with no need of a blackboard.

Blueprint ● Blueprints is the visual scripting system inside Unreal Engine 4 and is a fast way to start prototyping (*See Prototype*) your game. Instead of having to write code line by line, you do everything visually: drag and drop nodes, set their properties in a UI, and drag wires to connect.

Branch ● The Branch node is a switch node (*See Switch Node*) that serves as a simple way to create decision-based flow from a single true/false condition. Once executed, the Branch node looks at the incoming value of the attached Boolean, and outputs an execution pulse down the appropriate output.
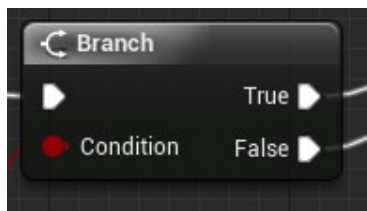
source: Unreal[1]

C

Camera • The Camera represents the player's point of view. how the player sees the world. For this reason, cameras only have relevance to human-controlled players. The `PlayerController` specifies a camera class and instantiates a Camera Actor (*See Actor*) which is used to calculate the position and orientation the player views the world from.

Canvas • The Canvas is an object that can be used during the render loop of the HUD (*See HUD*) to draw elements - text, texture and material tiles, arbitrary triangles, and simple primitive shapes - to the screen. Unless you are making use of some specialized alternative, drawing with the Canvas is the method used to create HUDs and UIs in games made with Unreal Engine.

Character • A Character is a subclass of a Pawn Actor (*See Pawn*) 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.

Class • A Class defines the behaviors and properties of a particular Actor (*See Actor*) or Object (*See 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 (*See Blueprint*).

Component • A Component is a piece of functionality that can be added to an Actor (*See 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 (*See Spot light*) Component will

allow your Actor to emit light (*See 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.

Context • Within the Environment Query System (EQS), Contexts provide a frame of reference for any Tests or Generators (*See Generator*) used. A Context can be as simple as the Querier (who is performing the Test) or more complex such as All Actors of a Type. A Generator, such as a Points: Grid, can use a Context that returns multiple locations or Actors. This will create a grid (of the defined size and density) at the location of each Context. In addition to Engine supplied Contexts, you can create custom Contexts in Blueprint (*See Blueprint*) with the `EnvQueryContext_BlueprintBase` class or through C++ code.

Controller • Controllers are non-physical Actors that can possess a Pawn (*See 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 (*See AI*) 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.

Controllers receive notifications for many of the events occurring for the Pawn they are controlling. This gives the Controller the opportunity to implement the behavior in response to this event, intercepting the event and superseding the Pawn's default behavior. It is possible to make the Controller tick before a given Pawn, which minimizes latency between input processing and Pawn movement.

D

Directional Light • The Directional Light simulates light that is being emitted from a source that is infinitely far away. This means that all shadows cast by this light will be parallel, making this the ideal choice for simulating sunlight.

E

Edge • A connection between two vertices (*See Vertex*).

Environment Query System (EQS) • The Environment Query System (EQS) is a feature within the Artificial Intelligence (*See AI*) system in Unreal Engine 4 (UE4) that is used to collect data from the environment. Within EQS, you can ask questions about the data collected through a variety of different Tests which produces an Item that best fits the type of question asked.

An EQS Query can be called from a Behavior Tree (*See Behavior Tree*) and used to make decisions on how to proceed based on the results of your Tests. EQS Queries are primarily made up of Generators (*See Generator*) (which are used to produce the locations or Actors that will be tested and weighted) and Contexts (*See Context*) (which are used as a frame of reference for any Tests or Generators). EQS Queries can be used to instruct AI characters (*See Character*) to find the best possible location that will provide a line of sight to a player in order to attack, the nearest health or ammo pickup, or where the closest cover point (among other possibilities).

F

Face • A closed set of edges (*See Edge*), in which a triangle face has three edges, and a quad face has four edges.

Flow Control • Nodes that allow for controlling the flow of execution based on conditions. (*See Switch Node*)

G

Game Mode • 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.

Game State • 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 (*See AI*) to spawn (*See 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).

Gameplay Framework • Contains the information about core systems, such as game rules, player input and controls, cameras, and user interfaces.

To show how the different pieces of game framework interlock, here is a simple example. Imagine a simple concept for a game, where a rabbit races a snail. The base of the game framework is the GameMode (*See Game Mode*). The GameMode sets the rules for the game, like the rule that whichever player crosses the fin-

ish line first is the winner. It also handles spawning the players.

A player is set up in a `PlayerController` (*See Controller*), which can possess a Pawn (*See Pawn*). The Pawn is the physical representation of a player in the game, while the Controller possesses the Pawn and can set rules for its behavior. In our example, there would be two Pawns, one for the snail and one for the rabbit. The rabbit would actually be a Character (*See Character*), a special subclass of Pawn which has built-in movement functionality including running and jumping. The snail, on the other hand, has a different style of movement, so it could extend directly from the Pawn class.

A Pawn can contain its own rules for movement and other game logic, but that functionality can also be included in a Controller. A Controller can either be a `PlayerController` taking input from a human player or an `AIController` with automated control by the computer. In this example, the player would

be controlling the snail, so the snail Pawn would be possessed by a `PlayerController`. The rabbit would be controlled by AI (*See AI*), with rules for when it should stop, sprint, or nap all being set up in an `AIController` which possesses the rabbit Character. Only human players care about the view provided by a Camera (*See Camera*), so only one `CameraComponent` in the snail Pawn would be used by the `PlayerCamera`.

During gameplay, input from the player would move the snail around the map, while the HUD (*See HUD*) was overlaid on the view provided by the Camera, showing information about who was currently in first place, and the race time that had elapsed.

Gate • A Gate node is a switch node (*See Switch Node*) that is used as a way to open and close a stream of execution. Simply put, the Enter input takes in execution pulses, and the current state of the gate (open or closed) determines whether those pulses pass out of the Exit
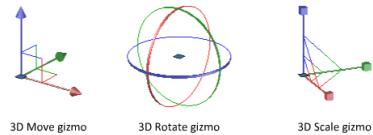
output or not.
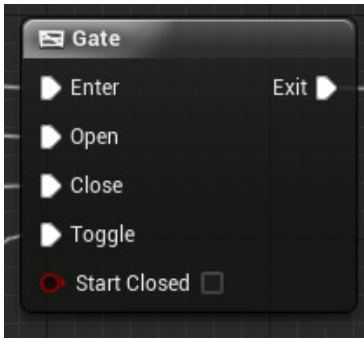
Generator • Within the Environmental Query System (EQS), Generators are used to produce the locations or Actors (*See Actor*) (referred to as Items) that will be tested and weighted, with the highest weight location or Actor being returned to the Behavior Tree. There are different types of Generators that you can use to retrieve information and they can be created either in Blueprint or C++.

Gizmo • 3D gizmos help you move, rotate, or scale a set of objects along a 3D axis or plane.

3D Move gizmo   3D Rotate gizmo   3D Scale gizmo

---

[2]https://docs.unrealengine.com/en-US/Engine/Blueprints/ UserGuide/FlowControl/index.html

[3]https://help.autodesk.com/cloudhelp/2020/ENU/AutoCAD-Core/images/GUID-A67A1239-F373-43DF-A434-0763DC2930AE.png

H

HUD • Heads-up Displays (HUDs) are the games' way of providing information about the game to the player and in some cases allowing the player to interact with the game.

Unreal Engine 4 provides multiple means of creating UIs (*See User Interface (UI)*) and HUDs. The Canvas class (*See Canvas*) can be used to draw directly to the screen at a low level, overlaid onto the world.

The game's Interface is used to convey information to the player and provide a means of prompting the user for directed input. A game interface generally consists of two main elements: the heads-up display (HUD) and menus or user interfaces (UIs).

User Interface (UI) • User Interfaces refer to menus and other interactive elements. These elements are usually drawn overlaid on the screen much like the HUD (*See HUD*), but in certain circumstances they could be part of the game world itself rendered onto a surface (*See Surface*) in the world. The most obvious examples of UIs are the main menu displayed when the game starts up or the pause menu shown when the player has paused the game. However, other UIs may be displayed during play. These could be used to show dialog between characters in the game or in more complex situations, such as in an RTS or RPG, they may be integral to the game play (*See Gameplay Framework*) itself allowing the player to choose weapons, armor, units to build, etc.

L

Level • When playing a video game, every object that you see or interact with resides in what is known as a Level. In Unreal Engine 4 terms, a Level is made up of a collection of Static Meshes (*See Mesh*), Volumes (*See Volume*), Lights (*See Light*), Blueprints (*See Blueprint*) and more all working together to bring the desired experience to the player. Levels in UE4 can range in size from massive terrain-based worlds to very small levels that contain a few Actors (*See Actor*).

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 (*See Actor*) 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.

Light • Unreal Engine 4 has four light types:

- Directional (*See Directional Light*)

- Point (*See Point light*)

- Spot (*See Spot light*)

- Sky (*See Sky light*)

- Rect (*See Rect light*)

Directional lights are primarily used as your primary outdoor light or any light that needs to appear as if it is casting light from extreme or near-infinite distances. Point lights are your classic "light bulb" like light, emitting light in all directions from a single point. Spot lights emit light from a single point, but have their light limited by a set of cones. Sky lights capture the background of your scene and apply it as lighting to your level's meshes.

Lighting Pass • Emphasis is placed on improving lighting (*See Light*), adding PostProcess effects and updating materials (*See Material*).

M

Macro • Blueprint Macros, or Macros, are essentially the same as collapsed graphs of nodes. They have an entry point and exit point designated by tunnel nodes. Each tunnel can have any number of execution or data pins which are visible on the macro node when used in other Blueprints and graphs.

Material • A Material is an asset that can be applied to a mesh (*See 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.

Mesh • In 3D computer graphics and solid modeling, a polygon mesh is a collection of vertices, edges and faces that defines the shape of a polyhedral object. The faces usually consist of triangles (triangle mesh), quadrilaterals (quads), or other simple convex polygons (n-gons), since this simplifies rendering, but may also be more generally composed of concave polygons, or even polygons with holes.
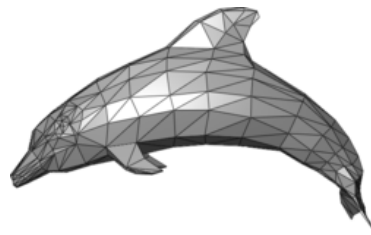
source: Wikipedia[4]

A mesh is consisted of 7 components:

vertices   edges   faces   polygons   surfaces
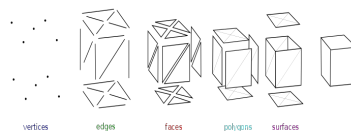
source: Wikipedia[5]

---

[4]https://upload.wikimedia.org/wikipedia/commons/thumb/f/fb/Dolphin_triangle_mesh.png/250px-Dolphin_triangle_mesh.png

[5]https://upload.wikimedia.org/wikipedia/commons/thumb/6/6d/Mesh_overview.svg/720px-Mesh_overview.png

- Vertices (*See Vertex*)

- Edges (*See Edge*)

- Faces (*See Face*)

- polygons (*See Polygon*)

- Surfaces (*See Surface*)

- Materials (*See Material*)

- UV Coordinates (*See UV Coordinates*)

Meshing Pass • Replacing the basic meshes (*See Mesh*) or primitives from the prototype phase (*See Prototype Pass*) with near final asset and applying basic materials (*See Material*).

N

NPC • A non-player character (NPC) is any character (*See Character*) in a game which is not controlled by a player. The term originated in traditional tabletop role-playing games where it applies to characters controlled by the gamemaster or referee rather than another player. In video games, this usually means a character controlled by the computer (instead of the player) that has a predetermined set of behaviors that potentially will impact game play, but not necessarily be true artificial intelligence (*See AI*).

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

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

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.

Point light • Point Lights work much like a real-world light bulb, emitting light in all directions from the light bulb's tungsten filament. However, for the sake of performance, Point Lights are simplified down emitting light equally in all directions from just a single point in space.

Polish Pass • Additional effects, audio and volumes are added, and final assets and details are tweaked.

Polygon • A polygon is a coplanar set of faces (*See Face*). In systems that support multi-sided faces, polygons and faces are equivalent. However, most rendering hardware supports only 3- or 4-sided faces, so polygons are represented as multiple faces. Mathematically a polygonal mesh may be considered an unstructured grid, or undirected graph, with additional properties of geometry, shape and topology.

Project • A Project is a self-contained unit that holds all the content and code that makeup an individual game and coincides with a set of directories on your disk.

Prototype • A first or preliminary version of a device or something from which other forms are developed.

Prototype Pass • The first pass (*See Workflow Pass*) when designing a level (*See Level*) is the prototype (*See Prototype*) pass.

In this pass, a very general level (*See Level*) prototype is laid out using either Brushes or basic geometric Static Meshes such as cubes and spheres. Basic materials (*See Material*) can be applied to any Static Meshes, but this is more for being able to differentiate different objects or areas within your level (*See Level*), rather than for any aesthetic effect. Basic lighting is also added to the level (*See Level*) at this point.

The prototyping pass is meant to be a quick and simple process where the basic gameplay areas are roughed out. This allows for testing gameplay within the level (*See Level*), and observing things like the layout and relative sizing of areas within the level (*See Level*), as well as how that affects players' movement throughout the game. By the end of this pass, you can have a good sense of the playability and environment of your area.

R

Ray tracing • Simply put, ray tracing is a technique that makes light in videogames behave like it does in real life. It works by simulating actual light rays, using an algorithm to trace the path that a beam of light would take in the physical world. Using this technique, game designers can make virtual rays of light appear to bounce off objects, cast realistic shadows, and create lifelike reflections.

Rect light • The Rect Light emits light into the scene from a rectangular plane with a defined width and height. You can use them to simulate any kind of light sources that have rectangular areas, such as televisions or monitor screens, overhead lighting fixtures, or wall sconces.

S

Sky light • The Sky Light captures the distant parts of your level (*See Level*) and applies that to the scene as a light. That means the sky's appearance and its lighting/reflections will match, even if your sky is coming from atmosphere, or layered clouds on top of a skybox, or distant mountains. You can also manually specify a cubemap to use.

Spawn • In video games, spawning is the live creation of a character, item or NPC (*See NPC*). Respawning is the recreation of an entity after its death or destruction, perhaps after losing one of its lives. Despawning is the deletion of an entity from the game world.

Spot light • A Spot Light emits light from a single point in a cone shape. Users are given two cones to shape the light - the Inner Cone Angle and Outer Cone Angle. Within the Inner Cone Angle, the light achieves full brightness. As you go from the extent of the inner radius to the extents of the Outer Cone Angle, a falloff takes place, creating a penumbra, or softening around the Spot Light's disc of illumination. The Radius of the light defines the length of the cones. More simply, this will work like a flash light or stage can light.

Surface • More often called smoothing groups, are useful, but not required to group smooth regions. Consider a cylinder with caps, such as a soda can. For smooth shading of the sides, all surface normals must point horizontally away from the center, while the normals of the caps must point straight up and down. Rendered as a single, Phong-shaded surface, the crease vertices (*See Vertex*) would have incorrect normals. Thus, some way of determining where to cease smoothing is needed to group smooth parts of a mesh (*See Mesh*), just as polygons (*See Polygon*) group 3-sided faces (*See Face*). As

an alternative to providing surfaces/smoothing groups, a mesh may contain other data for calculating the same data, such as a splitting angle (polygons with normals above this threshold are either automatically treated as separate smoothing groups or some technique such as splitting or chamfering is automatically applied to the edge between them). Additionally, very high-resolution meshes are less subject to issues that would require smoothing groups, as their polygons (*See Polygon*) are so small as to make the need irrelevant. Further, another alternative exists in the possibility of simply detaching the surfaces themselves from the rest of the mesh. Renderers do not attempt to smooth edges across noncontiguous polygons.

Switch Node • A switch node

reads in a data input, and based on the value of that input, sends the execution flow out of the matching (or optional default) execution output. There are several types of switches available: Int, String, Name, and Enum.

In general, switches have an execution input, and a data input for the type of data they evaluate. The outputs are all execution outputs. Enum switches automatically generate the output execution pins from the Enum's properties, while Int, String, and Name switches have customizable output execution pins.

There are different nodes than can help you with switching logic:

• Branch (*See Branch*)

• Gate (*See Gate*)

T

Transform • A transform or transformation is composed of three components:

- Scale

- Rotation

- Location

(*See Transforming Actors*)

Transforming Actors • Transforming Actors (*See Actor*)

refers to moving, rotating, or scaling them, which is an important part of the level-editing process. There are two basic ways to transform Actors in the Unreal Editor. Both ways apply the transformations to all currently selected Actors.
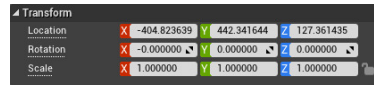
source: Unreal[6]

---

[6]https://docs.unrealengine.com/Images/Engine/UI/LevelEditor/Details/Customizations /Transform/Transform_Properties.webp

U

UV Coordinates • Most mesh (*See Mesh*) formats also support some form of UV-coordinates which are a separate 2d representation of the mesh "unfolded" to show what portion of a 2-dimensional texture map to apply to different polygons (*See Polygon*) of the mesh. It is also possible for meshes to contain another such vertex (*See Vertex*) attribute information such as colour, tangent vectors, weight maps to control animation, etc (sometimes also called channels).

V

Vertex • A position (usually in 3D space) along with other information such as color, normal vector and texture coordinates.

Volume • Volumes are three-dimensional Actors (*See Actor*) used to alter the behavior of areas within levels. Volumes can be used for behaviors like:

- Causing damage to the player or other Actors inside the Volume.

- Blocking certain Actors from entering the Volume, acting as a collision surface.

- Opening a door when an Actor enters the Volume.

- Changing the way a level calculates its lighting or visibility.

W

Workflow Pass ● A phase

3   Document info

This work currently contains 58 words, and 1 people has been contributed to the project.