Game Development Using Unreal Engine

5th Grade Media Technology

Organisation :
Schulzentrum Ybbs an der Donau

Address:	Schulring 1 und 6 3370 Ybbs an der Donau
Tel.:	0 74 12 52 575-11
Fax.:	0 74 12 52 575-16
E-Mail:	hak(at)sz-ybbs.ac.at
www:	www.sz-ybbs.ac.at

Autor **Markus Meyerhofer, MSc BSc BEd**

Contents

Introduction

1.1 History

In 1998, the game development studio Epic Games released a first-person shooter called *Unreal*.The game was a hit and it spawned other successful sequels such as Unreal Tournament; the grandfather of all First Person Shooter / Massive Online Games.

The same year the game was released, Epic Games decided to start licensing the engine it had built to create it, which it simply called the *"Unreal Engine"*. Therefore other game development studios, instead of having to write their own game engines from scratch, could instead pay a fee to use this game engine that *Epic Games* had already built.

So this turned out to be a great business model for *Epic Games* and they continued to invest heavily in developing their game engine. So since 1998, newer versions of the engine have been announced; the latest (stable) version of it is, at the time of writing *Unreal Engine 5.5*, the major version 5 was released in 2022.

So, the Unreal Engine has been used to create countless numbers of blockbuster games and game series including the likes of *Fortnite*, *BioShock*, *Gears of Wars*, *Splinter Cell*, *Rainbow 6*, *Borderlands*, *Dishonored*, *Mass Effect*, the *Batman:Arkham* series and so forth.

The biggest disadvantage of the *Unreal Engine* was that, for a long time, it was too expensive for anyone but large companies and wealthy individuals. However, in 2015, Epic Games changed the licensing for the engine to make it completely free to download and use as well as open source. There was a catch, though: developers had to pay a 5% royalty fee on all sales over $3000.

Recently *Epic Games* raised this amount to $1 million dollars. As of today, if a developer makes over $1 million dollars on a project created with the Unreal Engine, they must pay *Epic Games* a 5% royalty fee on any inconme earned beyond that first million.

Educational organizations is granted the engine free of charge.

1.2 Unreal Editor vs Unreal Engine

As this document's title suggests, it discusses development of games using the *Unreal Engine*. What this actually means is: The developed games are **using** the *Unreal Engine*, i.e. its runtime and frameworks.

During development, the developer uses a development environment called the ***Unreal Editor***, which in turn provides the runtime as mentioned above.

1.3 Demo Game

This document contains the theoretical foundations of game development with the *Unreal Engine*. That is, it contains explanations in the various topics (as layed out in the table of contents) along with simple examples presented

where deemed appropriate.

However, accompanying this course an example game will be developed, which is discussed in a separate document.

Figure 1.1: Sample Game Screenshot

This game will be a simple first person platform game in which the character can walk, jump, look around, use items and so on, using the keyboard but also a game controller. It will he shown how to create an item collection system and implement game logic.

Topics like collisions, damage, "death" and how to interact with the game world are being discussed as well as how to create a *Heads-Up Display (HUD)* and creating and using menus.

1.4 Unreal Engine Installation

Firstly, the engine can be downloaded from unrealengine.com.

In order to download and install the engine, the *Epic Games Launcher* is required, which can be downloaded from Epic Games. Hint: download the version for your platform; i.e. either *Microsoft Windows* or *macOS*. Also zipped packages for the *Linux* operating system are available, albeit installation is not as straight forward.

An *Epic Games* account is required, which can be created using the *Sign Up* link at the bottom of the web page.
Notice: This course assumes *Unreal Engine version 5.5* is installed.

1.5 Unreal Projects

1.5.1 Project Management

Within the context of the *Unreal Engine* a project is simply the unit that stores all the information for an individual game.

Handling of projects is carried out by the *Unreal Games Launcher*; i.e. new projects can be created using the launcher, existing projects can be loaded using the launcher, existing projects, like projects downloaded from the net, can be opened using the launcher.

The launcher provides a number of templates, reflecting different game types.

1.5.2 Asset Management

In addition to handling games, the launcher also provides a section giving access to game asset libraries; at the time of writing, the new market place Fab.com Market Place.

1.6 Levels

In the context of the *Unreal Engine*, *Level* can be defined as a collection of objects and their properties that together define an area of gameplay. Informally, levels are just the virtual location a section of game takes place in.

For a person used to play computer games, the concept is trivial, for non-gamers this idea is not necessarily that clear.

For the developer this has the concrete consequence that during development the *Unreal Editor* loads each level in turn or, in other words, developers work on one level at a time.

That being said, the *Unreal Engine* allows for insanely large environments, which are basically limited by hardware only.

1.7 Actors

Within the context of the *Unreal Engine*, an /**menuActor** is simply any object that can be added to a Level.
This object includes any object thinkable: Game assets, simple static meshes, etc. Actors can be visible or invisible, players sometimes can interact with an Actor, or not. So, again, an Actor is basically any entity that can be placed into, and moved around in, a Level within the Unreal Editor.

1.8 Meshes

Important Actors are so-called Mesh Actors. In this context, "mesh" is a 3D modeling term, and simply refers to a 3D object consisting of a mesh of vortexes. So when a game is played, most of the objects in the game will be a meshes of sorts.

Static meshes refer to meshes with no moving parts. The Unreal Engine's Starter Content, for examples, provide some Static Meshes in the form of furniture and some basic architectural objects.

Typicall, the vast majority of meshes used in games are created in external 3D modeling applications, such as Maya, 3D Studio Max, Blender, etc and subsequently imported into the Unreal Editor.

1.9 Geometry Brushes

A Geometry Brush, often times simply called Brush, is simply an Actor used to represent 3D space; in Unreal Engine's Place Actor's panel there are box brushes, cone, cylinder, stairs, and so on. Unreal Editor's Place Actors Panel refers to them as Geometries.

Brushes very similar to a mesh, but there are a few key differences between meshes and brushes. Brushes are only used for basic geometric shapes while meshes can be crafted into objects with a high level of detail.

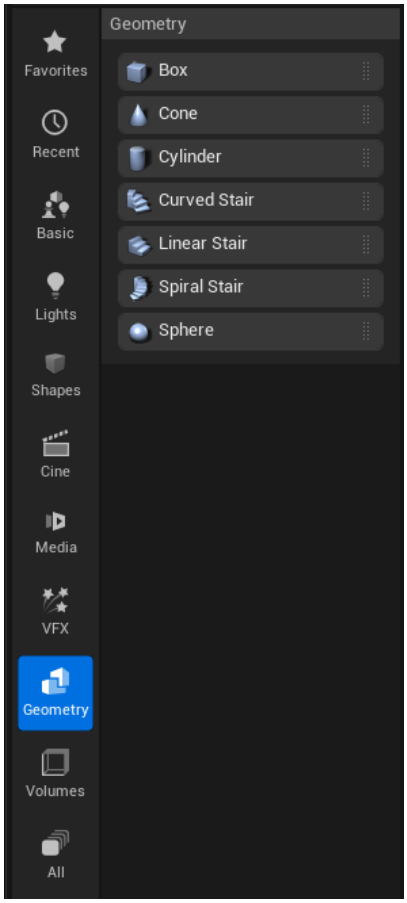

Figure 1.2: Geometry Brushes in Place Actor's Panel

Brushes are useful for quick level design but are less memory-efficient than meshes. Therefore, brushes are generally used to prototype Levels early on, and are then replaced with better-looking and better-performing meshes for the final project.

Notice, that both Meshes as well as Geometry Brushes can be applied a material.

1.10 Light Actors

In *Unreal Engine*, Light Actors represent visible light in the real world.

Thanks to a lot of complex, mathematical algorithms that the Unreal Engine uses, those will behave much like light does in the real world. It will make objects that it hits more visible, depending on the intensity of the light and the Material of the object.

It will reflect off the surface of objects and light up other objects indirectly. It will cast shadows if a visible, opaque object is in its path.

Light Actors are used to represent only the light itself, and not any of the objects from which the light emanates from. As an example, if a working flashlight shall be used in the game, a Light Actor could be combined with a Static Mesh Actor looking like a flashlight. The light actor would be placed at one end of the flashlight in a way so it looks like the light is emanating from the mesh.

1.11 Hardware Requirements

The *Unreal Engine* not only is a huge framework in itself. During development, not only the *Unreal Editor* is loaded into memory, but also the game, i.e. all its assets, content, game logic and so forth.

This also entails hardware acceleration of visuals, i.e. the engine relies heavily on graphics acceleration using *Graphics Processing Units (GPUs)*.

As a consequence, working with the *Unreal Editor* is not recommended on computers with low performance computers.

The official hardware recommendations issued by *Epic Games* notwithstanding, the following minimum requirements should be fulfilled:

- Minimum RAM: 16 GB
- GPU: dedicated, 4 GB GDDR. Computers and laptops without dedicated graphics acceleration are not recommended
- SSD: 100 GB free. The use of fast hard drives (SSD) is recommended. The installation requires around 60 GB of disc space. Notice: this reflects the bare minimum only. Unreal projects tend to grow rather large, since they contain not only the game logic, but also the game assets, which can grow very large.

Anecdotically, the author did some testing using the game described in the accompanying document. Here are the results:

- Desktop PC: High-End CPU, 64 GB RAM, 24 GB Nvidia 4090 $>$ 120 FPS. Basically, not limited by hardware.
- Lenovo P52 Workstation notebook, Intel i7 8700, RAM 64 GB, Nvidia Quadro 4000 GPU, 6GB GDDR: $>$ 60 FPS.
- Macbook Air M2, 24 GB unified RAM: $<$ 15 FPS. Game is unplayable

The Level Editor)

2.1 Editor Overview

The level editor is the part of the *Unreal Editor* in which you place the object, position it, rotate it, alter its size or set its various other attributes.

Using the editor, it is possible to visualize the level in different ways, like:

- With lighting
- Without lighting
- Wireframe mode
- etc.

The main window of the Unreal Editor is the Level Editor itself. All of the other sub-editors will open in their own separate windows or tabs (The respective setting can be found in Edit > Editor Preferences > General > Asset Editor Open Location).

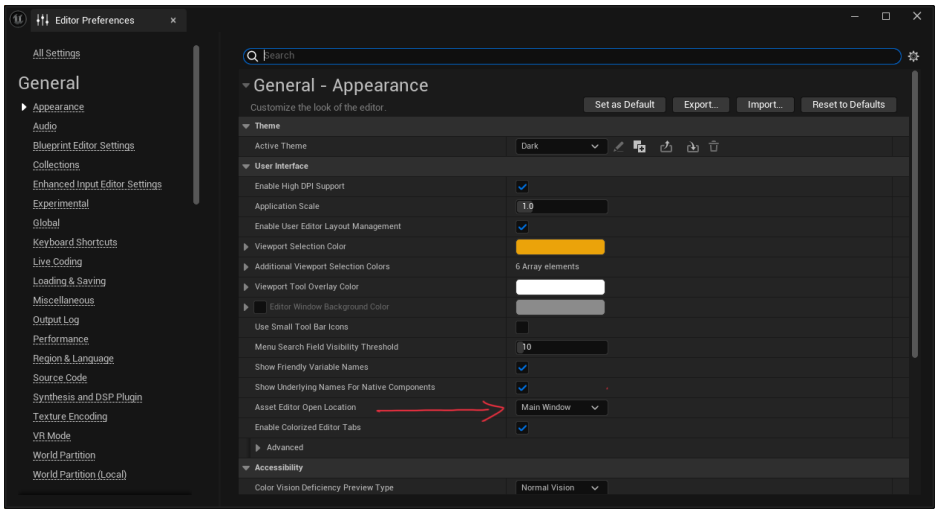

Figure 2.1: Open Asset Editor Location

For example, if a Material is double-clicked, it will open the Material Editor in a separate window/tab.

2.1.1 Editor Panels

The level editor consists of a number of panels:

Firstly, the large area in the middle is denoted as *Viewport*.

The thin strip above that is called “Toolbar”.

At the bottom of the screen, hidden within the “Content Drawer” is the “Content Browser”.

The rest of this strip is the “Bottom Toolbar.”

On the right side of the screen is the “Outliner” at the top, and below that, the “Details” panel.
These panels can be moved and resized and represent just the default layout of the current release and could change in future releases.

Of course, the students are free to alter the default settings as deemed; however, it is recommended to use the default settings unless stated otherwise since this is what this document assumes.

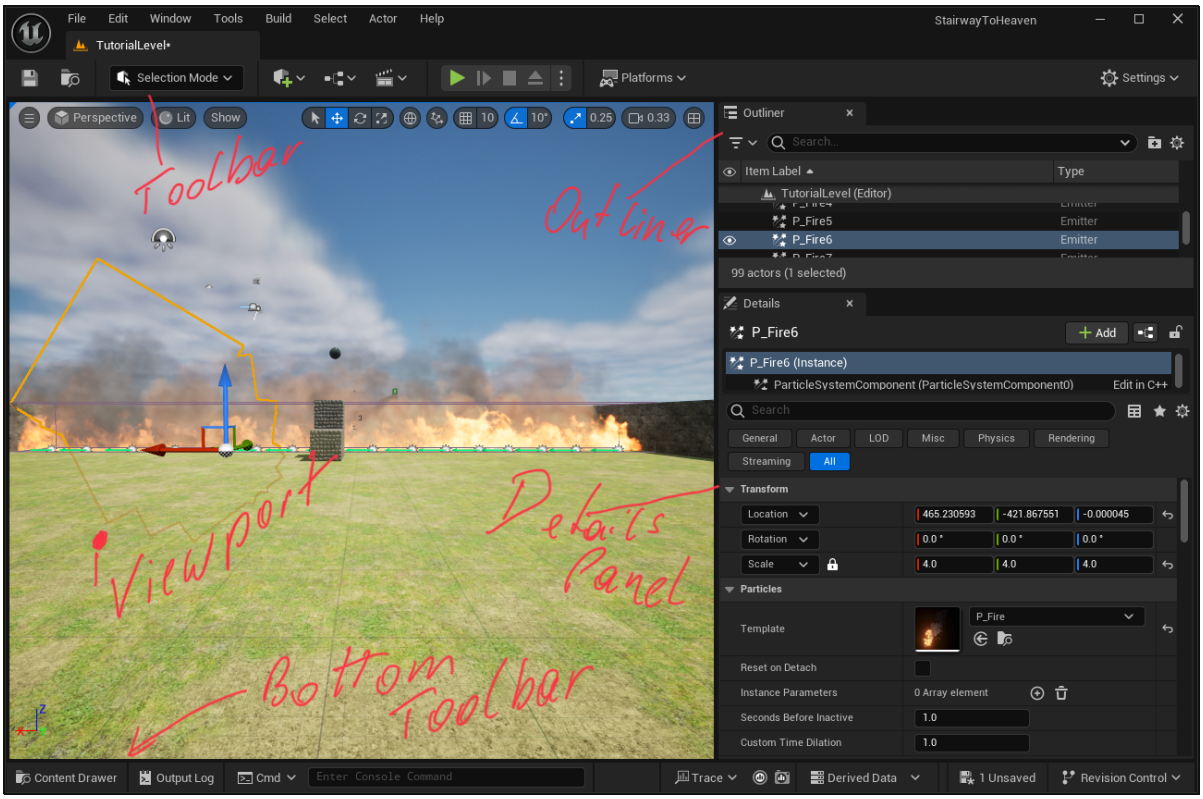

Figure 2.2:

Find an overview on what these panels are for below:

- **Viewport:** Provides a visual representation of the game. The developer sees the game world in the same fashion as the players, i.e. the environment created, characters and objects the player interacts with. In addition, objects used to provide the game mechanics, which are invisible to the player, can be seen and interacted with. These entail objects like cameras, event triggers or barriers invisible to the player.

  Also, objects can be manipulated using the viewport.

- **Toolbar:** The Toolbar is a strip of buttons meant to provide quick access to common and/or important functions, such as saving, changing the editor mode, or playing the game.

- **Content Drawer**, which can be found in the Bottom Toolbar: Used for storing and organizing content that can be added to the game.
  This includes, but is not limited to, content such as meshes, materials, music, sound effects, visual effects, and so forth.

- **Bottom Toolbar:** The Bottom Toolbar is a strip of buttons that give you access to information about the Editor and allow you to quickly change Editor settings. It includes:
  - **Content Drawer** (also called Content Browser) can also be used to import assets - like 3D objects created using third party tools like *Blender* - into the editor. Naturally, also assets downloaded or purchased from the internet can be imported into the editor using this.
  - **Output log:** Output Log will log everything that's going on in the Editor and will display warnings and errors.
  - **Command Editor:** This is a command line interface where used to issue commands to quickly change Editor behavior.
  - **Derived Data:** lets developers change setting and statistics about the editor.
  - **Derived Data Panel:** lets developers change setting and statistics about the editor.
  - **Source Control Panel:** In case the Project is connected to some kind of source control, such as GitHub or any other SCM tool, the Source Control menu will display the Project's status and allow you to take various actions relating to source control.

- **Outliner** is used to list and group the objects in your level in a way that makes them easy to find when you want to select and edit them.

- **Details Panel** Allows for developers to view and edit all project details

As can be seen, the *Unreal Editor* provides lots of controls over how the interface looks.

What's more, each individual panel can be resized or relocated. To accomplish this, developers simply need to click in an empty area of the edges between adjacent panels.

As already mentioned, it is recommended to use the standard layouts for this course.

To close a panel, click on the X on the right side of the panel's tab. Conversely, in order to open a panel, go to the menu bar, and under Window, select the panel you wish to open.

Panels that are on the side of the screen can an be collapsed into the sidebar by right-clicking on the panel and select "Dock to Sidebar."

To permanently dock them again, it can be right-clicked to select Undock from Sidebar.

Finally, panel tabs an be hidden by right-clicking and choosing Hide tabs.

These custom layouts can be saved and loaded using (Load/Save Layout)

2.2 Select Editing Mode

The Level Editor has eight different editing modes.

To choose the desired mode, it simply can be selected using the dropdown in the Toolbar.
Alternatively, the mode can be selected by holding the Shift key and pressing the corresponding number key (1-8), depending on the desired mode.

Pressing Shift and the 4 key at the same time will spawn the fourth editing mode, which is Mesh Paint Editing Mode.Shift + 5 is going to spawn the fifth mode and so forth.

However, since it is by far the most used mode, in this document the Select Editing Mode only will be discussed. This is the first mode, therefor it can be opened using Shift + 1.

The Select Editing Mode is the primary mode to place and manipulate actors in a level.

2.2.1 Adding Actors Using the Create Menu

One way of adding actors into a level is by clicking the Create button (the rectangular shape containing the +)

As an example, adding a cube to a scene is as simple as clicking Create > Shapes > Cube. Alternatively a shape can simple be dragged from the Create menu into the level to place is.

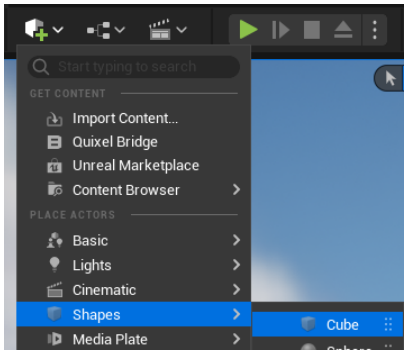

Figure 2.3: Create Menu

2.2.2 Adding Actors Using the Content Browser

Another way to add Actors to a Level is through the Content Browser. However, notice that there are a few key differences between it and the Create menu.

The Create menu is used for simple, common, generic actors, while the actors in the content browser tend to be more complex.

Furthermore, the list of Actors in the Create menu remains static.

2.2.3 Deleting Actors

Deleting an actor is a simple as selecting the actor followed by clicking the Delete-key.

Alternatively, right-clicking an actor and subsequently selecting Delete also does work.

2.2.4 Docking the Place Actors Panel

At the very bottom of the Create menu is a button that can be pressed to dock the menu as a panel called the Place Actors Panel. This is useful particularly in the early stages of level development, since actors can be placed right away, i.e. without clicking the Create button first.

This works just like the menu, except that it works using click and drag only.

2.2.5 Importing Actors (from) External

Naturally, not all assets, including actors, required during development are already in the projects; after all, games rarely use *Unreal Engine* provided assets only.

Hence, actors (as well as other assets) created outside of the *Unreal Editor* can be imported into the editor using the Content Browser by simply dragging and dropping.

2.2.6 Categories of Actors

The Place Actors Panel is divided into categories of different groupings of actors

Figure 2.4: Categories of the Place Actors Panel

For starters, the "Basic" category – the "Basic" category simply contains the most commonly used actors.

To the left of the "Basic" category is the "Recently Placed" category. This will be a list of the user recently placed into a Level. This is useful when working with a small set of the same types of actors for a while. This way, the developer could just keep the Recently Placed category open and drag and drop everything from there.

To the right of the "Basic" category is the "Lights" category. As mentioned, a light in *Unreal Engine* is an Actor that is meant to represent the light projecting from some source.

Next is "Shapes" which contains Static Meshes in the form of some basic geometric shapes.

Next to the "Shapes" category is the "Cinematic" category. his has actors used for creating Cinematics, which is basically a rendered 3D video. In the context of gaming, these would be used, for examples, for game cutscenes.

Next up is "Visual Effects", which, as its name suggests, contains actors that add a variety of effects to your level. More on this later in the document.

The Geometry category contains the Geometry Brushes that were briefly introduced earlier.

The "Volumes" category is used to define gameplay volumes. A Volume is a 3D area of space that is invisible to the player and serves a specific purpose depending on its type. For example, a Blocking Volume will prevent actors from being able to enter that volume, a Pain Causing Volume will cause damage to an actor who enters that volume, and so on.

And lastly is the "All Classes" category, which contains all the Actors from the categories mentioned above, plus some additional actors not found in any of the other categories, either because they are less common or just didn't fit nicely into one of the other groups.

And so obviously, this list is somewhat long, and so this makes this search bar up here useful. So by typing text in this search bar, the user can quickly narrow down the results to what she is looking for.

2.3 The Viewport

This section will discuss how to navigate the viewport, as there is a number of options available; this document discusses three different ways of navigating.

In case the user intends to follow along, it shows its example using an new project:

- Create a new project.
- Select the empty project template, make sure that Target Platform is set to Desktop, Quality Preset is set to Maximum, and finally tick the Starter Content box.

- Give the new project a meaningful name.
- Finally, click Create.
- Navigate to Content Drawer » Starter Content » Maps folder.
- Open the Minimal_Default level (e.g. by double clicking it)

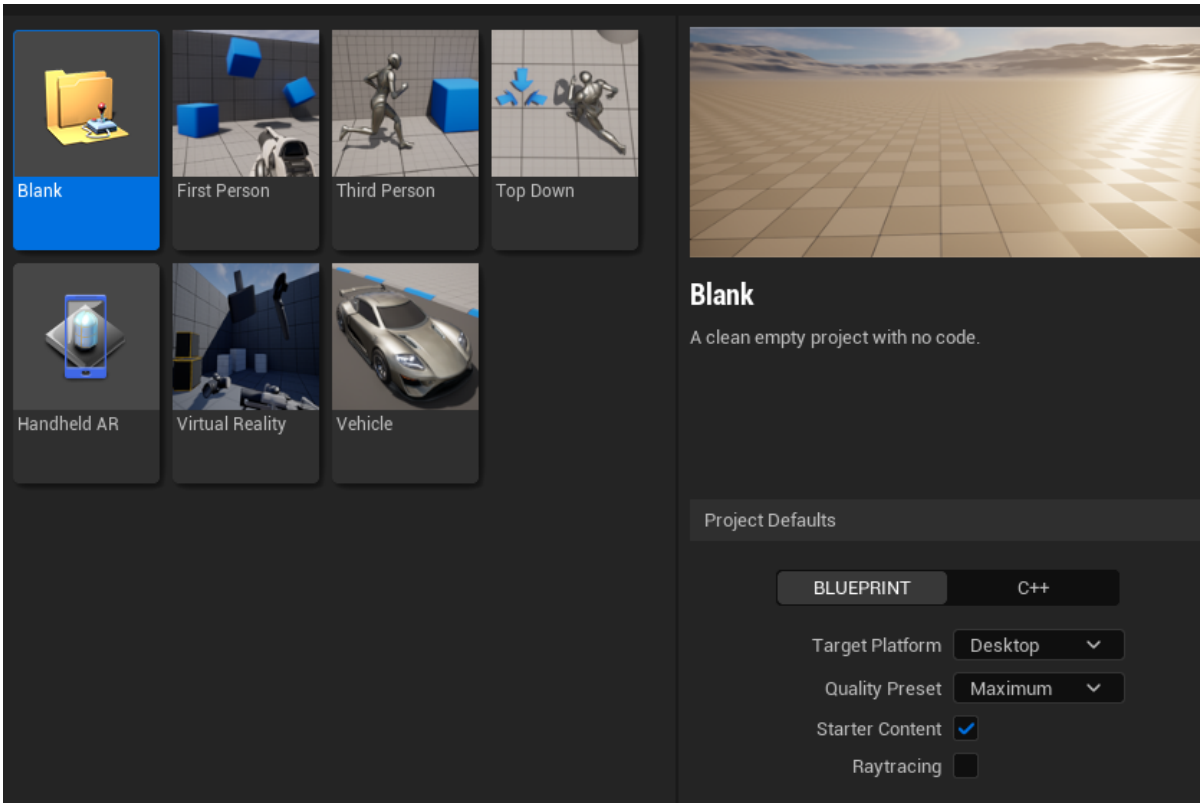

Figure 2.5: New Blank Project

This Level starts out with some of the meshes from the Props folder of the Starter Content already placed in the scene.

Pressing F11 will switch to the full screen viewport mode.

2.3.1 Mouse Navigation

Left Mouse Button (LMB)

This mode uses the LMB; holding it down and moving the mouse around allows the camera to move forward or backwards as well as to rotate left and right.

Notice, that there is no way of moving up and down using this movement; it only allows traversing the x and y axes. It is also impossible to move right or left directly; it allows for rotating left and right only.

Middle Mouse Button (MMB)

For moving left and right, hold the LMB and the RMB at the same time and drag the mouse (this is equivalent to holding the MMB, depending on what mouse you've got available.). This allows to move directly left as well as right and up and down.

Right Mouse Button (RMB)

This won't move the camera along any axes, it only allows for camera rotation.

2.3.2 WASD Navigation

WASD navigation - i.e. navigating using the W, A, S, D keys for the movements forward, backwards, left and right, respectively. This mimics the way players use to navigate in first-person view games.

WASD works while holding down the RMB. This mode allows to rotate in arbitrary directions as well as moving as described using the WASD keys.

In addition, the camera can me moved up and down using the Q and E keys, respectively, while Z and C can be used to zoom the camera in and out, respectively. Notice, that zooming in and out is just temporarily; if the user lets go the RMB, the zoom level is going to go backwards to where it has been.

2.3.3 Focusing

Using the F key provides a powerful feature particularly useful with the third mode of navigation (discussed below). F focuses on the *currently selected* object. For example, selecting on of the chairs and hitting the F key is going to focus the viewport on this chair.

Notice, that this also works when the object is selected in the outliner. Double-clicking an object in the outliner has the same effect as using f, i.e. it focuses on the selected object. In smaller levels this feature does not make a big impression; however, in large levels this is very useful, since the viewport can be focused without the need for navigating a large level.

2.3.4 Maya Navigation

Maya Navigation, phrased after the famous 3D modeling app, is performed holding down the Alt key.

LMB Maya Navigation

Holding down Alt + LMB and subsequently dragging the mouse is going to *tumble* or *orbit* the camera around a point of interest. This point is not necessarily the selected object. However, if an object is focused on (F), the viewport focuses on this particular object. Now, using Alt + LMB will revolve the viewport around the object; a very useful feature.

RMB Maya Navigation

Holding down Alt + RMB while dragging the mouse will *zoom* the camera to or away a point of interest. Again, focusing on an object will lead to moving to or away from this selected object.

MMB Maya Navigation

Finally, Alt + MMB will cause the viewport to *pan* up, down, left and right.

Maya navigation, therefore, is particularly helpful because using it enables easy and free navigation around a level using *Maya Navigation* only.

2.3.5 Camera Speed

At times, the viewport camera speed seem either way to fast or much too slow.

The camera speed can therefore be adjusted using the camera speed slider, which can be found on the upper right corner of the viewport.

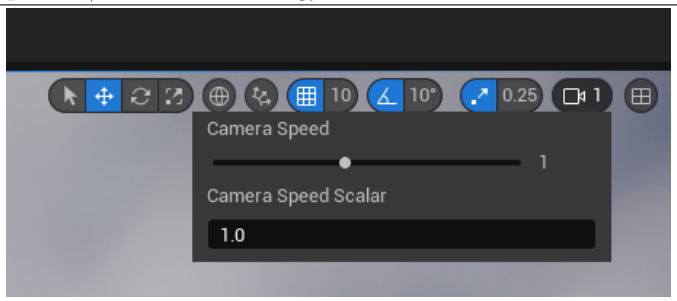

Figure 2.6: Adjusting Camera Speed

2.3.6 Widgets

In the *Unreal Editor*, actors can be manipulated using translation, rotation and scaling widgets.

These widgets can be found on the upper right section of the viewport; the widgets are also connected with keyboard commands.

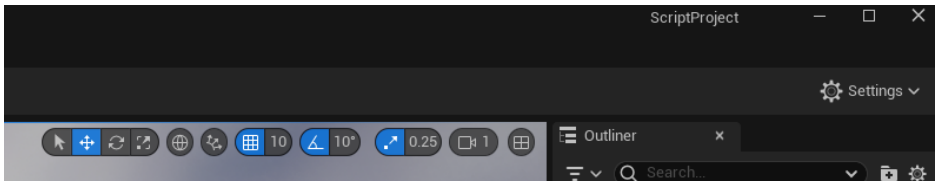

Figure 2.7: Translate, Rotate and Scaling Widgets

As can be seen in 2.7, some widgets are displayed in Blue, indicating those are active. To activate a different widget, simply click on the widget desired to activate it. Alternatively, the respective widgets can be activated using the Q, W, E and R keys. While these keys seemingly appear in arbitrary order, they are in a row on the keyboard and activate the icons in the same order as they appear in the viewport.

The widget all the way on the left is selected, none of the widgets will be used.

To summarize, Q disables the widgets, W will activate the Translation Widget, E will activate the Rotation Widget, and R will activate the Scaling Widget.
Hint: Hovering over over an icon and it will tell what the widget does and how to activate it.

Translation

Selecting a chair and subsequently pressing W activates the Translation Widget. The three different colored arrows coming out of the chair are aligned with the X, Y, and Z axes of the Level. In order to move a chair in just one direction, left-click on one of the arrows, and use the mouse to move the chair back and forth in that direction.

For example, clicking red arrow, no matter what direction the mouse is dragged, the chair will only move this direction, along the X-axis. Similarly, if the chair is to be moved along the Y-axis, select the green arrow instead or the blue arrow for movement along the Z-axis.

Notice the connectors between every pair of axes. These are used if movement along **two** of the axes is required, but not on the third axis. Using mentioned connectors allows just that.

Holding down the Shift key while moving an actor will move the camera in parallel to the selected actor. This is useful is the actor is to be moved to a far of location on the level.

The white sphere in the center allows for free movement along all three axes.

Holding down the Shift key during translation will create a copy of an actor.

Additinoal Hints:

In order to select multiple actors, the Ctrl key can be used.

Ctrl + D will create a copy of an actor.

Rotation

This is activated using E. Using the Rotation Widget enables rotating an actor around any of the three axes in space.

Similar to translation, holding down the Shift key during rotation will create a copy of an actor.

Scaling

This is activated using R. Using the Scaling Widget enables scaling of an actor around any of the three axes in space. Clicking on a connector rather than one of the axes will scale the object in the selected axes only.

Clicking the white cuboid in the center is going to scale the object uniformly in all three dimensions.

Similar to translation, holding down the Shift key during scaling will create a copy of an actor.

Changing the Pivot Point

The pivot is the point in space around which ab actor rotates. It is located at the center of the Transformation Widget, which, for this chair actor, is located

at the bottom-center of the actor by default. It can be changed by using MMB. Clicking and holding the MMB the pivot can be dragged to a new location. Now rotating the actor, it will rotate around the new pivot point. Notice, though, that this moves the pivot point temporarily only. I.e., deselecting the actor and selecting it again, the pivot point is back at its default location.

2.3.7 World Space vs Local Space

Let us assume a chair is rotated a bit here from its default rotation using the Translation Widget. Thereafter, the chair is translated, say, up the Z-axis, a bit. Then, if the icon to the left of the Translation, Rotation and Scaling Widget show an icon of the Earth, it means that these axes here are oriented to "world space", i.e. no matter which way the chair is rotated, these arrows will always point in the same direction, aligned to the level coordinates, and thus the chair will always move in the same direction relative to the world.

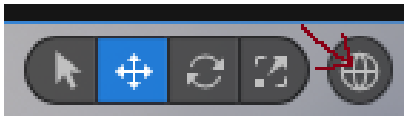

Figure 2.8: World Space Icon

However, this button or the shortcut Ctrl + ‘ is clicked, this icon will change to a cube which means that the axes are now oriented to "local space", in this case the local space of the chair. Notice: On keyboards using a german layout this does not work as intend, since the ‘ requires clicking the ‘ key, followed by a white space character. Use the icon to switch between world and local space instead.

This setting does not apply, however, to the Scaling Widget. The Scaling Widget will always be in local space, and it won't allow you to toggle when the widget is selected.

2.3.8 Snapping

End Snapping

Snapping is a way to perfectly align actors with one another within a level. There are several ways of doing this. The first method involves using End. An actor selected, pressing End, will snap that actor directly onto the nearest surface. This is immensely useful for aligning objects, especially for getting actors to sit directly on the floor or some surface.

Surface Snapping

Another way of using snapping is called Surface Snapping. Clicking the grid shaped icon on the top viewport will display a small pop-up enabling Surface Snapping. With Surface Snapping on, using the Translation Widget it will snap the selected actor to the surface of a nearby actor.

Notice, that this only works when moving the object in 3 dimensions. It does not work when moving actors around in just 1 or 2 dimensions at a time. In other words, the actor must be dragged using the white sphere in order for Surface Snapping to work.

There are several setting that can be adjusted regarding Surface Snapping:

The first setting is called Rotate to Surface Normal and it is set to On by default.
Let us assume that a chair was rotated a bit in a way that its bottom surface was no longer in alignment with the surface of the floor. If Surface Snapping is on, but Rotate to Surface Normal is off, the chair will snap to the floor, but it is not going to rotate to align with the surface. Conversely, with Rotate to Surface Normal on the chair will also be rotated so that its bottom surface is perfectly aligned with the surface of the floor.

The Surface Offset setting tells the editor how far away the surfaces of the two actors should be when they snap together. This setting can be used to let objects seemingly hover above the ground.

Vertex Snapping

This is used if there is a requirement to snap to actors together with precision.
To use Vertex Snapping in conjunction with the Translation Widget, pressing and holding V will cause several blue dots once the object are close together. Each of these dots is a vertex an actor can be snapped to.

There is a catch, though. Usually, pivot points (as well as the center points of actors) are in the middle of the object. Therefore, if two objects are to be snapped together at some surface, those are not necessarily helpful. In this scenario, temporarily moving the pivot points using the MMB may do the trick.

Now, moving a pivot point freely may not be precise in the first place. Luckily, this situation can be resolved. While still holding the MMB, Alt - V will snap the pivot to one of the actor's vertices. Now, with the pivot point set, V is going to snap the two actors together.

Grid Snapping

Grid Snapping is useful for aligning objects across distances. Grid Snapping is activated using the grid symbol in the upper part of the viewport, to the right of the translation, rotation an scaling widgets.

With Grid Snapping on, moving an actor is going to move the actor in increments as given by the grid.

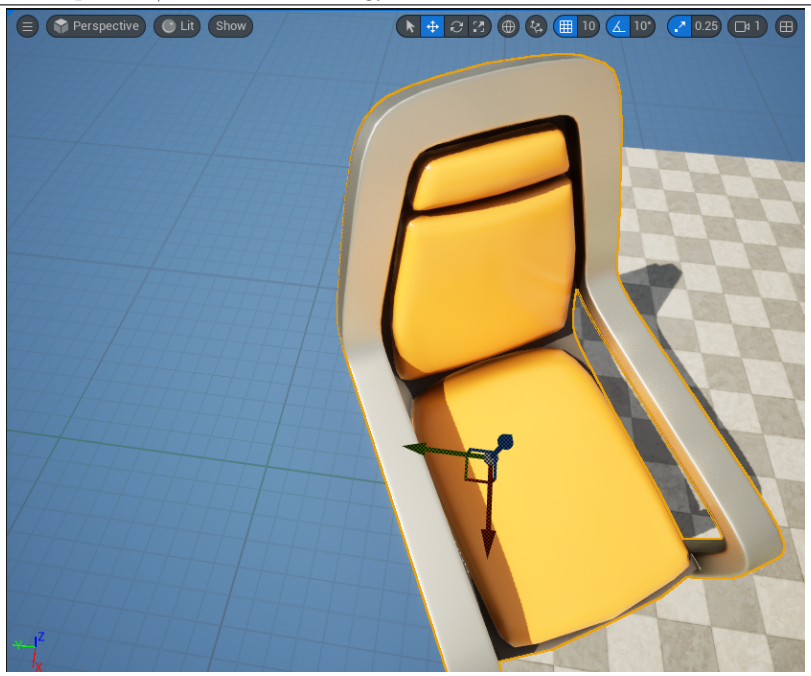

Figure 2.9: Grid Snapping Example

As depicted in 2.9, the now ”snaps” to each line of the grid, making it impossible to move the chair any smaller amount than that given.

The grid size can be altered by clicking the number symbol to the right of the Grid Snapping icon.

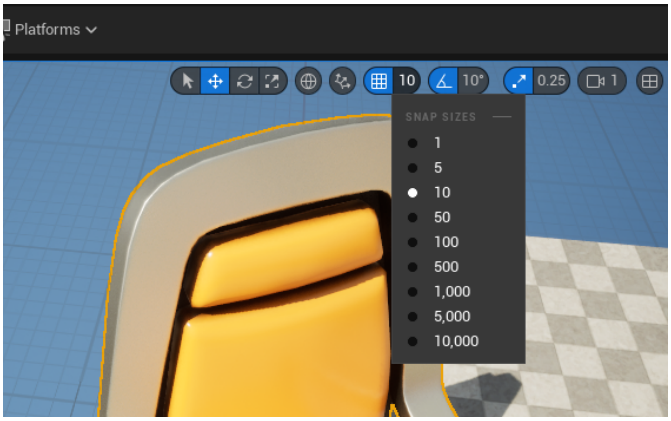

Figure 2.10: Adjusting Grid Snapping

Notice, that in *Unreal Editor* 1 grid unit is supposed to represent once *Centimeter* in the real world.

Snapping affects rotation as well, rotation increments will also apply.

Notice, that in *Unreal Engine* rotation is measured in degrees.

Similarly, snapping also works with *scaling*.

2.3.9 Level Viewing

There are different ways of viewing a level.

As mentioned earlier, F11 toggles viewport full screen viewing. This mode is also known as *Immersive Mode*.

In addition, there is a number of different views available in *Unreal Editor*. These can be changed using the respective icon in the upper left section of the viewport.

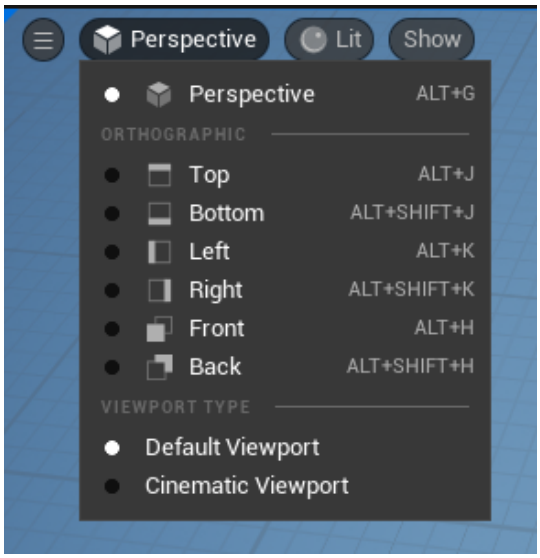

Figure 2.11: View Ports

2.11 shows that currently, the Perspective View, which is the default view, is selected. There are also views available, viewing the level from Top, Bottom, Left, Right, Front and Back.

Some of these view modes are intended to view subtle details regarding light. These are Lit, Unlit and Wireframe.

Lit is the default view mode of the Level Editor and is for viewing the Level with full lighting and rendering, similar to what the player will see in-game.

Unlit is for viewing the Level without the lighting having any impact on what is visible. Essentially, this allows to see the base colors of all objects without the effects of light or shadows affecting the color. Unlit is also useful for working in levels or areas of levels where the lighting would otherwise be too dark to easily see the level.

Wireframe is a three-dimensional model in which only the lines and vertices of an object are visible. Wireframe Mode basically allows to see just edges and vertices the level is composed of. This might be helpful for aligning objects, visualizing the structure of actors and provides a architectural view of the level.

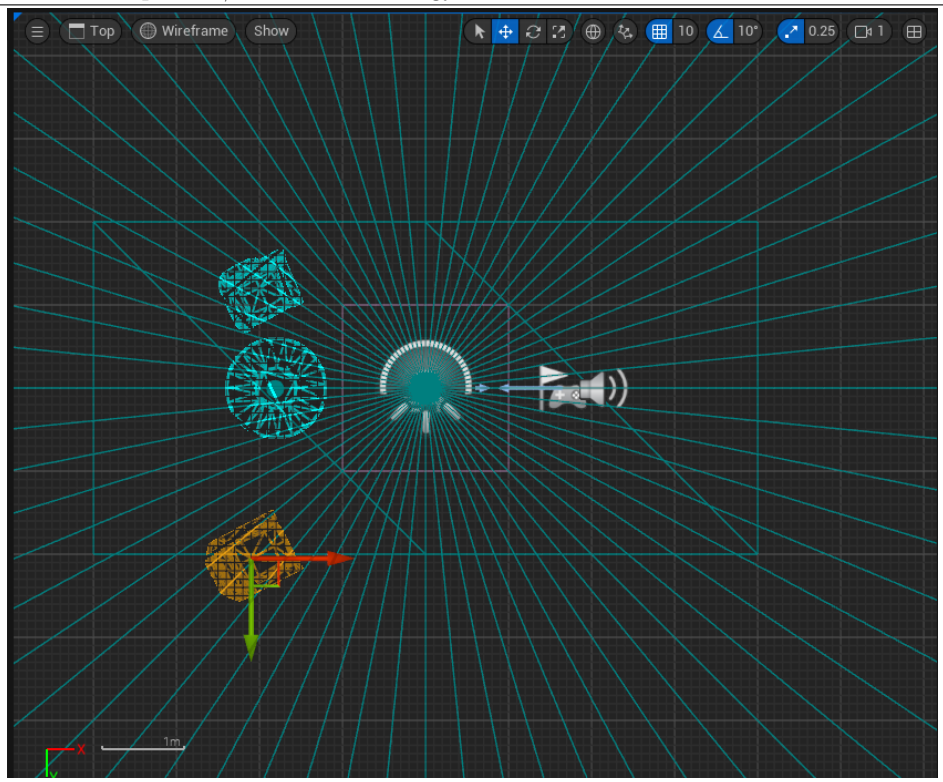

Figure 2.12: Wireframe Mode

To the left of the View Modes menu is a menu that allowing to changes between the default 3D perspective view of the level, to an orthographic view of the Level.

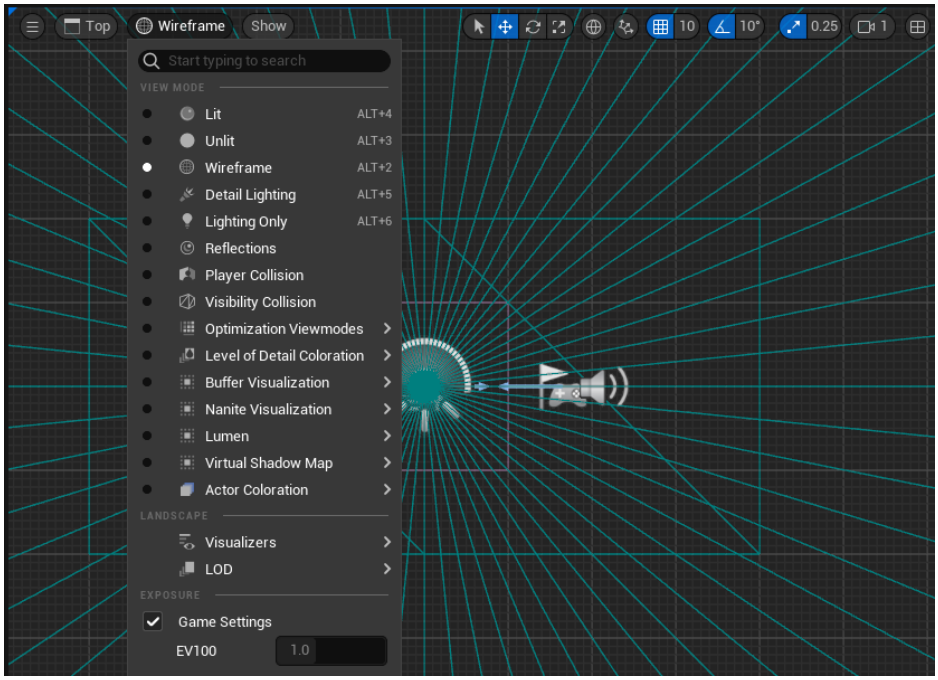

Figure 2.13: View Mode Settings

The several other views, which are all orthographic, share share two common traits:

The first is that they default to a wireframe view mode, although this can be modified.

But the main thing about orthographic views is that they are meant to be 2-dimensional views, rather than 3-dimensional. For example, in the Top view, the vision is directly parallel with the Z-axis, so it is as if looking directly down at a level, and only seeing the X and Y axes.

And in Bottom view we are looking directly up at the level. The Left and Right views are looking down the Y-axis and only seeing the X and Z axes.

Finally, the Front and Back views are looking down the X-axis and only seeing the Y and Z axes.

Orthographic views are useful for aligning the objects in your level in just 2 dimensions at a time. And when paired with the wireframe view mode, the idea is to get a 2D architectural view of a Level.

Show Flags

To the right of the View Modes icon lies the related Show Flags icon. Show Flags are, in essence, check boxes which tell the editor whether or not show various types of actors or effects in the viewport.

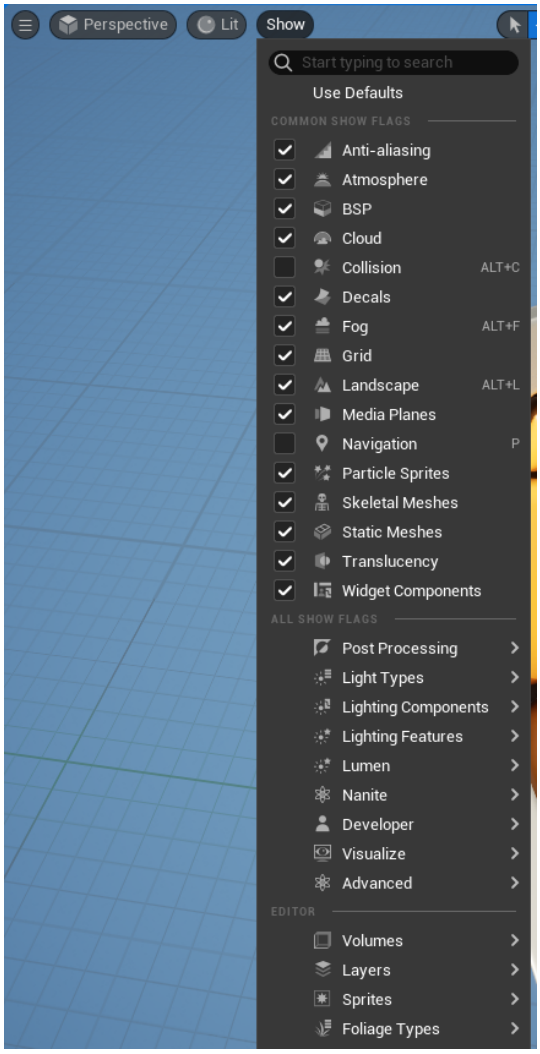

Figure 2.14: Show Flags

For example, unchecking the Static Meshes show flag, all of the static meshes in the level will disappear from the viewport. This doesn't delete the static meshes from the viewport, it only makes them invisible for the time being. It also doesn't make them invisible in the game itself, but only in the viewport.

Another feature similar to this is provided by the G key, which toggles the viewport in and out of game view. In game view, it will hide all Actors and icons that are invisible in-game. The PlayerStart actor, for example, will be hidden, along with the icons for Light Actors, etc.

Game View is meant to show the developer, in the Viewport, exactly what the player would see from that perspective in the game.

Viewport Options

Viewport Options can be found by clicking on this small button all the way in the upper left of the viewport.

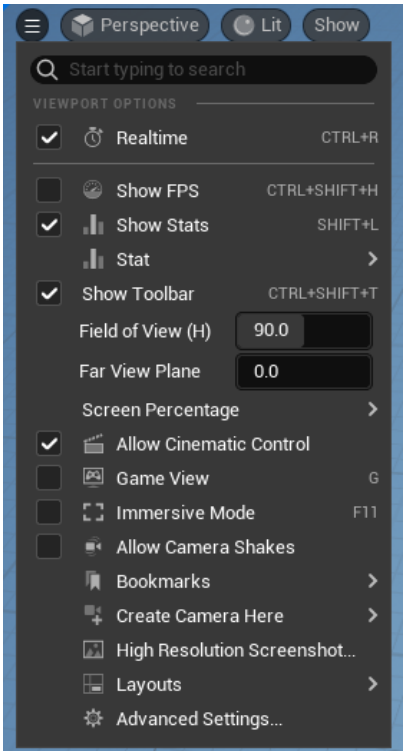

Figure 2.15: Viewport Options Menu

For brevity, only a number of options will be discussed here:

The first option on the menu is Realtime, which toggles realtime rendering of the viewport on and off. If this is enabled, effects and animations will not be active; i.e. only a static image will be rendered in the viewport.

Show FPS should be fairly self-explanatory.

The next option is Show Stats, which toggles the ability to display statistics in the viewport. With this on, if you go down to stat, you will find a large number of different stats available for display.

Bookmarks is used to bookmark viewport camera locations and rotations. It's easiest to just use the keyboard shortcuts but you can also use the menu if you wish.

To create a bookmark, press Ctrl + any of the numbers between 0-9 on the keyboard, and then to recall the bookmark, simply press that number. For example, to bookmark a particular camera position, press Ctrl +1. To return to that position, press 1.

Create Camera Here will simply create a Camera Actor at the location and rotation that the viewport camera is currently in.

Advanced Setting is just another way of getting to the viewport section of Editor Preferences, where many more advanced options for fine-tuning how the Viewport functions can be found.

2.3.10 Piloting Actors

Sometimes it is useful when placing actors, especially Camera and Light Actors, to see from the perspective of the actor itself. This makes it easier to have the actor point at an exact location.

For example, if a light actor shall be placed in a level with the intention of it being placed directly above a different actor, the spot light can be piloted. Right clicking on a light, for example, will display an option to the right of an airplane icon saying Pilot Spot Light. Alternatively, with the light selected pressing Ctrl + Shift + P will do the same.

Now the developer can see directly in the direction of the light so it be directed much easier.

2.4 The Content Drawer

This section is going to provide an overview of the Content Drawer, sometimes also referred to as Content Browser.

By default, the Content Drawer is collapsed. It can be found on the bottom left margin of the *Unreal Editor* window; it also can be opened by using Ctrl + Space Bar. It will collapse automatically once it goes out of focus.

To keep it docked, the Dock in Layout icon may be used; to close it, the X icon in the tab may be used

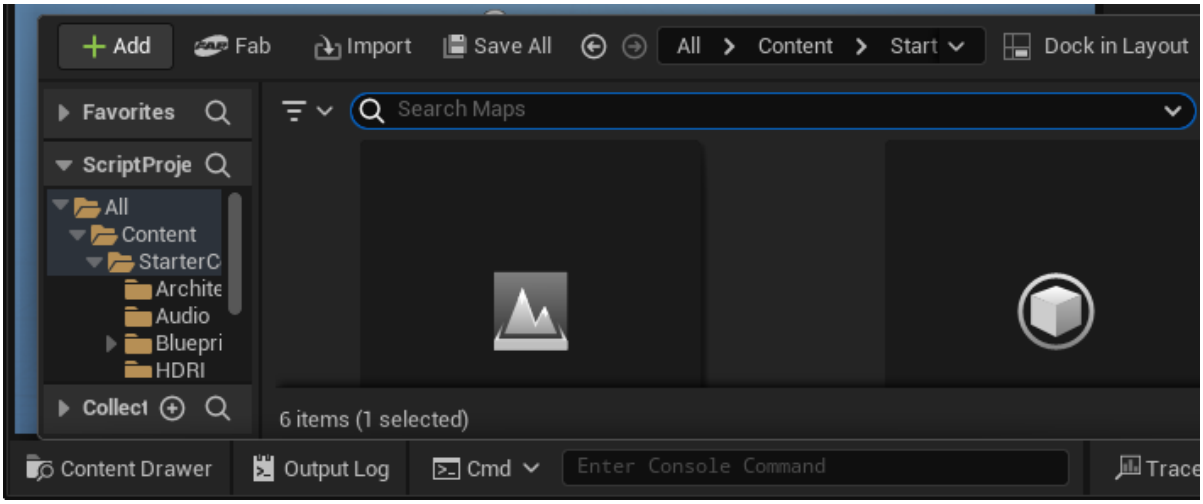

Figure 2.16: Content Drawer

The Content Drawer can be used to drag and drop content into the level, similar to the Create menu.

For example, to add a small bush actor into the level, one may open the Content Drawer, find All > Content > StarterContent > Props > SM_Bush and simply drag it into the level.

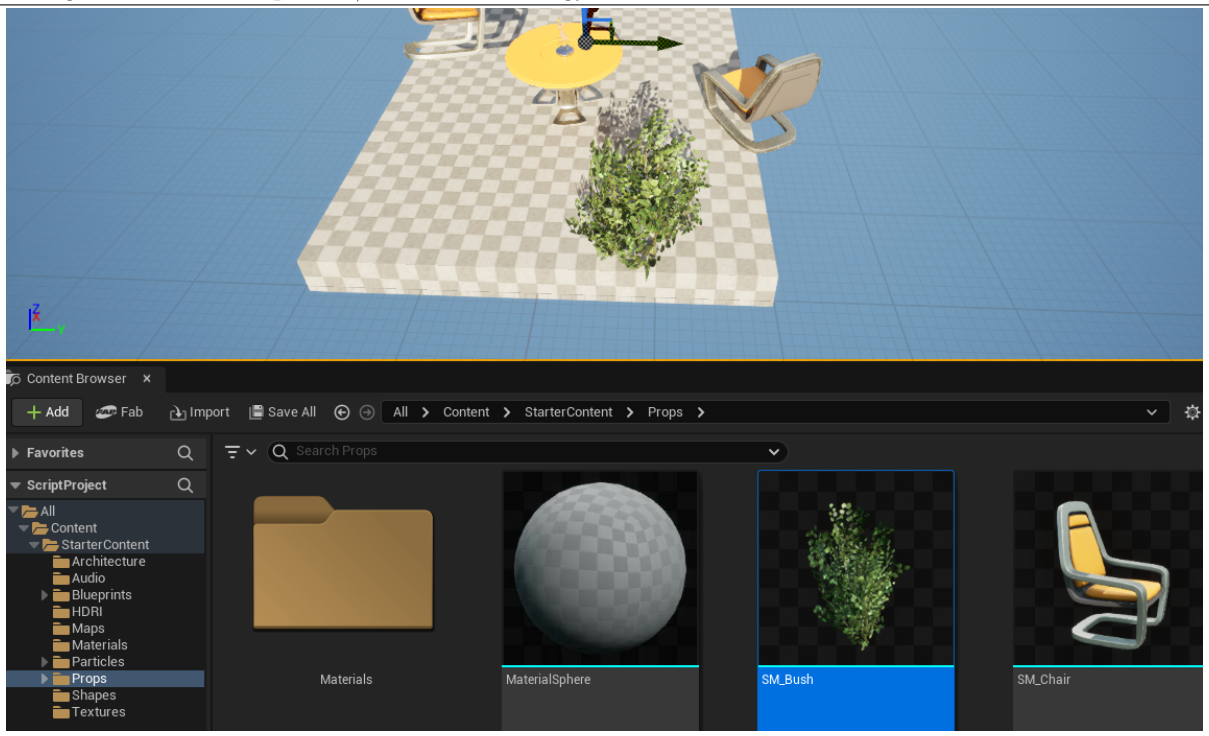

Figure 2.17: Content Drawer - SM_Bush Example

Looking at the sample's floor it is apparent that there is no material applied to it. A material can be applied, obviously, by dragging a material onto it, e.g. by selecting a material from the StarterContent » Material folder.

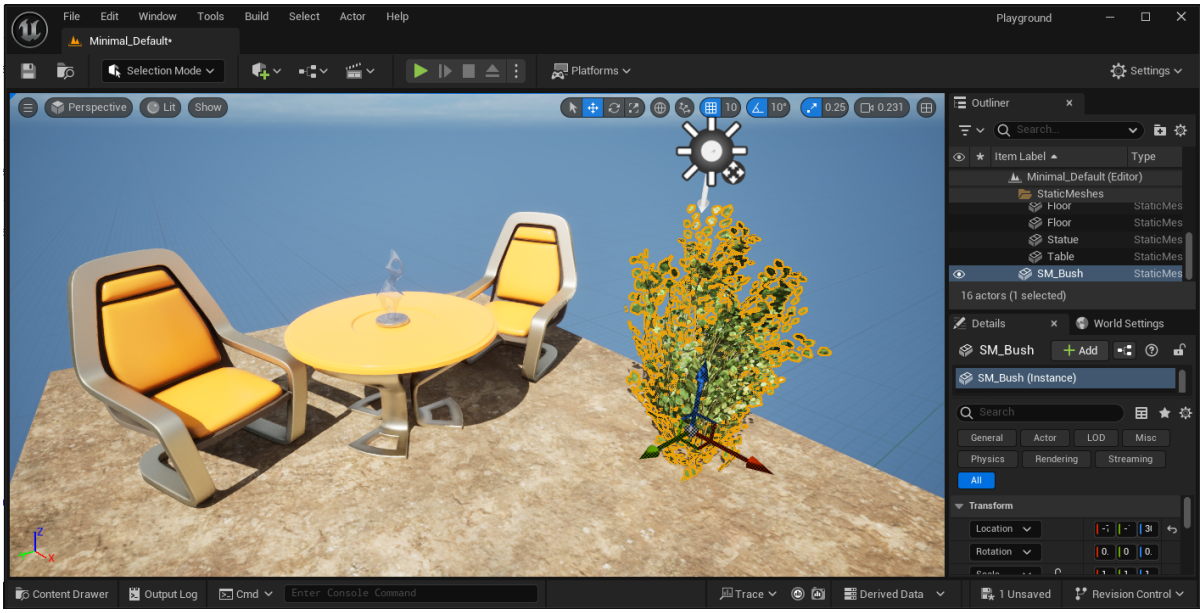

Figure 2.18: Material Applied to Floor

While the Create menu contains a list of generic, built-in actors, Content Drawer can be used to create actors, or to import in actors created outside of the *Unreal Editor*.

Notice, that the right part of the Content Drawer is referred to as Asset Window, since it contains a list of assets, while the left part - the one resembling the file system browser - is called Sources Panel.

In order to navigate large projects, a Search Field is included into the panel.

By default, the search option searches File Names; i.e. entering the term *"image"*, the search should find all files within the selected folder containing the word "image". It does not find assets that are of an image type.

To the right of the search bar sits a Filter icon. This allows for filtering of search results based on the type of content that is searched for.

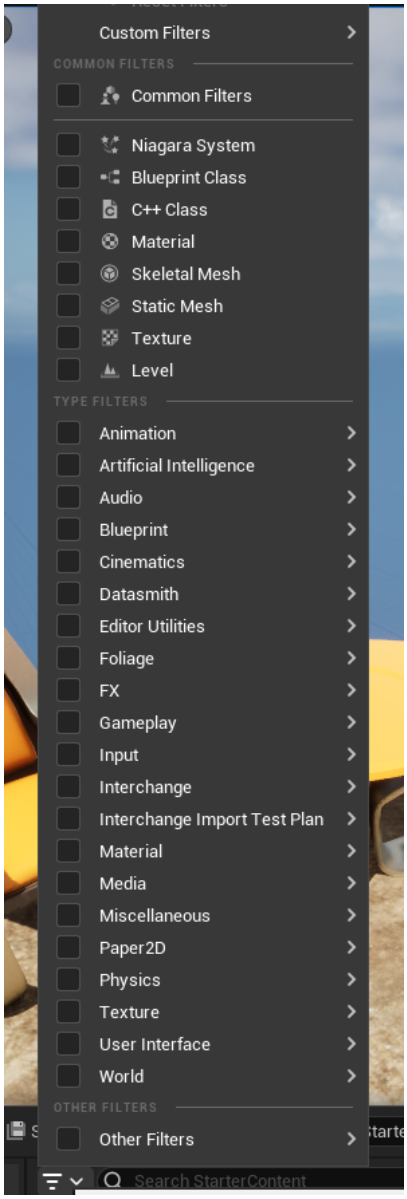

Figure 2.19: Search Filter

Notice there is a search field on top of the filters window, which can be used as well. Also note, that once a filter is set, an option Reset Filters appears, which can be used to remove all filters at once.

Search and filter functionality can be combined here. E.g. typing the word "*la*" will yield all filters containing the string "la" as well as a search for these characters. Ticking the desired type will shorten the list of matching objects significantly.

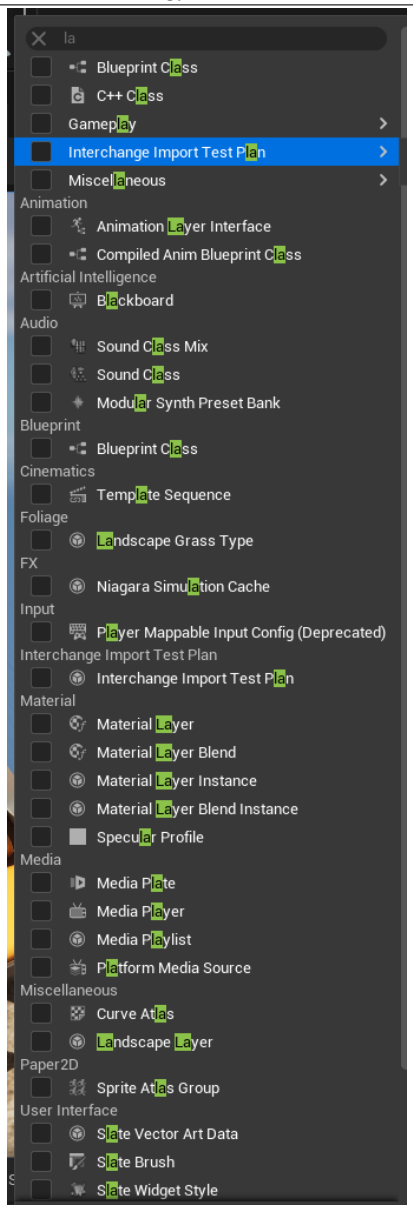

Figure 2.20: Filtering Example

At the bottom of the Sources Panel the Collections Panel is located. Collections can be used by to organize assets into groups. The advantage here is that these are logical collections:

Take the chair included into the starter content. Should this chair added to a group of ”Yellow” objects or, rather, to a group of ”furniture” objects? Basically, obviously, both. However, copying the chair is inefficient. Collections solve this problem by being logical groups, i.e. assets are not copied, just linked to.

New collections can be created by simply clicking the respective add (+) button. It is also possible to directly add an object using the search bar, since once an object has been found using the search, the + button appears to the right of the search bar. This, though, is a dynamic collection: not an object link is added, but the search is saved. I.e. the search is carried out each time the collection is navigated to.

2.4.1 Importing Assets

The Content Drawer can be used to add or import assets, i.e. from other project or external sources. At the time of writing, there are two basic options:

- The + Add menu, which reflects the classic way of adding and importing of assets
- The Fab menu, which has been introduced very recently. It appears that this method of adding assets is in heavy development at the time of writing. Basically, it provides the means to directly add assets from the *Fab.com* platform. It is expected that *Fab* integration will be made much more comfortable in the near future.

For example, adding the assets as provided in the First Person template is a s easy as + Add » Add Feature or Content Pack » First Person » Add to Project.

This way, also the assets of the Starter Content can be added, if the user didn't do so during project creation.

2.4.2 Saving Assets

It is important to note that added assets are not persisted by dragging them into the project. Those assets need to be saved first; which can be accomplished using File » Save All or the shortcut Ctrl + Shif + S.

2.4.3 Content Drawer Settings

The Settings menu of the Content Drawer can be used to adjust its settings as desired.

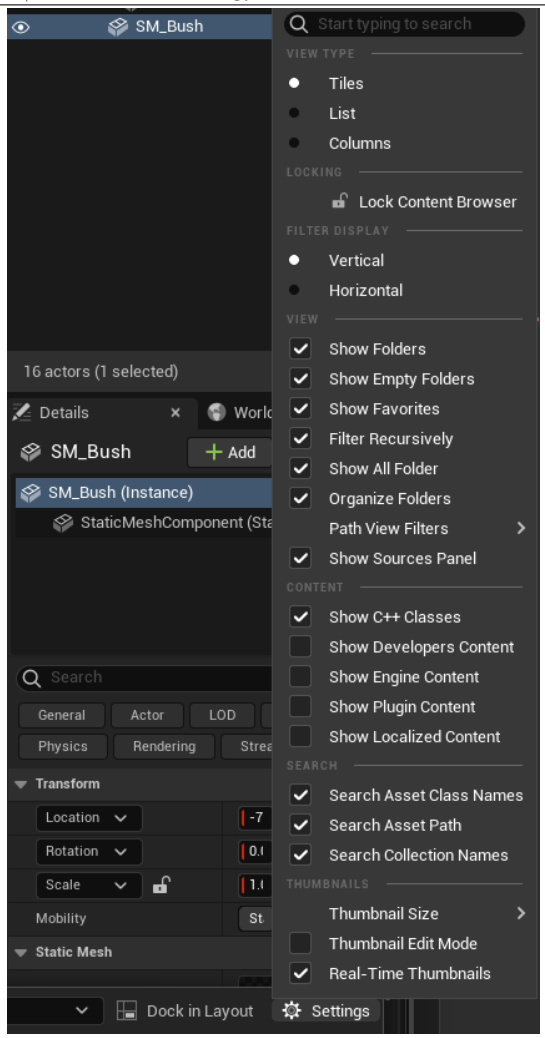

Figure 2.21: Content Drawer Settings

See 2.21 for details.

2.5 The Details Panel

The Details Panel is the context sensitive area to the right of the viewport, just below the Outliner. It displays lots of information about the selected actor, and most of it can be edited.

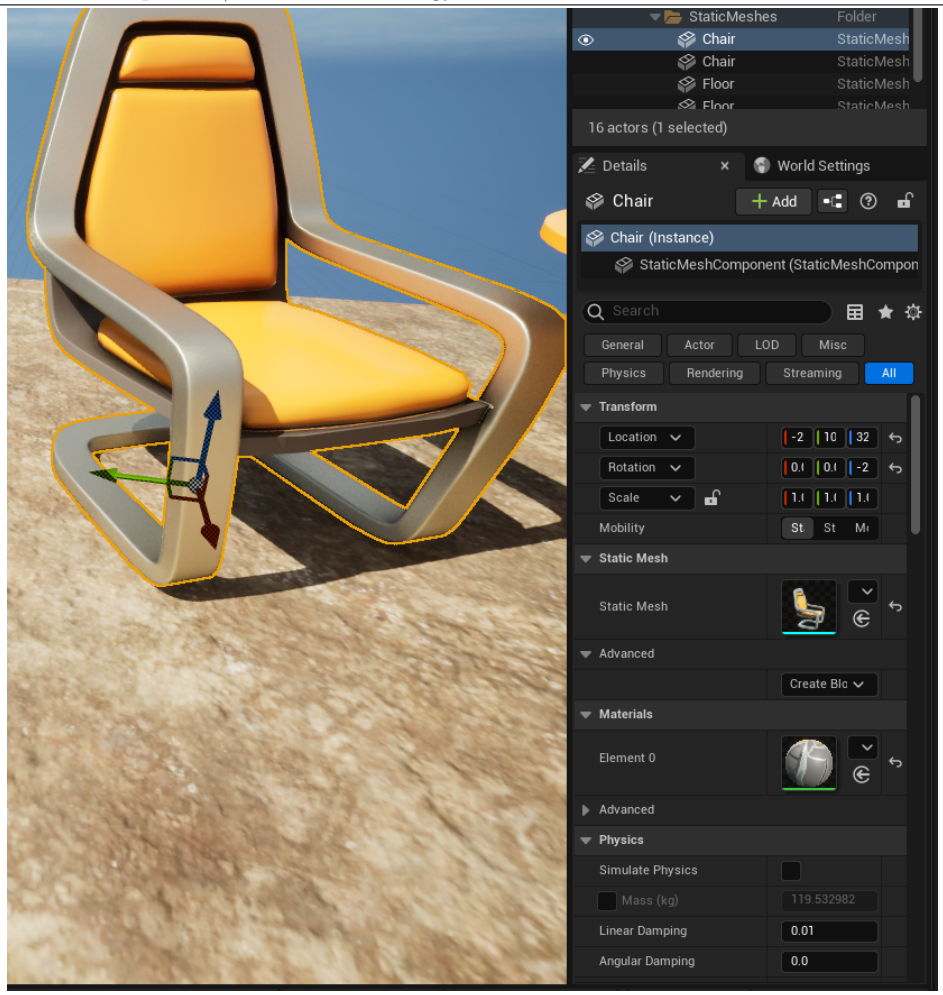

Figure 2.22: The Details Panel

At the top of the Details Panel, the name of the actor is displayed, and it can be edited of course.

To the right there are buttons for adding components to the actor besides as well as for creating a Blueprints for this actor. Below that, the component structure is depicted. More on Blueprints and Components later in this document.

Since the options within the Details Panel can become quite numerous, a search bar is included, enabling search for desired options.

The Details Panel mostly consists of properties of the selected actor, grouped into Categories.

A group of buttons, carrying names like General, Actor, and so on, can be used to slim down the number of options depending on the activated option.

2.5.1 Details Panel Settings

To the upper right the Settings menu for the Details Panel is located.

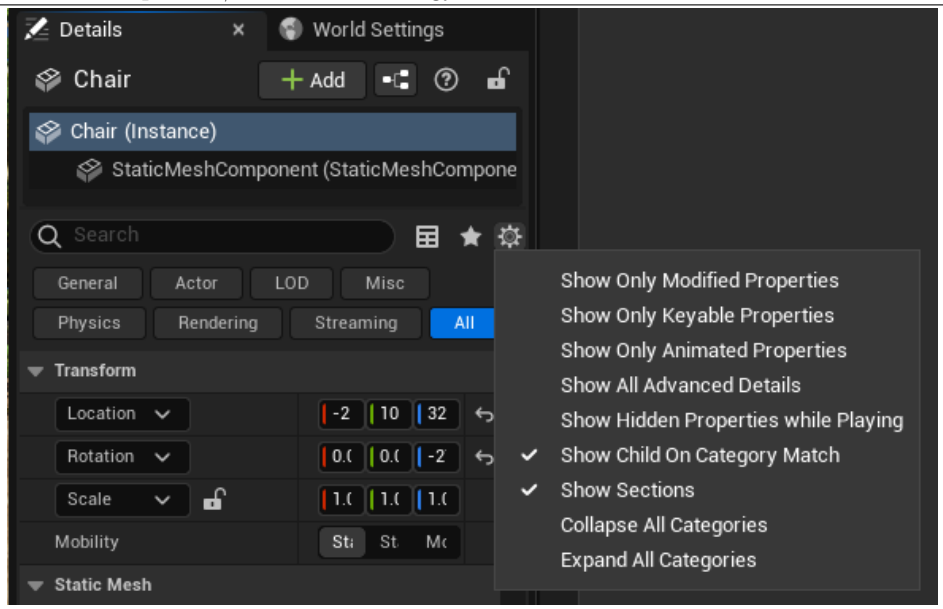

Figure 2.23: Details Panel Settings

Most of the settings - see 2.23 - seem self explanatory.

2.5.2 The Transform Category

The Details panel has a lot of functionality in it that is specific to the type of actor selected. However, the Transform category is common to all actors.

In section 2.3.6, transforming actors using the Translation, Rotation, and Scaling widgets have been discussed. However, there is another way of moving, rotating, and scaling actors, using the Transform category of the Details Panel.

Of course, the widgets are useful when placement and scaling do not need to be exact and level design is to proceed quickly. But for fine precision or when exact values are a requirement, the Details Panel may be used to insert exact values manually.

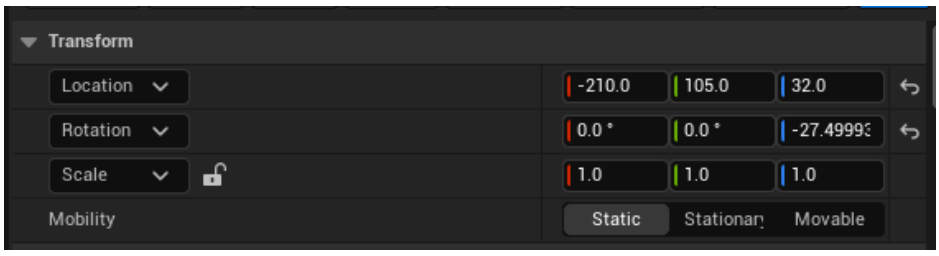

Figure 2.24: The Transform Category

Numbers can be entered manually or left-clicked and subsequently mouse-dragged left or right to de/increase the values, respectively.

As figure 2.24 shows, there is an additional setting present, denoted Static, Stationary and vMovable.

Mobility is a setting that applies mainly to static mesh actors and light actors.

Static means that the actor will remain stationary the entire time, while Moveable means that it is possible for the actor's location to change. To clarify, the "static" in "static mesh" refers to the fact that the mesh doesn't have any moving parts relative to itself, while the "Static" mobility setting means that the actor's location will never change.

These settings are used for efficiency. When Movable is set to Static, this means the actor will never move, therefore things like lighting can be computed beforehand.

With this set to Movable, this is impossible, therefore the game will be computationally more expensive. Note, though, that using Nanite (a new rendering technique) only works with vMovable lights; since Nanite is very efficient, the recommendation for now is to use the Movable option by default.

2.6 The Outliner

By default, the Outliner is located in the upper-right of the Level Editor. Basically, it is an organized list of all the actors in a Level.

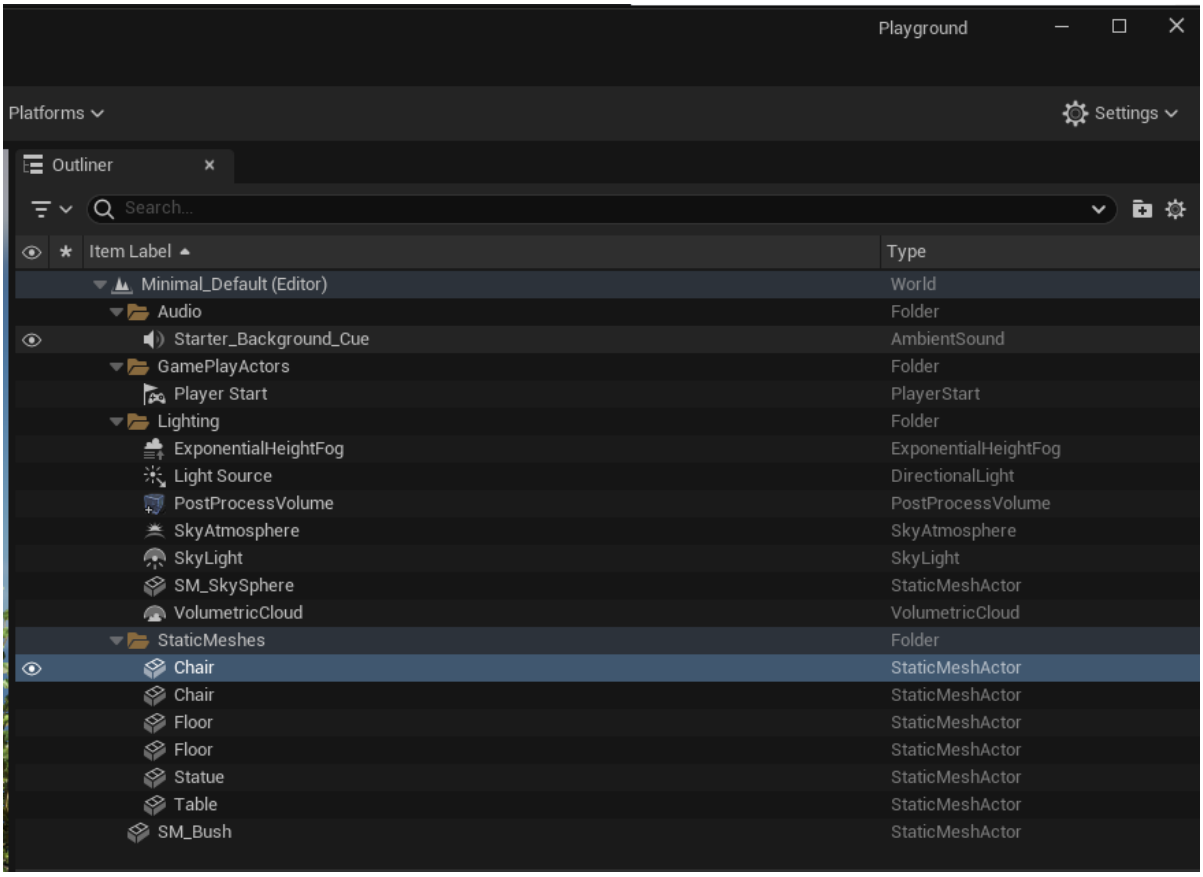

Figure 2.25: The Outliner

Clicking an actor in the Outliner is going to select it in the viewport (and vice versa). Double-clicking it or pressing F will focus on that actor. The actor can be renamed here as well.

To the left of each actor (below the eye symbol) the visibility can be set. Setting an actor invisible there (by toggling with the LMB) will make the actor invisible in the viewport. Note, that this only applies to the viewport; the actor will still be visible when the game is played. This is useful when an actor is hidden by an other actor or there is in general a lot of clutter present in the scene.

2.6.1 Attaching Actors

The Outliner can be used to attach actors to one another. In Unreal Editor, each actor can have a parent actor it is attached to. Simply dragging an actor onto a different actor will make that actor a child of the other actor; the relationship is denoted by the indentation. This is useful, as moving a parent actor moves all the attached child actors as well.

2.6.2 Grouping Actors

Actors can also be grouped, which is somewhat similar to attaching as discussed in 2.6.1, albeit without the parent-child relation.

To group actors, select all actors that are to be grouped an press Ctrl + G; this will formally group all the actors together. Groups are made visible by green brackets surrounding them. Shift + G will ungroup them. Alternatively, with an actors selected, right-clicking them, under Groups » Ungroup will also remove the group.

Grouping actors into folders also helps organizing them, a particularly useful feature in larger projects.

Also notice the search bar, which can be used to find actors as well.

Actors

In sections prior, it has been discussed what an actor is, how to place actors into a level, how to translate, rotate and scale them. In this chapter we will discuss different types of actors, how each type can enhance game levels in sophisticated and unique ways. An actor used to prototype levels will be discussed along with actors representing physical objects.

Also, actors that generate light, fog or define a volume of space will be discussed.

3.1 Static Meshes

A mesh is simply a 3D model of an object. There are two specific types of meshes that can be used as actors in *Unreal Engine*. These are the Static Mesh and the Skeletal Mesh.

A Static Mesh is a type of mesh that does not bend, deform, or change shape in any way. A Static Mesh can still move around on the screen, it just can't animate.

So, for example, a Static Mesh in the shape of a cube could be used to represent a cardboard box, hat box could slide across a surface, or fall off a table, or fly across the room... but it could not have flaps that open and close.

For objects with moving parts, Skeletal Meshes could be used. With a skeletal mesh, it is possible to define individual parts of the mesh and how those parts connect and move with one another.

There is a number of different Static Mesh actors in the Create menu. In the Shapes category, there are Static Meshes in the shape of cubes, spheres, cylinders, cones, and planes. Any one of these could be dragged into a level, positioned, rotated and scaled as desired.

Usually, though, meshes will be imported from external sources like a 3D modeling software or platforms like Fab.

3.1.1 Replacing Static Meshes

Sometimes during development, meshes shall be replaced by other meshes; maybe a better one or to replace a prototype with the real thing.

Of course, the hitherto used mesh could be deleted and the new one added to the level. However, this is not necessarily very efficient, since the actor may have Blueprints attached or have been customized in some other way.

Therefor, a better way would be to use the Details Panel to replace the Static Mesh that the actor is using. So in *Unreal*, the Static Mesh itself is actually a property of the StaticMesh Actor. Therefore, the developer could just select the actor, then go to the Static Mesh category of the Details Panel to replace the mesh being used by that actor. Notice, that there is a search bar; for example, if the developer intends to change the chair for a couch, she could just click the Static Mesh, search for couch and select the desired mesh.

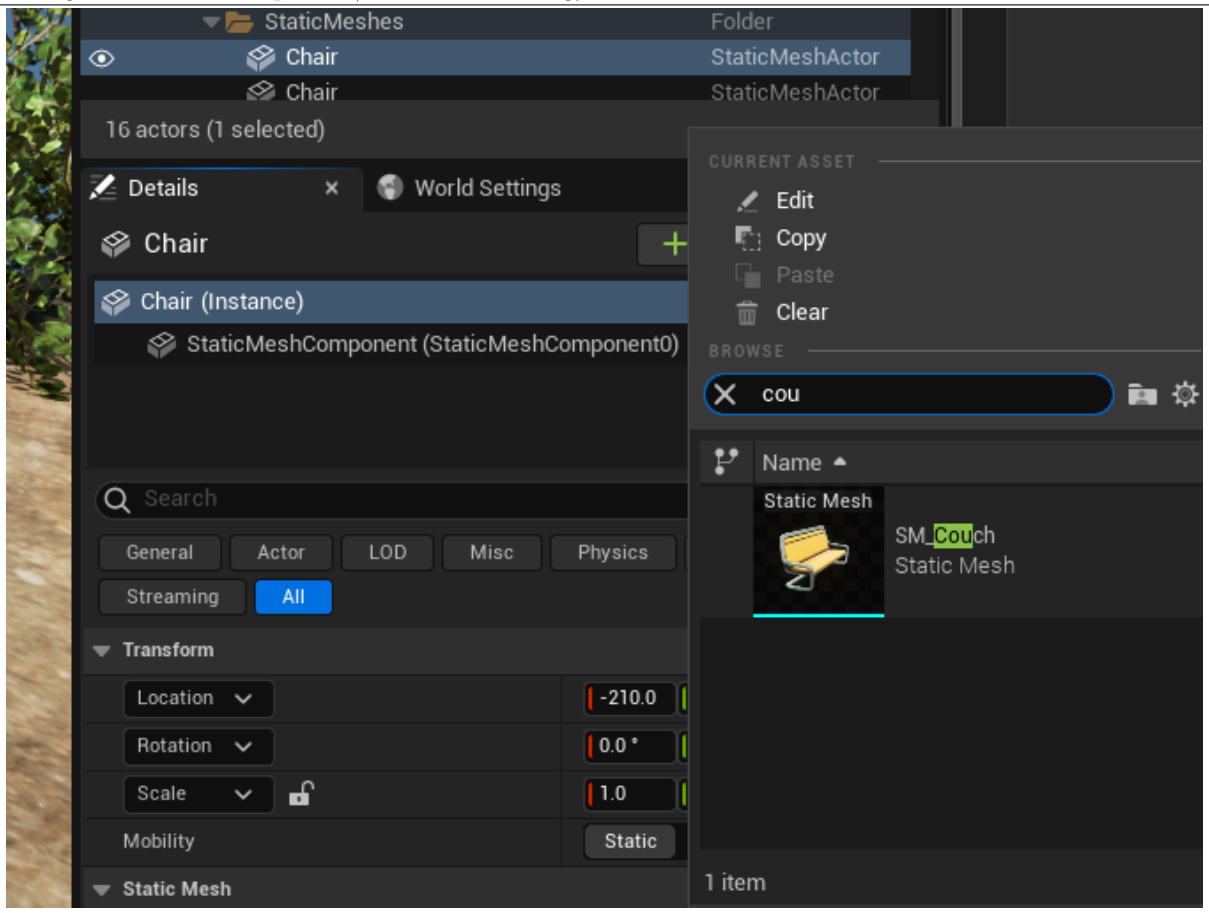

Figure 3.1: Replacing Static Mesh Component

The replaced actor will have the same properties the original had, including position, rotation etc. Technically, the actor has not been replaced; it was just the actors static mesh that changed.

Alternatively, selecting the couch in the Content Drawer, changing back to the Details Panel and clicking the ← icon details section will accomplish the same.

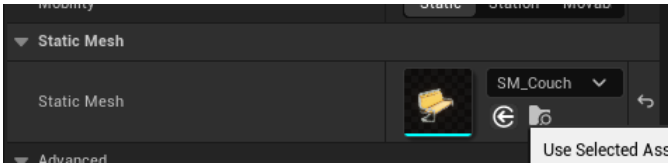

Figure 3.2: Change Mesh Arrow

3.1.2 Physics

By default, Static Meshes have physics turned off. For example, placing a cube inside a level, somewhere above the ground, and starting the game will not cause the cube to fall down to the floor.

There are several reasons for this. Checking the cube's details panel reveals the issues:

- The mesh is set to static. This tells the engine, that the actor will never change its position. Setting this to Movable enables, for example, to create Blueprints containing some movement logic telling the cube how to move.

- In the Physics section, Simulate Physics is not ticked. Selecting Simulate Physics will cause the cube to fall once the game is started.

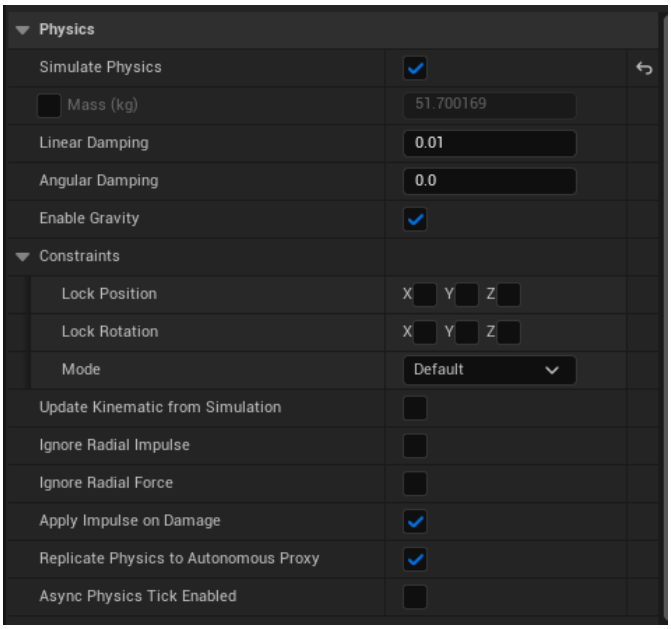

Figure 3.3: Physics Section of Details Panel

Notice (see 3.3) that other physical properties, like mass (measured in kilograms), could be set as well. Apparently, heavy objects will feature different behavior compared to light objects.

These properties are also taken into account if the actor is pushed, hit by a projectile, and so on.
The properties Linear Damping and Angular Damping refer to the amount of drag applied to movement and rotation, respectively. For objects lying on a floor, drag is equivalent to friction.

This is used, for example, to model block of ice featuring a small amount of drag, while rough rocks will of course feature a large amount of friction.

Similarly, an object with a small amount of Linear Damping will spin for a long time when hit by an object.

Also notice, that Gravity can be switched on and off separately, which is very useful in space simulation games.
The Constraints section locks position and rotation properties to the 3 axes; the Mode property can be used to select from some presets or to define a custom plane that doesn't run directly parallel to one of three main axes.

Finally, Replicate Physics to Autonomous Proxy is used in multiplayer games and should be checked if the server is in charge of keeping everything in sync, and unchecked if the clients are responsible.

3.2 Geometry Brushes

In the world of 3D modeling, a Brush or more precisely, Geometry Brush is simply a 3D area of space. So this is nearly identical to the understanding of what a Mesh is, but there are several key differences between Brushes and Meshes.
First off, Brushes are used for more basic shapes. E.g. in the Place Actors panel, in the Geometry category, the available Brushes are located. These consist of basic geometric shapes and as well as some Brushes in the shape of stairs.

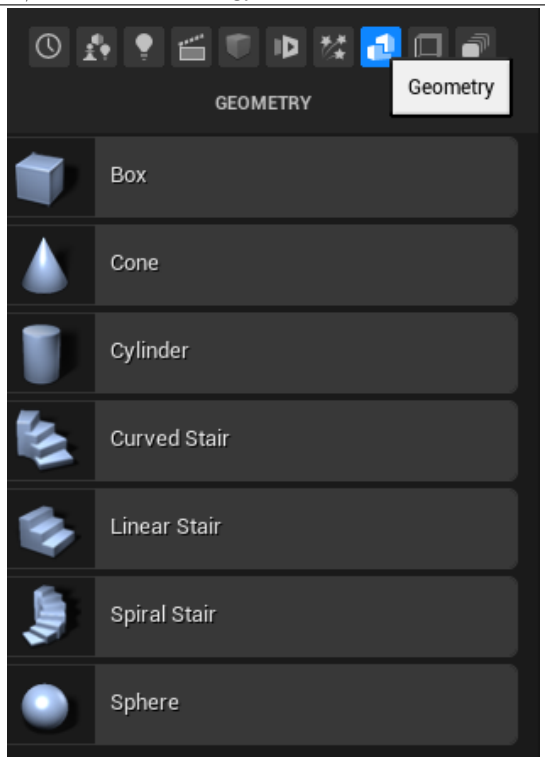

Figure 3.4: Geometry Brushes in the Place Actors Panel

Notice that there is also a Shapes category, which contains Static Meshes rather than Geometry Brushes. As stated earlier, meshes can be much more complex than this.

The meshes contained in Starter Content, have mostly been created in a 3D modeling program to take the shape of a chairs, couches and so on.

Geometry Brushes, Static Meshes and Memory

The second key difference is in how the Unreal Engine handles brushes and meshes in memory. Brushes are value types, while meshes are reference types, i.e. when brushes are copied, each copy increases the amount of memory required, whereas when meshes are copied, only a pointer to the object is stored.

Thus, meshes a much more memory efficient, hence are advantageous with regard to performance.

In other words, even though meshes look better an require less memory, brushes still have a reason to exist: brushes are easier for prototyping. I.e. brushes are often times used to quickly prototype a level, only to be replace later on by meshes.

It is not uncommon that initial versions of levels consist exclusively of brushes, whereas the shipped version of the game does not consist any brushes at all.

3.2.1 Brush Types

Just as any actor, properties of brushes can vary wildly. There is one property, though, that is common to all Geometry Brushes: the Brush Type. Brushes can be either **Additive** or **Subtractive**.

As expected, an *additive* brush is one that adds geometry to a level. A *subtractive* brush, on the other hand, is going to remove from a mesh shape. This is useful, for example, to "*add holes*" to a shape, like windows or doors of a house.

In other words, if a subtractive brush overlaps an additive brush, the subtractive part will be removed from the additive part. This feature is one of the main reasons why brushes are so well-suited to quickly sculpting overall level design. By adding and subtracting geometry in this way, it is possible to sculpt the layout of a Level much

faster than it is possible when using meshes.

Once a brush is added to a level, it can be modified using the Details Panel.

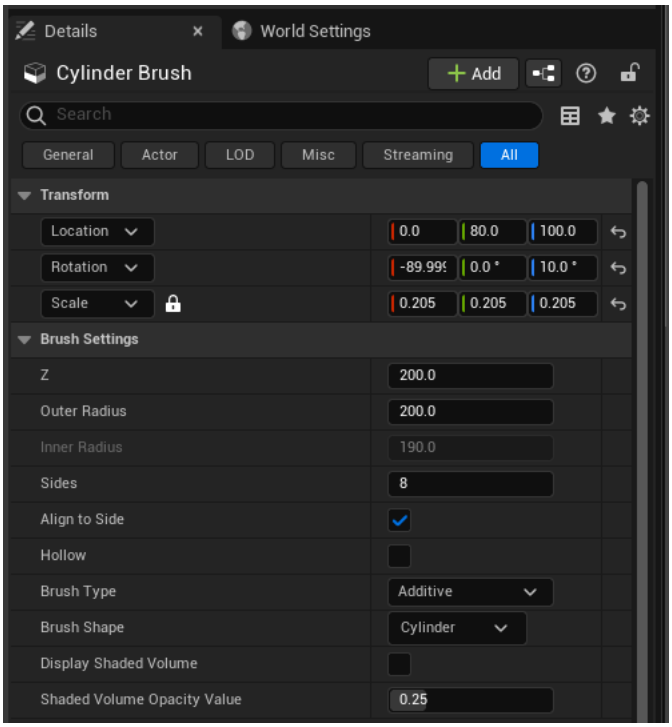

Figure 3.5: Geometry Brush Details Panel

As depicted in figure 3.5, the brush type can be set easily using the Additive/Subtractive pull-down menu. Note, that it is also possible here to change the shape of a brush; for example, a cone mesh could simply be swapped to a cylinder brush using the pull-down menu.

Some of the properties in the Brush Settings are dependent on the shape of the brush being used. For example, a Box brush will have X, Y, and Z properties denoting the length of the box in those dimensions. Adjusting those will change the size of the box. So the box brush has these X, Y, and Z properties, but the Cylinder and Cone brushes have their size determined by just a Z length, and an Outer Radius property.

So the Z denotes how tall the cone or cylinder is, and the Outer Radius is how wide around it is. For the Cone, this is the radius as measured at the base. For the Sphere brush, radius is the only property used to determine its size, and so on and so forth.

Note, that curved edges are not exactly curved. Rather, those ”curves” are approximated by a series of flat sides. The more sides, the smoother the shape appears and the more the shape appears like curved to the human eye. The cylinder in figure 3.5 has its Sides property set to 8. Increasing this would make the cylinder appear more cylinder like, since curves appear more ”curved”.

The Cone and Cylinder brushes also have this property Align to Side. If this is checked, it will align the sides of the Brush with the Grid.

With that checked, this side of a Cone here is perfectly parallel to the Grid lines. But if this is unchecked, it won't necessarily align with the grid.

Another property of Brushes that is common to the Box, Cone, and Cylinder brush, is whether or not the Brush is Hollow. If this property is checked, instead of the Brush being solid all the way through, it will be hollow inside, and the brush will essentially be a shell, with walls of some thickness. For box brushes, this is set with the Wall Thickness

property.

Also check out the other, numerous properties of brushes.

3.2.2 Stair Shaped Brushes

Stair-shaped brushes have their own unique properties.

Linear Stair

These possess `Length, Height`, and `Width` properties, each affecting one of the three axes in space.

Adjusting the number of steps for the staircase will increase the stair's length. The `Add to First Step` property is essentially a `Step Height` property for the first step only.

Curved Stair

The `Curved Stair` has some of the same properties as the `Linear Stair`, such as `Step Height, Step Width, Number of Steps`, and `Add to First Step`. It also has some other properties as well that pertains to its curve. The first of these properties is the `Inner Radius`.

Spiral Stair

The `Step Height` property is a little different on the `Spiral Stair` than it is on the other two stairs. With the `Spiral Stair`, the `Step Height` won't affect how tall each step is, it will affect how much each step overlaps the step above and below it.

The `Num Steps Per 360` determines how many steps there are to make one full circle. Consequently, the `Num Steps` property sets the total number of stairs. So, for example, if both of these numbers are set to the same value, the stairs are going to wrap around exactly once. If the `Num Steps` is double the `Num Steps per 360`, the stair is completing exactly two full circles, and to forth.

3.3 Materials

While this chapter is named "Actors", technically materials are not actors. Rather, they are properties of mesh and brush actors.

Materials within the context of the *Unreal Engine* are assets that can be applied to surfaces of meshes and brushes to make them look like they are made out that substances. This also entails the look of surfaces. They do not, though, possess any physical properties like friction; these are properties of the mesh or brush itself.

To apply a Material to a surface, it must be selected `Content Drawer` and dragged into the level viewport, onto the surface desired.

There is a catch, though: if the mesh/brush consists of a multitude of surfaces, like for example a stair brush, applying material can be a tedious task, since the Material needs to be applied to each and every surface.

Therefor, there is a second option to apply materials: that is, by applying the material *first*, i.e. select the desired material in the `Content Drawer`, thereafter - with the material selected - drag the desired actor into the level. This way, the material will be applied to all surfaces of that actor.

To apply a material to *all* surfaces of an already existing brush, go to the `Details Panel` > `Geometry` category, click `Select`, select `Select All Adjacent Surfaces (Shift + J)`. Now all surfaces of this particular brush are selected, so now the desired material can be dragged and dropped from the materials content drawer.

Another way to apply a material to a surface is the `Details Panel` > `Surface Material` category. This works similar to replacing the `Static Mesh` discussed earlier, e.g. by using the materials drop down menu:

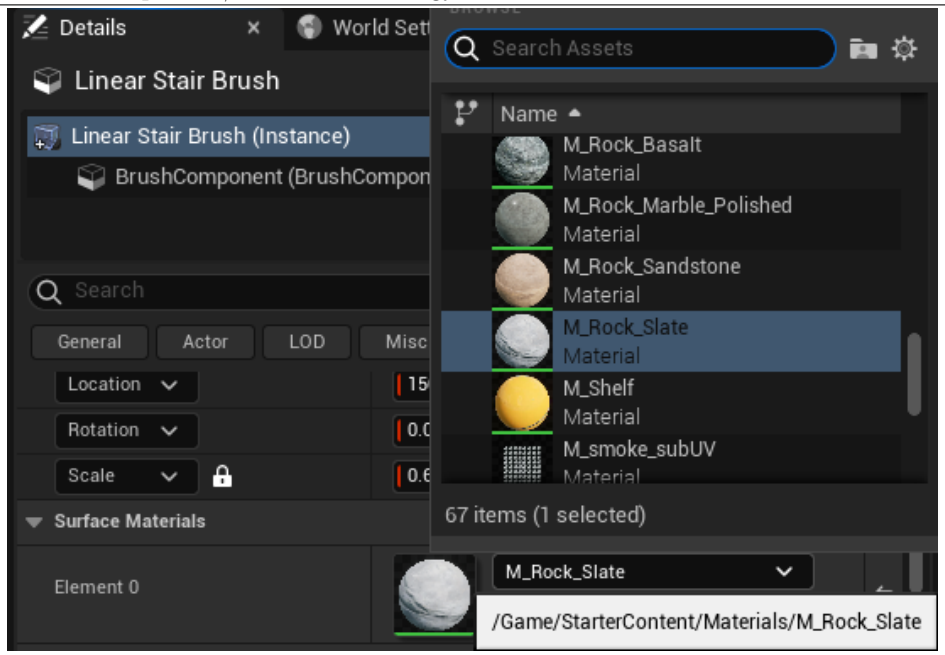

Figure 3.6: Details Panel Material Category

Another method is to select the material in the Content Browser and subsequently clicking the ← icon in the Details Panel.

3.3.1 Elements

Single meshes possess the ability to have different materials materials applied to different parts of it. When a mesh gets created in a 3D Modeling program, such as Maya or Blender, if it has different materials applied to different parts of its surface, once that Mesh gets imported into the Unreal Editor, each of those sections of surface become known as Elements. It is then possible to apply different materials zu each element. If a material is imposed on an object in the viewport, this material will affect the given Element only. Now, if a brush containing different elements with different materials applied, this will be visible in the Details Panel » Materials category.

3.3.2 Textures

Basically, Textures are what Materials are made of, i.e. a Material is made up of one or more textures, and each texture is just an image file that defines one of the properties of the material. While one Texture may be the actual colors of the material, another Texture maps its smoothness or roughness, and so on. This data is combined to form the composite Material. There is a dropdown showing the Textures that make up the currently applied Material, and if selected, it will open that Texture in the Content Browser.

3.4 Lights

In *Unreal Engine*, a light is an actor generating light within levels. Lights are not meant to represent an object producing light, like e.g. flashlights, lights only represent the light itself.

Therefore, in order to create said flashlight, the developer needs to create a mesh in the form of a flashlight, containing materials and all the things making up a decent object, and add the light actor to create the light itself.

There are 5 distinct types of light within *Unreal Editor*.

3.4.1 Directional Light Actor

The Directional Light Actor is used to emulate light coming from an extremely long distance away, such as outer space. All the light will hit the level at the same angle, meaning all shadows produced by this light will be parallel. This Actor is used primarily for sunlight and moonlight.

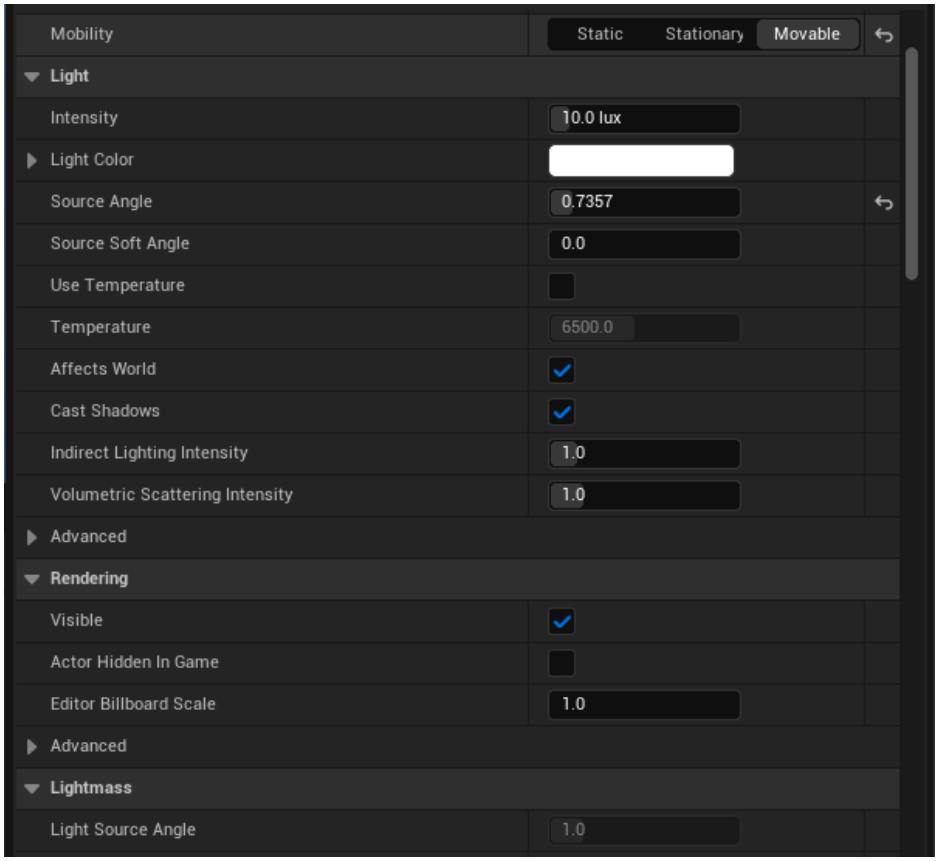

Figure 3.7: Light Properties

Intensity Controls light brightness. If the intensity of a light is increased, the light get brighter and vice versa.
Color The default light color is white, but it can be changed using the Details Panel.
Source Angle and Source Soft Angle These basically affect how soft the shadows produced by the light are. The higher the value, the softer the edges of the shadows will be. The default value of 0.5357 for Source Angle is meant to represent how shadows should look in natural sunlight. These settings take advantage of raytracing technology.
Temperature Sets the light temperature. By default it is not active, but it can be activated by ticking the respective box.
Cast Shadows Shadows are computationally expensive, which is why they can be deactivated to improve the performance of a game. If the game world should be as realistic as possible, this should be enabled.
Indirect Lighting Intensity In case light gets reflected off another surface, the reflected light is called indirect lighting and can also light up objects in the level. This property will determine how much this reflected light affects the other objects it shines upon.
Volumetric Scattering Intensity This property applies to fog; when light passes through fog, it will scatter in various directions. The higher the Volumetric Scattering Intensity, the more the light will scatter.

3.4.2 Point Light Actor

A Point Light produces light emanating in all directions at the same time. This is useful for mimicking the light coming from a light bulb, or fire, for example.

It has many of the same properties that the Directional Light has, plus a few additional ones.

Attenuation Radius This determines how far from the source the light will still affect objects; in the viewport, it is represented by a blue sphere (see figure 3.8 below). The higher the Attenuation Radius, the larger this sphere will be, and the farther the light will extend from its source.
Source Radius, Soft Source Radius, and Source Length The light from a Point Light will actually emanate from a single point in the level. Now, if that light is placed above a shiny floor, there could be a reflection of the light source in the floor, in which case the reflection should resemble the light source. These three properties can adjust the size and shape the light source will appear in reflections.

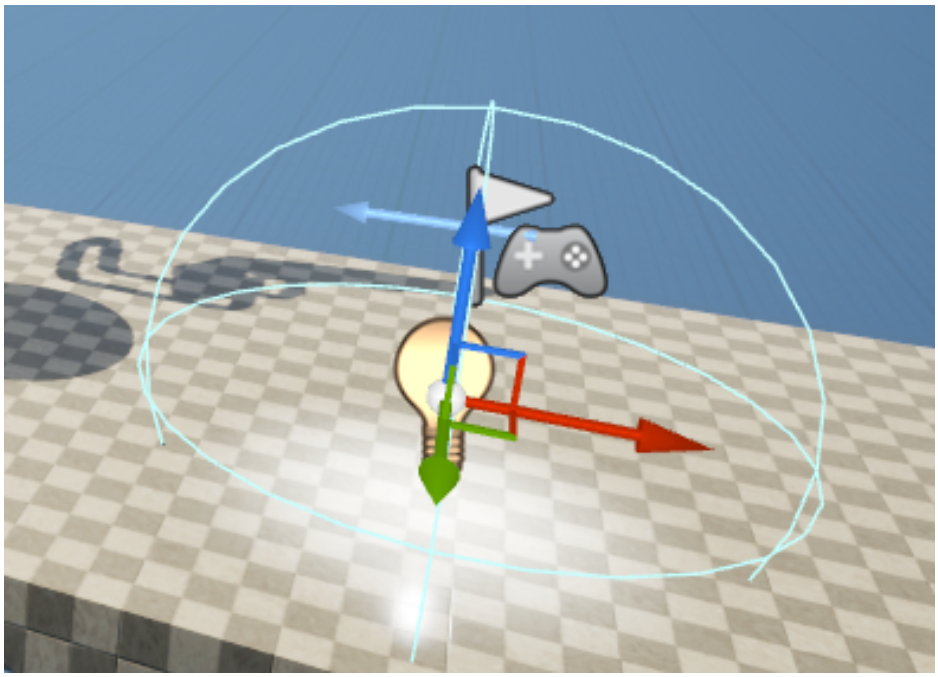

Figure 3.8: Attenuation Radius Sphere

3.4.3 Spot Light Actor

The Spot Light, on the other hand, will emit light in the shape of a cone. Therefore, this is like the light coming from a flashlight, or as the name suggests, a spot light, resembling the lights used in theaters.

The Spot Light is very similar to the Point Light, except that instead of shining light in all directions, it shines it in a specific direction, in a cone shape. Therefore, the Spot Light features all the same properties as the Point Light, with the addition of the following properties:

- **Inner/Outer Cone Angle** Within the inner cone of the Spot Light, the light will he at its brightest and will be just as bright at any spot within the inner cone. From the outer edge of the inner cone to the outer edge of the outer cone, the Intensity of the light will gradually fall off to nothing.

3.4.4 Rectangular Light Actor

Rectangular Lights, a. k. a. Rect Lights, projects light sources out of rectangular planes. The idea behind this is to simulate light sources resembling those that emanate from monitors, TV sets, smartphones or overhead lighting, etc.

The Rectangular Light features some unique properties as well. These are:

- **Source Width and Source Height** Used to denote the size of the light source.
- **Barn Door Angle and Barn Door Length** Used to emulate Barn Door Angle and Barn Door Length properties to emulate those kinds of effects.
- **Inner/Outer Cone Angle** Within the inner cone of the Spot Light, the light will he at its brightest and will be just as bright at any spot within the inner cone. From the outer edge of the inner cone to the outer edge of the outer cone, the Intensity of the light will gradually fall off to nothing.
- **Source Texture** Textures can be applied to a light, which in turn appears like as if it is being filtered through that texture.

3.4.5 Sky Light Actor

The Sky Light Actor is an actor meant to emulate light reflected off the atmosphere and other distant objects. The idea is: if a light comes from the sun or moon, a large chunk of that light approaches as direct sun- or moonlight, representing the Directional Light Actor discussed above. However, another chunk of that light hits particles in the atmosphere or clouds and subsequently get reflected off these objects. The Sky Light Actor represents that light that gets scattered in the atmosphere or reflected off of other objects. It therefore comes through as weaker, indirect, from all sorts of different angles. In simpler terms, it represents the faint glow of the atmosphere.

Properties unique to the Sky Light are:

- **Real Time Capture** If enabled, actors like the Sky Atmosphere or Volumetric Clouds will affect the Sky Light. Therefore if mentioned actors are present in the level, this property should probably be enabled.
- **Source Type** Recall: the Sky Light is used to represent the reflection of light from the atmosphere or far away objects in the sky such as clouds or mountaintops. And to determine at what distance this light should appear to emanate from, it needs to be defined at what distance the sky should be considered to start at. By default, this Source Type property will be SLS Captured Scene, which just means that the sky will be defined as any point that is the Sky Distance Threshold away from the Sky Light actor. Therefore, if the Sky Light actor is placed at the center of the Level, with a Sky Distance Threshold of 150,000, we are saying that the sky should begin 150,000 units from the center of the Level.
- **Real Time Capture** If enabled, actors like the Sky Atmosphere or Volumetric Clouds will affect the Sky Light. Therefore if mentioned actors are present in the level, this property should probably be enabled.

3.4.6 Mobility Settings

The light actor's mobility settings somewhat are at a transitional stage at the time of writing.

Basically, as all actors, light actors possess one of three mobility properties: **Static, Stationary** or **Movable**. That is,

- **Static**: cannot move, cannot change any other property at runtime.
- **Stationary**: cannot move, but it *can* change color, brightness or other properties at runtime. In particular, some of its vertexes may change, i.e. it can change, transform its shape.
- **Movable**: can move and change other properties at runtime.

As one may expect, the more flexible a light actor is with regard to movement, the more computationally expensive it becomes:

Static lights create static lighting, which means all of its calculations can be processed at compile time, which is why it requires little processing power at runtime.
The Stationary and Movable settings produce dynamic lighting, which means calculations must be performed at runtime, hereby requiring valuable performance that might be required elsewhere.

The issue is that hitherto, the Static setting was the way to go if at all possible.

That being said, as of *Unreal Engine 5.2*, a new lighting system called Lumen became standard.

Lumen is a super efficient, almost magic-like system, rendering dynamic lighting scary realistically while using very little processing power. Therefore, the new recommendation is to use the Movable setting by default.

There is a drawback, though, since Lumen does not support Static lighting. Hence, by default any light from light actors whose Mobility is set to Static will be ignored at runtime.

Also, building the lights is not a requirement any longer.

3.5 Atmosphere & Clouds

This section discusses actors that can help in adding realistic looking atmosphere and clouds to levels, since just adding a Directional Light is not sufficient to create realistic outdoor scenes.

For example, adding a Directional Light actor to a simple scene adds lighting, the scene, however, looks nothing like a realistic outdoor environment.

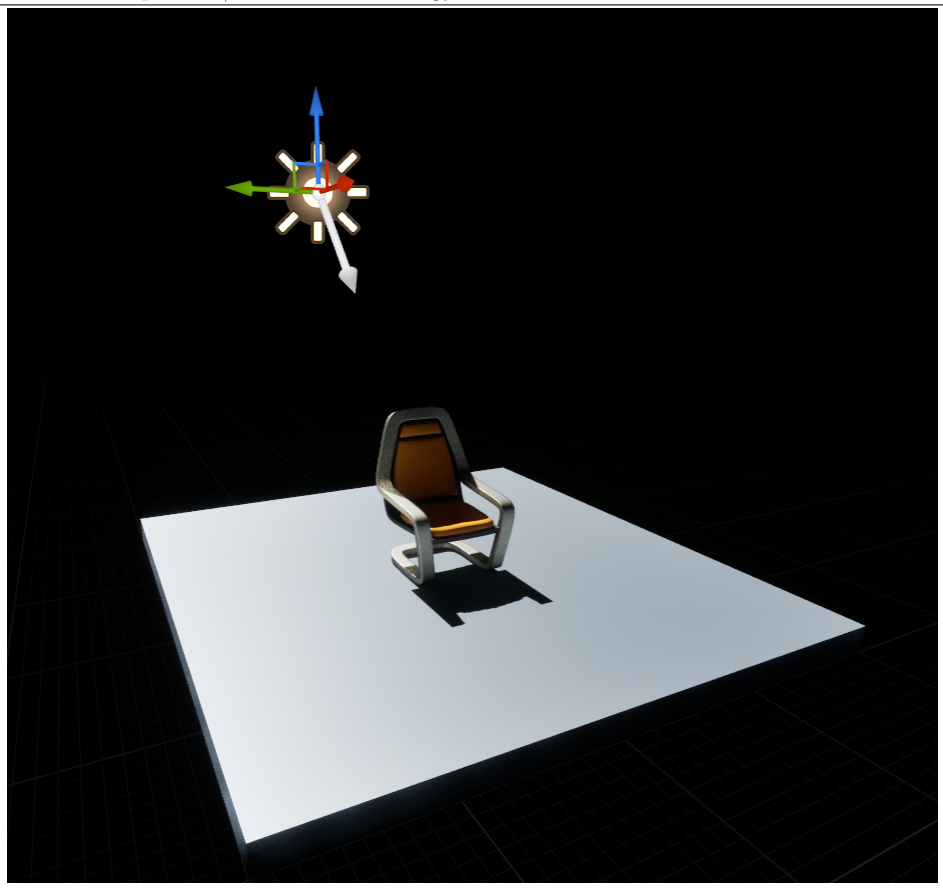

Figure 3.9: Directional Light Scene

If a Sky Atmosphere Actor is added (Create » Visual Effects » Sky Atmosphere), the same scene looks like this:

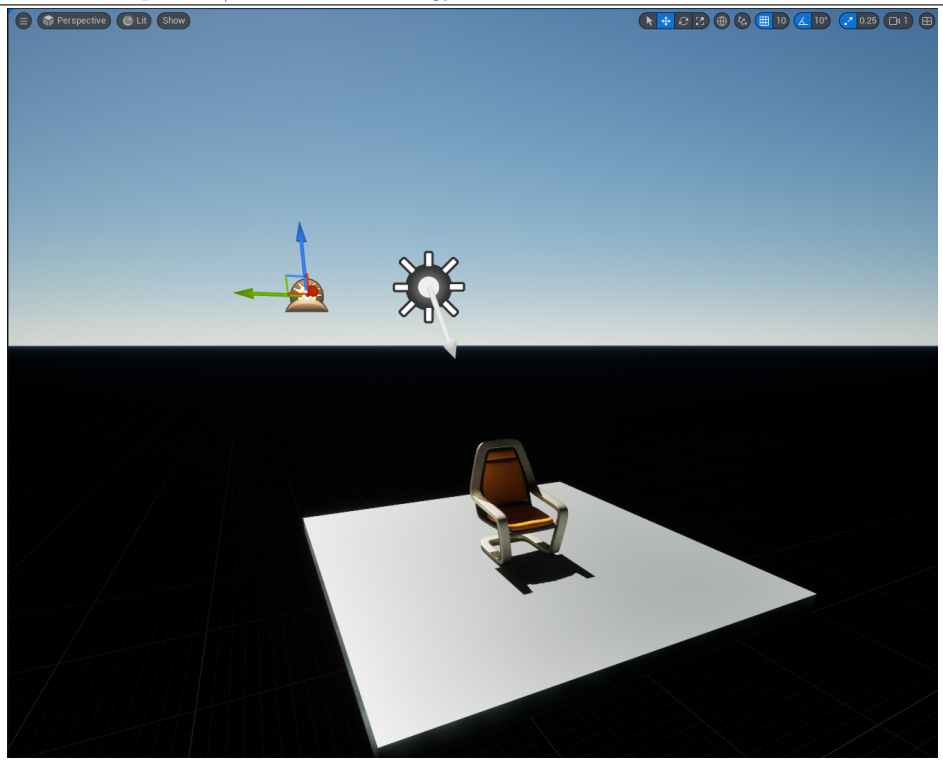

Figure 3.10: Scene: Directional Light & Sky Atmosphere

3.5.1 Creating an Atmosphere

This improves things a little, however, the scene still does not exactly look realistic. Noe, that the sunlight in the scene actually is the Directional Light. Now, selecting the Directional Light, then checking the Details Panel 〉 〉Atmosphere and Cloud category, there is a property Atmosphere Sun Light, which is on by default. This Directional Light Actor will be linked to the Sky Atmosphere actor as long as this setting is activated. Notice, that there are properties affecting the Directional Light affecting the sun disk of the Sky Atmosphere.

For example,

- Source Angle increases the sun disc size
- Rotation changes the position of the sun in the sky. This is because rotation denotes the direction the rays of the sun hit the scene, and this in turn gives the position the sun has to be in order to match this direction (see figure 3.11). Alternatively, this can be accomplished hold down Ctrl + L and moving (i.e. dragging) the mouse. One cool thing about the Sky Atmosphere actor is that it will accurately render what the sky should look like at different times of the day. Therefor, if the Y-rotation of my Directional Light is set to 180 and the Z-rotation to 0, the sun disc will be located on the horizon and the sky and scene will look like a sunset.
- Rotation

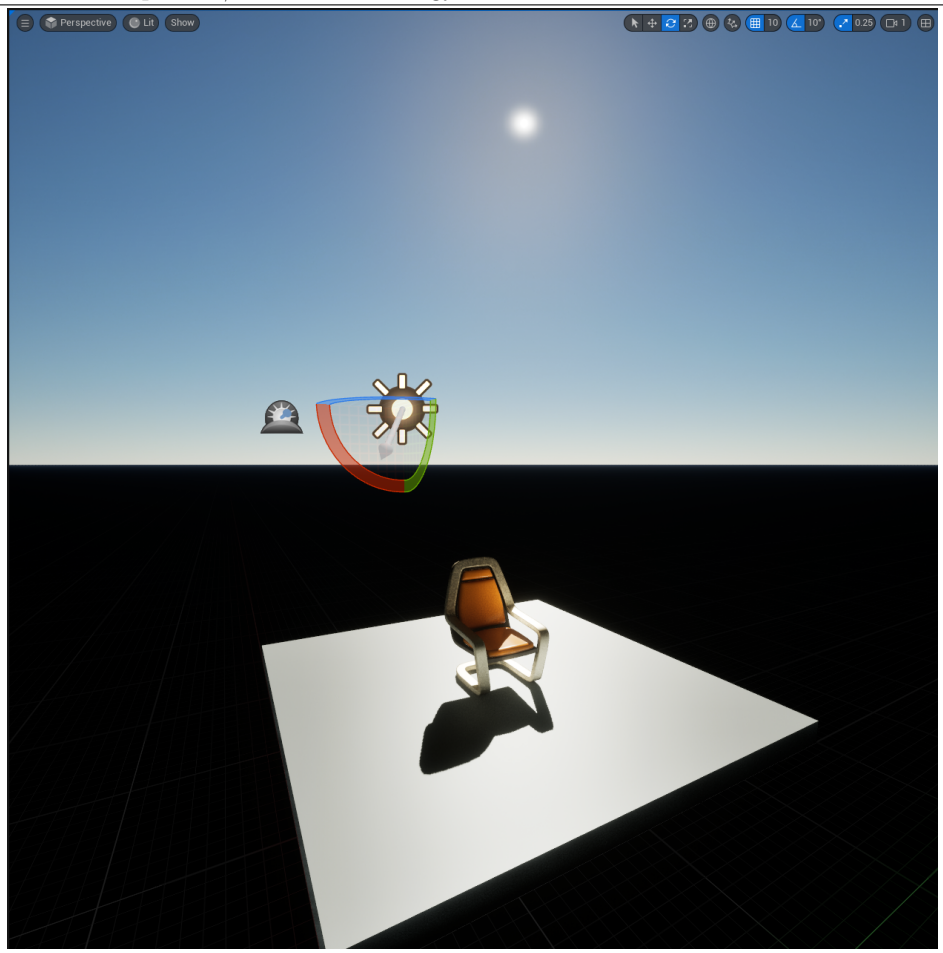

Figure 3.11: Rotation Changed Shadow

At the contrary, noteworthy properties of the `Sky Atmosphere` actor are:

- `Atmosphere Height` the height of the atmosphere, measured in kilometers. This actor allows for seamless transition from ground to outer space.
- `MultiScattering` This settings determines how much scatter, that is, light bouncing around in the atmosphere there should be, where 1.0 is the default.
- `Transform Mode` This defines where the bounds of the planet should be considered to exist, relative to the location of the `Sky Atmosphere` actor. If this is set to the default value, `Planet Top at Absolute World Origin` the surface of the planet, and thus the bottom of the atmosphere, will be considered to be at the world origin of the level, which is located at coordinates (0, 0, 0). If this value is set, moving the `Sky Atmosphere` actor has no effect on the location of the atmosphere in the level. If this is set to `Planet Top at Component Transform`, the surface of the planet, that is, the bottom of the atmosphere will be considered to be at the location of the `Sky Atmosphere` actor. Moving the actor will change the location of the atmosphere in the level. When this is set to `Planet Center at Component Transform` then the center of the planet, not the surface, will be considered to be at the location of the `Sky Atmosphere` actor.
- `Ground Radius` Defines the size of the planet, in kilometers.
- `Ground Albedo` Defines the amount of sunlight reflected off the surface of the planet. This also affects the light scattering discussed above.
- `Rayleigh Scattering`: The scattering of light in the atmosphere due to small particles, such as air molecules. This defines how the sky appears at sunset. Lower values mean a less dense atmosphere and thus less light is scattered and higher values mean a denser atmosphere and thus more light is scattered, affecting the colors of the sunset.

- Mie Scattering: Light scattering due to larger particles, such as dust or air pollution. In this type of scattering, light gets absorbed, causing the sky to appear hazy. The higher this value, the hazier the atmosphere will appear.
- Absorption: This is somewhat similar to the Rayleigh category, and basically you use the Absorption Scale property to set how much light gets absorbed in the atmosphere, and the Absorption property to set which color gets absorbed the most. This is just another way of controlling the overall color of the sky.

3.5.2 Creating Clouds

Clouds are created using the Volumetric Cloud actor. This actor can be found in Create » Visual Effects » Volumetric Cloud. Noteworthy properties are as follows:

- Layer Bottom Altitude: Specifies the distance from the ground, in kilometers, that the clouds should begin to appear.
- Layer Height: Specifies the distance from the ground, in kilometers, that the clouds should stop appearing.
- Material: This gives the actor its flexibility. The overall appearance is determined by the material set in this category. *Unreal Engine* includes a default Material here, called SimpleVolumetricCloud; however, this can be changed to different materials.

3.6 Player Start Actor

In levels without an instance of this actor, the player is going to start off position (0, 0, 0). To assign a different starting spot, this actor is used. It is available in Create » Basic » Player Start.

The Player Start actor can also be used to specify the direction the player should be facing when the Level starts, indicated by a light blue arrow (see figure 3.12).

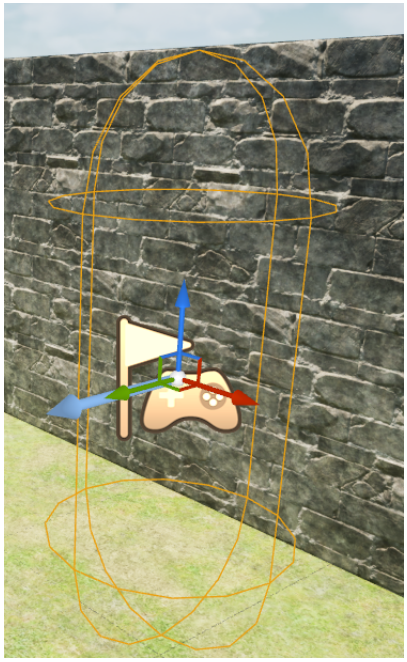

Figure 3.12: Player Start Actor

Hint: Usually, the game starts at the Player Start actor position. To temporarily change that to the current camera position, click the 3 dots to the right of the Play button and select Current Camera Location. This will cause the level to start at the currently active camera position.

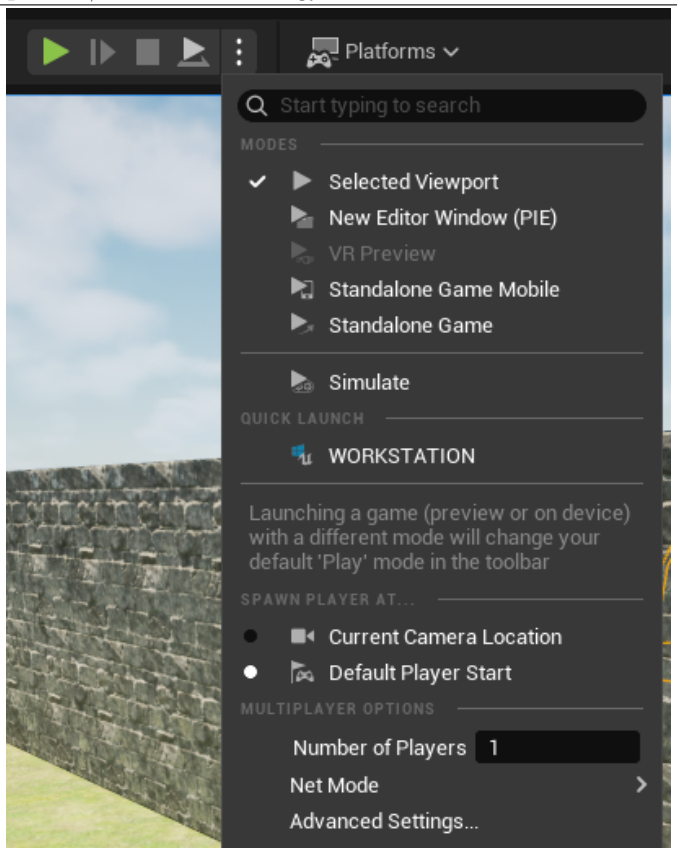

Figure 3.13: Player Start Menu

3.7 Components

Components are different objects or functionality that can be attached to actors. There are many different kinds of Components. Some of the types of Components are objects that are also used as actors on their own.

As an example, Static Mesh can be added as a Component to another actor. Or a light can be attached as a Component to another actor, combined representing a flash light.

Other Components, such as Movement Components, do not possess an own Actor type and are only used as Components on other actors. For example, a Rotating Movement Component attached to an actor will cause that actor to rotate, but would otherwise not be of much use on its own.

3.7.1 Adding Light Components

For example, a flashlight could be created by attaching a light onto a static mesh in the form of a flash light like so:

1. Drag a cylinder into the level. Scale and rotate it so it resembles the body of a flashlight.
2. Select the cylinder, Details Panel > Add
3. A list of component is shown, grouped by category. Enter the term ”light” into the search bar.

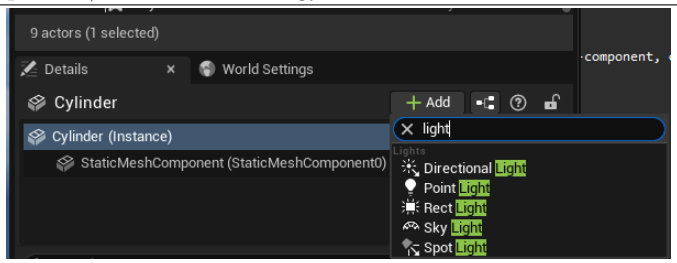

Figure 3.14: Add Light Component

3. Add a spot light to the cylinder.

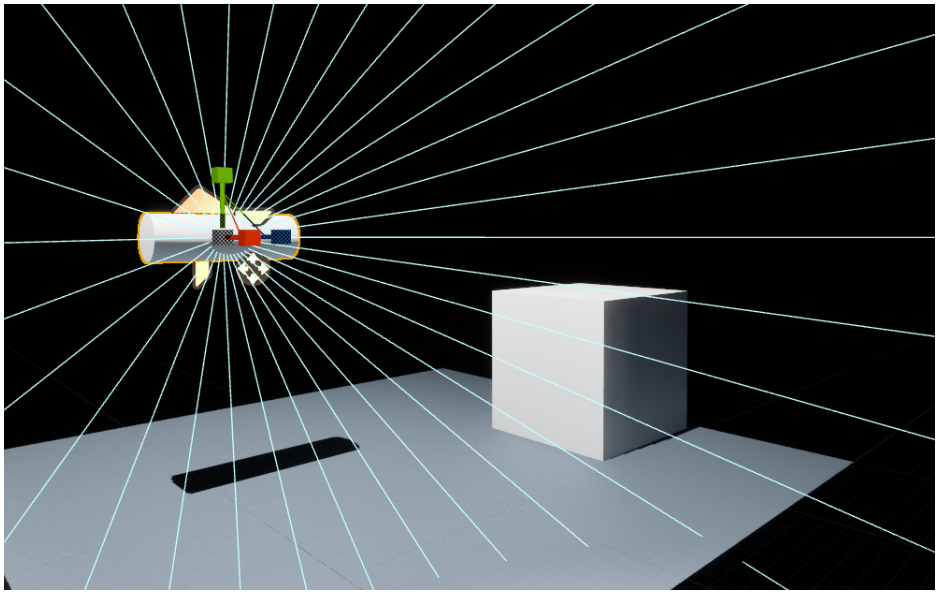

Figure 3.15: Mesh With Spotlight Component

The component is added to the cylinder, as indicated in the component structure of the Details Panel. Notice the spotlight became a child of the cylinder's static mesh component. Conversely, the cylinder's static mesh became the spotlight's parent component. This is an important idea. Now that the light became a child component, moving the parent, i.e. the cylinder entails that the child, i.e. the light, moves with it. It became attached.

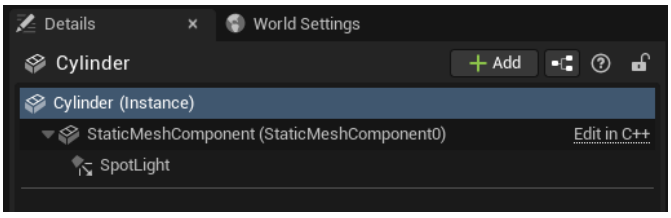

Figure 3.16: Cylinder with Spotlight Component

To add a component from the Content Drawer, click Add, add a generic Static Mesh; then in the Details Panel, replace the mesh as shown earlier.

3. Rotate the Spotlight, so it faces the "front" side of the flashlight.
4. Rename cylinder and spotlight to reflect its purpose.

3.7.2 Adding Non-Actor Components

Adding a non-actor component like a rotational movement to a mesh, like a cube, follows the same procedure.

E.g. to add a rotational movement to a cube, select the cube, set its Mobility to Movable, click Add in the components section of the Details Panel, select RotatingMovmenent. If the level is played, the cube will rotate.

With the movement selected, a number of properties regarding the movement can be set (see figure 3.17) .

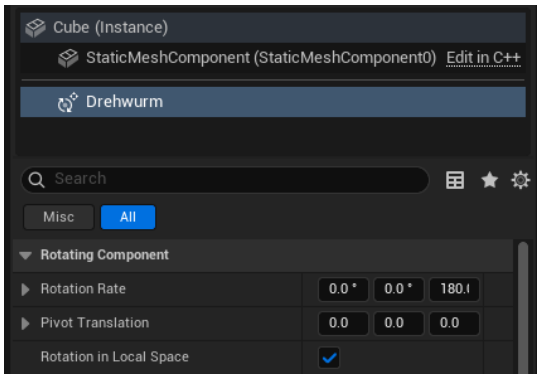

Figure 3.17: Rotational Movement Properties

Obviously, the rotation rate settings determine the rate of rotation around the 3 axes. Notice the Pivot Translation section. This can be used to set the pivot around the object is meant to rotate. If the pivot is set outside of the mesh, the object will revolve around this pivot point.

3.8 Volumes

In the *Unreal Engine*, a Volume is a 3D area of space that is invisible to the player and serves a specific purpose depending on its type. A variety of Volumes is available in the Volumes category in the Create menu.

Notice, that Volumes are brushes, therefore there are respective settings in each Volume's properties panel.

3.8.1 Volume Types

Find a list of noteworthy Volumes below:

- **Blocking Volume** Prevents actors from being able to enter that Volume.
- **Camera Blocking Volume** Just like a Blocking Volume but is meant to block only the Camera. This is useful in third-person games when the camera is supposed to keep out of certain parts of a level.
- **Trigger Volume** Probably the most important type of volume. They are used to trigger events when an actor enters or exits them. These events fire and can be handled using Blueprints or the *C++* programming language.
- **Pain Causing Volume** Causes damage to an actor. Notice, that an actor's Pain Causing property needs to be active for this to work. Also notice the settings Damage Per Second, Entry Pain as well as Pain Interval. These should be largely self-explanatory.
- **Kill Volume** Destroys actors entering.
- **Physics Volume** Allows for changes in phyics within the volume. E.g. one could define the Terminal Velocity, various Priority settings, Fluid Friction, etc.
- **PostProcessVolume** This volume is used to tweak the rendering settings for just certain areas of a level.

Blueprints

4.1 Introduction

This chapter provides an introduction to the concept of Blueprints. Basically, Blueprints kind of are the core of defines the *Unreal Engine*.

They are the engine's scripting an programming framework, designed to be extremely flexible, allowing for near unlimited creativity when defining gameplay and game mechanics. This is true even for non-coders, even though understanding programming concepts like object orientation and inheritance help in understanding the underlying concepts. This is because under the hood, Blueprints are nothing but a sophisticated UI wrapped around a *C++* framework. But again, programming skills are not required.

So far, this document dealt with game environments and the object that live within this environment. That being said, without those objects interacting, games would turn out rather boring.

Blueprints are assets containing data and instruction; this is the element to deal with game concepts like player health, energy, scores and any other data structure the game mechanics required.

Therefore, Blueprints are what enforces the game logic.

There are two main types of Blueprints - The Level Blueprint and Blueprint Classes.

4.1.1 The Level Blueprint

A Level Blueprint is used to hold data and instructions used to affect the mechanics of a particular level. This may include elapsed level time, the number of assets a player collected, and so on.

This may include scripted events, like, for example, enemies spawned when a player character reaches a particular area of the level. I.e. anything that affects the whole level, regardless of player characters.

Also, events occurring just once inside a level might be handled best in the Level Blueprint.

Ending Game

In this tutorial it is shown how to use the Level Blueprint to end the game after 2 seconds.

- Open Toolbar > Blueprints > Open Level Blueprint
- Inside the Level Blueprint Editor, identify the Event Graph, the area of a Blueprint where the logic is scripted, using the UI.

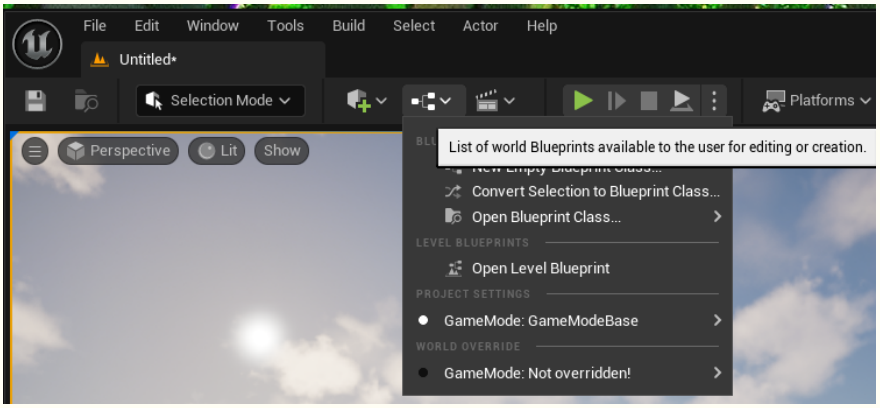

Figure 4.1: Blueprints Menu

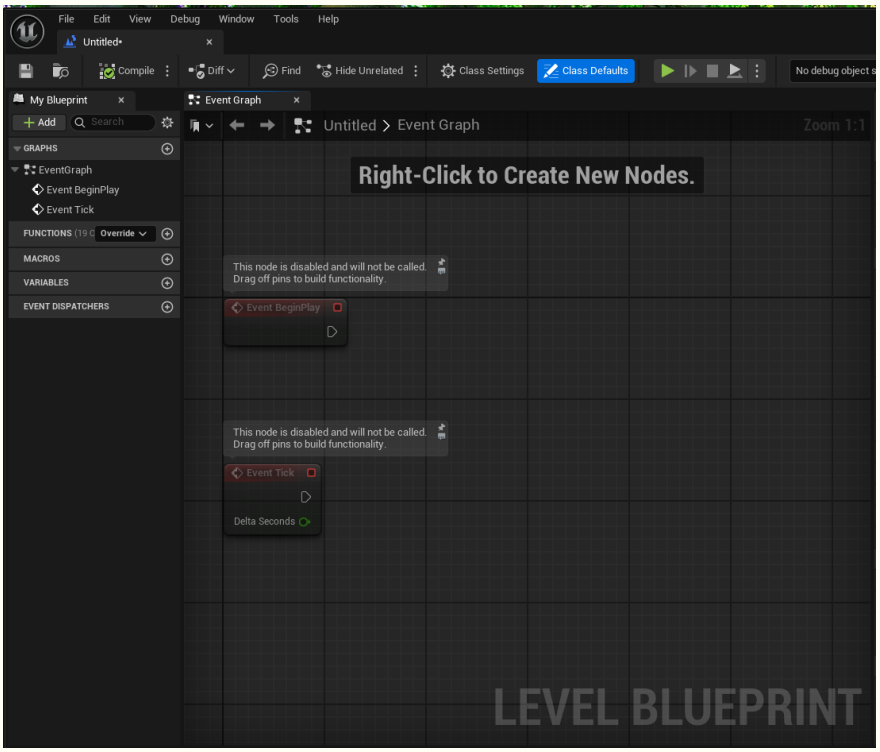

Figure 4.2: Level Blueprint Editor's Event Graph

By default, the Level Blueprint starts off with two nodes in the graph. These are common Nodes to start off with, which is why they're here by default. They are disabled at the start, so they are transparent, but they can be used right away by connecting them to another node.

The first node is the Event BeginPlay node. It will be activated by level start.

Ending Game

The top strip of an Event node will be the color red and it features an icon depicting an arrow inside of a diamond symbol. The second also an event node. The Event Tick node is a node that is activated on every tick of gameplay. Before every frame of the game is drawn on the screen, any logic connected to the Event Tick node will be executed.

This is useful, for example, in situations where certain conditions needs to be checked constantly, therefore when met will have an immediate effect on the game.

It will be shown how it is possible to end the game; however, in oder to make this visible, a short delay of 3 seconds will be added; i.e. once the game starts, the StartGame event fires, calling the node in the Blueprint, which in turn will wait for 3 seconds before ending the game.
Unsurprisingly, to add this delay, a Delay Node will be used.
So to cause a delay in the execution of the logic, a Delay node will be used. One way to create a new node is to Right-Click in any empty space in the graph, then select the node from the menu.

There is a whole plethora of nodes available to choose from, organized into categories; of course, there is a search box available to shorten things.

The Delay Node is a Function Node. Function Nodes are light blue and have an icon of a lowercase *"f"*. In Blueprints, a function is a node that performs a specific task, basically it reflects the concept of methods from object oriented programming. Obviously, the task of the Delay function is to wait for a specified amount of seconds before passing execution on to the next node.

The icons o the edges of nodes are called Pins. Pins on the left side of a node are input pins and pins on the right side of a node are output pins. Pins with these white icons looking like a Play button are execution pins.

Clicking on a node and mouse-dragging off it creates a "wire", which can be connected to the input pin of another node, denoting a connection.

- Right-Click onto an empty spot in the Level Blueprint Editor, enter *"delay"*, select the respective node.

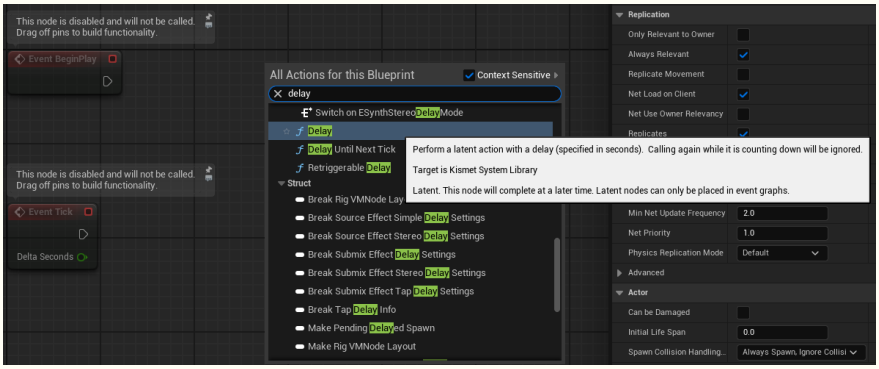

Figure 4.3: Add Delay Node

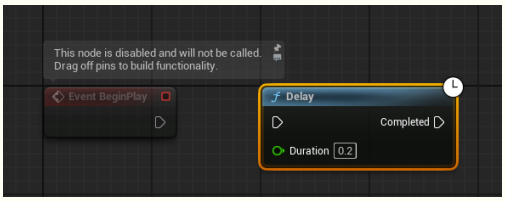

Figure 4.4: Delay Node Added

Ending Game

Now, the delay duration could be specified by entering the respective number in the node, or by connecting an different node that outputs a number. For now, the duration (3 seconds) will be hard coded inside the node. Just for clarification, notice that the delay node does **NOT** cause the game to pause. It just delays the execution of the execution path it is integrated with.

- Click And Mouse-Drag to connect from the Event BeginPlay node to the Delay node; connect to the Delay Node Input Pin.
- Click And Mouse-Drag from the Delay Node Output Pin onto the Blueprint. Let go, which should open die add node dialogue as before when Right-Clicking. Search for quit, then select Quit Game.

  Notice, that the Quit Game node has some additional input pins; these are used in multiplayer game scenarios and can be used, for example, to quit the game of a particular player, e.g. when his character has been "killed".

  There is also this Quit Preference setting where different behavior may be selected: Quit to have the application actually close as well as Background to make the application not actually close but just run in the background instead.

  Finally, Ignore Platform Restrictions provides an option to override any restrictions relating to quitting games that might exist on the platform the game is running on.

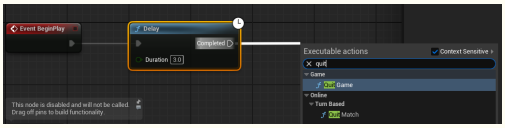

Figure 4.5: Connect with Quit Game

.

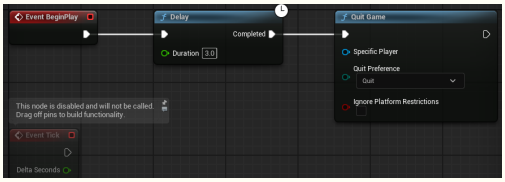

Figure 4.6: Quit Blueprint

- Click Toolbar 〉 Compile, save, exit the Blueprint.
- Run the game. It should sit idle for 3 seconds, then quit.

Finally, one note on Comments. Comments are a way of summarizing the functionality of Blueprints.

- Comments can be created by selecting the affected Blueprint, and either Right-Click 〉 Create Comment From Selection or pressing C.

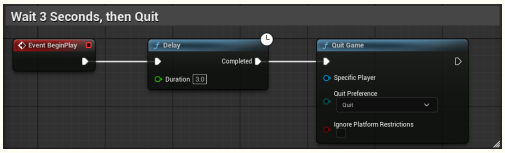

Figure 4.7: Comment Added

4.1.2 Blueprint Classes

Blueprint Classes are a way to turn any actor or asset into a Blueprint, allowing for objects with custom behavior and/or data. If this sounds familiar, there is a reason for that.

This does not only smell like *Object Oriented Programming* and/or *Inheritance*, it actually is. As mentioned above, Blueprints are just wrappers around a *C++* framework. And, that actually shall be taken literally: Under the hood, when manipulating objects using the Blueprint Editor, *C++* code is generated; which is the reason why there is the requirement to *compile blueprints*: the generated *C++* code must be compiled before it can be run. Therefore, since the system is following *Object Oriented Programming* concepts, it seem quite natural that the Blueprint system works exactly the same, which is a good thing, as this concepts have been successful for a reason.

Plus these ideas are simple to understand as well as rather useful when it comes to game development as well.

Case Study: Haunted House

If we were to build a haunted house, including a chair that mysteriously floats up and down, like it were possessed.

In this scenario, the player cold try to shoot the chair with a gun, destroying the chair if hit with a sufficient number of projectiles. This could be achieved using Blueprints. More concretely, it could be achieved by creating a Blueprint Class out of the chair mesh.

Within this Blueprint, the chair movement could be specified, along with the chair's health (default value of 100). Each time the chair is hit, a predetermined value might be subtracted (say, 10); in case the health value deteriorates below 0, the chair shall be destroyed.

One benefit of this system, being derived from object orientation, is that once a Blueprint Class has been created, an arbitrary number of instances of this class can be instantiated; exactly like in object oriented programming. Keep in mind that after all, Blueprint Classes are nothing but *C++* classes, derived from a superclass (that would be, of course, the generic Blueprint Class).

Concretely, once the Blueprint is completed, it is available in the Content Drawer; so an instance of it can be dragged into the level as often as required. Each time this happens, a new instance of the haunted chair is created inside the level.

Each of the chairs would float up and down, starting from its initial position. Of course, each chair would possess its own health variable.

4.2 Variables

Just like in regular programming, *Variables* are used to store data. However, *Unreal* comes with its own set of data types (see figure 4.8).

Figure 4.8: Unreal Data Types

Variables can be created within Variables sub-tab located in the right hand tab of the Blueprint Editor.

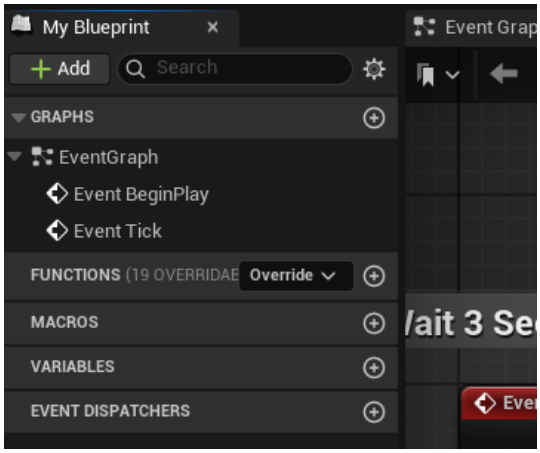

Figure 4.9: Variables

Clicking on the plus sign on that tab will create a new variable which can then be named as well as its data type selected.

Speaking of data types. *Unreal Engine* comes with a significant number of data types; unfortunately, discussing those is beyond the scope of this document.

Most of the data types resemble data types as known from programming languages lice *C, C++, Java or C#*. A few of them, though, are rather unique or worth mentioning for other reasons:

- **Byte**: Takes the least amount of memory, therefore very efficient. Can be use to store a single number ranging from 0 to 255.
- **Text**: Used for, well, text. Its noteworthy because it features a localization feature. Use this if the values it holds may be translated into a variety of different languages.
- **Vector**: This data type is used to store three float values. So this is useful for defining points in space, RTG or other color values, or, frankly, or anything that is defined with three values.
- **Rotator**: Stores numbers that describe an object's rotation in 3D space.

- **Translator**: The Transform data type is used to hold data that describes an object's position, rotation, and scale in 3D space.

Ending Game Continued

Continuing the tutorial from the preceding section, the hard coded value denoting the delay duration is going to be replaced by a variable instead. The respective variable will be called *"DelayDuration"*. Its value is going to be of type Float, matching the Data type of Delay > DelayDuration. Now, the data type is color coded, where the color Green denotes Float. That said, there is no need to memorize all the data type colors; simply hovering over a variable will bring up information about the variable (see figure 4.10).

To use the variable, drag it into the Blueprint Graph, what brings up a menu asking if a getter or setter should be used.

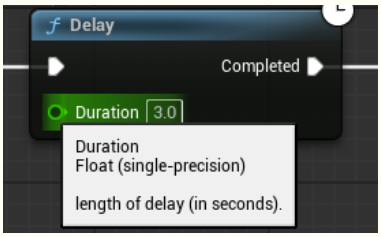

Figure 4.10: Additional Variable Information

- Blueprint Editor > Right Tab add a new variable; name it DelayDuration. Set its type to Float.
- Drag the variable into the Blueprint graph, let go the LMB, select getDelayDuration.
- Drag from the DelayDuration > Output Pin and connect the wire to the Delay Node > Duration Pin

Notice, that at this point, the variable has not been initialized, not even a proper initialization value has been set. To set this, with the variable selected, it is possible to set a default value in the respective Details Panel. Doing so will reveal that in order to set a default value the Blueprint must be compile first. Which makes sense, because the variable is not known to the Blueprints system prior to compilation. Therefore, the Toolbar > Compile button is used to compile the Blueprint. Now, the default value can be set; its default in the Details Panel is 0.0.

- Select the Delay Duration variable. With it selected, head over to the Details Panel.
- Notice under the Default Value category, the system asks for compilation.
- Click Toolbar > Compile.
- Set the Default Value > Delay Duration value to 3.0;.
- Toolbar > Compile again.
- Play. Notice, that the game behavior did not change at all. However, the Delay node's Delay Duration now is set by a variable instead of being hard coded.

At this point, the delay duration is set by reading a default value from the variable. To set this duration after the game started, a respective setter can be used, this time setting the delay duration to 6 seconds:

- Drag from the Delay Duration variable and create a Get node.
- Connect the get node's output pin with the Duration input.
- Drag a Delay Duration set node into the graph. Set its Delay Duration to 6.

Ending Game Continued

- Right-Click the Event BeginPlay output node and select Break This Link.
- Connect Event BeginPlay with the Set node.
- Connect the Set set node with the Delay node.
- Compile and save.

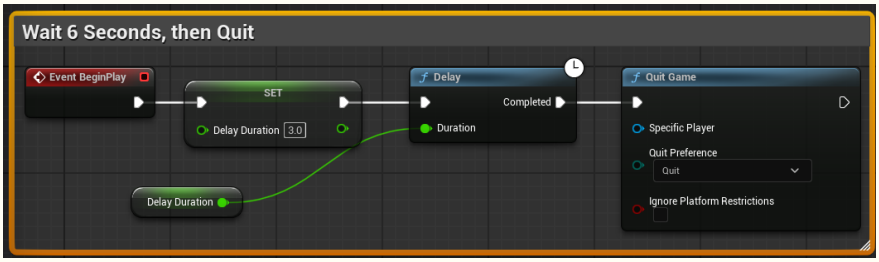

Figure 4.11: Blueprint: Wait 6 Seconds, then Quit

The default value of Delay Duration is still 3.0, but before the execution flow approaches the Delay node, the value of Delay Duration is getting changed to 6.0.

So far, so straightforward. Note some of the variable properties:

- **Description**: Obviously, used to describe what the variable is used for. Not so obvious is that once this is set (and compiled), this description will be used as tooltip when hovering over the variable.
- **Instance Editable**: If this is ticked, the variable appears in the Details Panel of the respective Blueprint class.
- **Category**: Used to group variables into categories. This is just used within the *Unreal Editor*. To create a new category, just write its name into the respective box; otherwise, use the drop-down to select the category.
- **Slider Range**: Allows to set a slider range.
- **Value Range**: Similarly, this allows to set the value range. Note, that the value range must include the slider range.
- **Replication and Replication Condition**: Used in multiplayer games; if Replication is set to Replicated, this means that the value is replicated over the network. The Replication Condition states under what conditions the value should be replicated.

4.3 Functions

In *Unreal Engine*, *Functions* are the equivalents of functions and methods used in regular programming languages. By extension, in *Unreal Engine*, a function is a specific node graph.

Welcome Message Function

In this tutorial it will be shown how to create a function that takes a name as input and outputs a welcome message including the given name.

- Go to the Level Blueprint > My Blueprint tab.
- Click + to add a function; name it *"WelcomeMessage"*.

  Notice, that whenever a new function is created, the editor spawns the node graph for this functions and adds its entry node. In the Details Panel for this function, input and output nodes can be added.

  For this function, a name - data type String - is required to be able to add it to the welcome message.

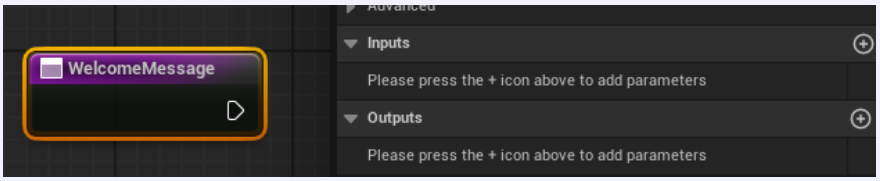

Figure 4.12: Function Details Panel - Add Input/Output

- Click Inputs > +, give it the name *"FirstName"* of type String.

  Notice, an input pin has been added to the blueprint.

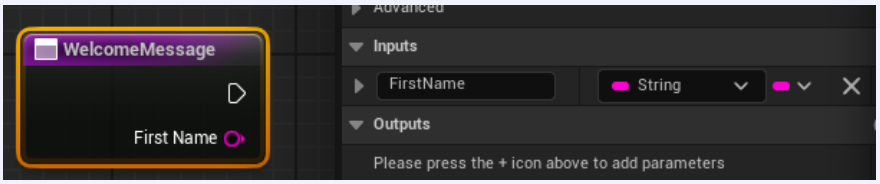

Figure 4.13: Input Pin Added to Function Node

- Click Outputs > +, give it the name *"Message"* of type String.

  This creates a Return Node, which is of course derived from the concept of return types in regular programming.

- Right-Click to create an Append node, which is used to concatenate two strings together. The Append node is of course itself a function. Be careful though when selecting the node: typing Append into the search bar will yield multiple instances of an append node, as functions going by this name are used in a range of classes. So, make sure to look for the String > Append function.
- In the A Node, type *"Hello, "*.
- Connect Welcome Message > First Name, to the B Node.
- Click Add pin
- Type *", Welcome to Game Development!"*

Welcome Message Function

- Connect Append » Return Value to Return Node » Message

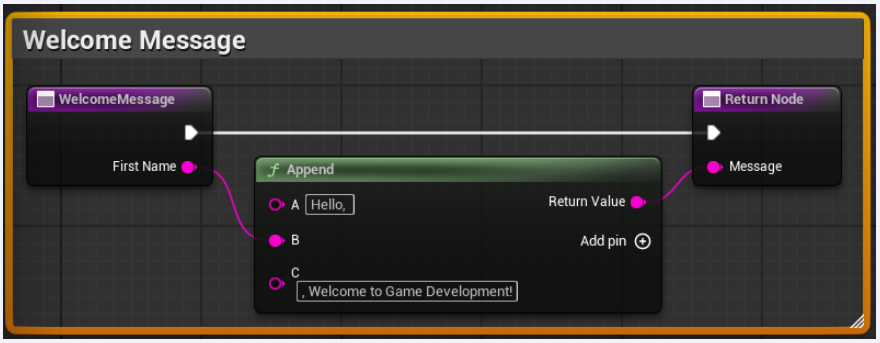

Figure 4.14: Welcome Message Function

- Compile, then save the blueprint.

  The function is now complete; it can now be called.

- Go back to the Event Graph.

  Notice that the Welcome Message function can be found in the My Blueprint tab to the right. Drag it into the Event Graph. Alternatively, it can by found by Right-Click, then search for *"WelcomeMessage"*.

- Connect the Event BeginPlay node to the Welcome Message.
- From the Welcome Message » Output Pin drag a wire onto a free spot in the graph. Select a Print String node.
- Hard code *"Mustafa Mustermann"* into Welcome Message » First Name
- Connect Welcome Message » Message to Print String » In String
- Connect Print String » Output Pin to Set » Input Pin

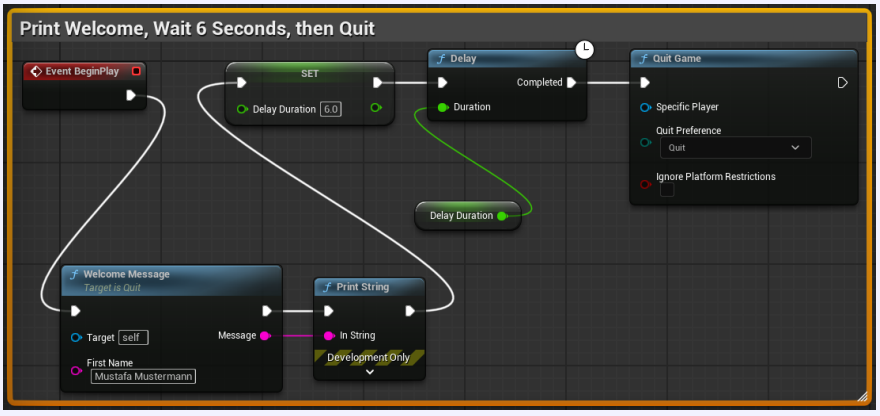

Figure 4.15: Welcome, Wait and Quit Blueprint

- Compile, then run the game

Now, as soon as the game starts, it is going to print *"Hello, Mustafa Mustermann, Welcome to Game Development!"* on the screen.

Notice the Details Panel, it features properties similar to variables, Like Description, also appearing as tooltips, or Categories.

Also note the Keywords property; this is used when searching functions. Adding keywords may increase visibility when searching.

4.4 Flow Control

Flow control functions can be found in the Node Menu » Utility » Flow Control. Most of the functions here actually do have their respective twin as know from conventional programming.

Noteworthy flow control node are:

- **Branch** node: The Branch node takes in a boolean value as its input and then continues execution either through the True output pin if the boolean value is true, or through the False output pin if the boolean value is false. This is equivalent to the *if statement*.
- **Until Next Tick**: Twill cause a delay, but for a duration of just a single tick of gameplay.
- **ForLoop**: Equivalent to *For Loops*
- **ForLoopWithBreak**: Similar to *ForLoop*, except that it is possible to break the loop before it is finished.
- **Do N** node; abbreviation for *Do N Times*. Like *ForLoop*. However, this blocks execution once the N occurrences have been exceeded. For example, an event should happen every time a particular key is pressed. This works as long as the number of key strokes did not exceed N; thereafter, key strokes will be ignored. However, this can be reset using the Reset Pin
- **DoOnce**: Basically, this is a *Do N* node, where $N = 1$: The only difference is that his node can start off deactivated; i.e. it must be reset to become active.
- **DoOnce MultiInput**: Like DoOnce, just for an array of input / output pairs. If reset, ALL of the pairs become active again.
- **WhileLoop**: Equivalent to *While Loops* known from classical programming.
- **FlipFlop**: alternates between having execution flow out of the A pin or the B pin every time the node is activated.
- **Gate**: Can be set to Opened or Closed. When the gate is open, execution flow entering the enter pin will flow out of the exit pin. Conversely, when the gate is closed, any execution flow entering the node will stop there, and the exit pin will not fire. Any time execution flows into the open pin, it will open the gate and vice versa with the close pin. Toggle will do just that, it toggles the gate open/closed.
- **MultiGate**: Execution enters a single execution input pin, but it will flow out of only one of the execution output pins. Output will not flow out of ALL pins, rather, just one pin at a time, depending on the options chosen.

4.5 Accessing Actors From The Level Blueprint

In order to make good use of the Level Blueprint, it is required to access actors from within it, reading data, making decisions, manipulating actors or spawning events.

Accessing Actors From Level Blueprint

In order to get access to an actor within the Level Blueprint, that actor needs to be selected in the Level Editor when Right-Click in the Event Graph of the blueprint.

- Left-Click the chair to select it
- Right-Click on an empty spot in the Event Graph, bringing up the events menu.

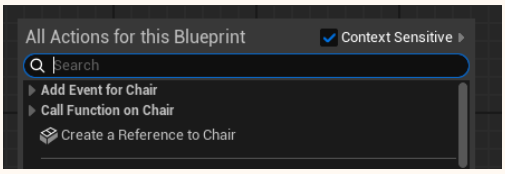

Figure 4.16: Content Sensitive Options For Selected Actor

Notice the options related to the chair on top of the menu. Here, events based on that actor may be called, functions may be invoked or a reference to the actor may be obtained. Note that for this to work, the Context Sensitive checkbox needs to be checked (see figure 4.16). Also note, that this is possible in the Level Blueprint only. This way, it is possible to access an actors properties. E.g. an actor's visibility could be changed, making it disappear.

- Right-Click to bring up the menu and search for *"Set Visibility"*. Notice, that context sensitivity must be deselected.
- Choose Rendering > Set Visibility

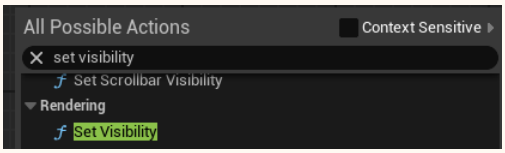

Figure 4.17: Choose Rendering > Visibility

- Connect Chair > Output Pin to Set Visibility > Target Pin. Note, that since visibility is a property of components, not the actors themselves, so this node is just getting the Root Component of the Static Mesh Actor, which is the Static Mesh component, so that that can be passed in as the target component for this node.

  The Set Visibility > New Visibility variable determines if the component is visible; so if left unchecked, this means the component is *not* visible.
- Connect Event BeginPlay to a Delay node, set its delay duration to 3
- Connect Delay > Completed Pin to Set Visibility > Input Pin

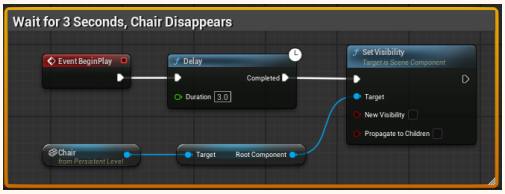

Figure 4.18: Chair Disappears Blueprint

Trigger Box Blueprint

This tutorial shows how to spawn events from actors. In particular, using a `TriggerVolume`, events are created whenever an actor starts or ends overlapping with the volume. The chair disappears when the actor enters the level and re-appears when the actor leaves the volume.

- Drag a `Create` > `Volumes` > `Trigger Volume` into the level. Name it *"Chair Trigger"*.
- With `Chair Trigger` selected, open the `Level Blueprint`.
- `Right-Click` to open the menu, navigate to `Add Event for Chair Trigger` > `Collisions` > `Add On Actor Begin Overlap`.
- Repeat with `Add Event for Chair Trigger` > `Collisions` > `Add On Actor End Overlap`.

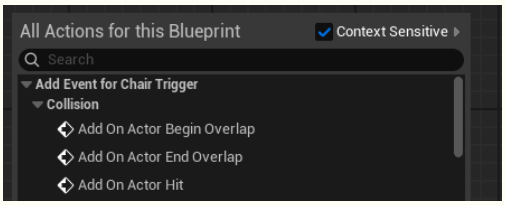

Figure 4.19: Adding Overlap Events

This way, there are not two events, one for entering the volume, one for exiting it.

- Connect `On Actor Begin Overlap (Chair Trigger)` to `Set Visibility` > `Input Pin`
- `Copy and Paste`
- Connect `On Actor End Overlap (Chair Trigger)` to the copied over `Set Visibility` > `Input Pin`
- Tick the copied over `Set Visibility` > `New Visibility`

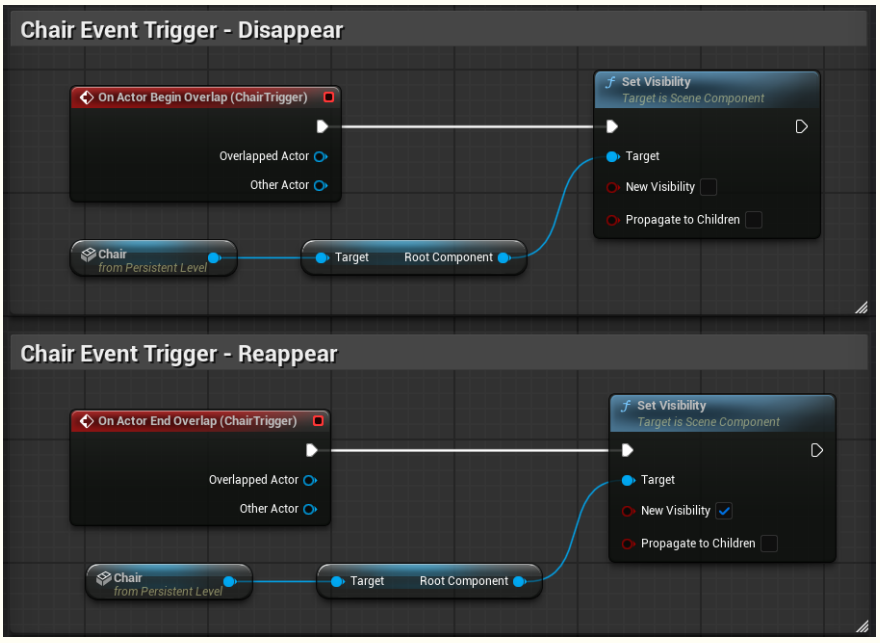

Figure 4.20: Trigger Box Blueprints

- `Compile`, save and run. Now, the chair disappears once the actor enters the volume just to reappear when leaving it.

4.6 Timelines

Timelines are used to create simple animations, such as changing the location, rotation or color of an object. Timelines define the position on a graph given a time period in order to define at which spot between two endpoints a value should be. For example, to define movement, a Timeline could specify that after 3 seconds the object should have move 6 meters in X-direction.

Timelines can be added by choosing it at the bottom of the nodes menu. Timelines can be named, of course.

Timeline

In this tutorial it is shown, how to adjust the intensity of a light source. So first, a Float Track is added, since the intensity is set using float value.

- Open the Level Blueprint.
- Add a Timeline by choosing it from the bottom of the nodes menu. Name it *"LightIntensityTimeline"*.
- Double-Click the timeline to open the Timeline Editor
- Add a new track by clicking + Track
- Select Add Float Track. This allows for adding tracks based on single float values. Light intensity IS a float value, hence this is chosen here.
- Rename the track to *"IntensityValue"*

A track is represented by a graph. This graph is defined by values occurring at specific times. For example, a value might be 0 at the start, 5 after 3 seconds and back to 0 after 6 seconds. The value is represented along the Y-axis, whereas time is represented at the X-axis. Points at a track are represented by so-called Keys.

A Key represents the time/value pair.

Timeline

- Hold down Shift and Right-Click in the graph. To change the placement of this key, either Left-Click it and drag it to where it shall be or use the boxes at the top to enter the X and Y values manually.

  Notice: To navigate the graph, Right-Click and Pan the mouse; to zoom in or out use the Mouse Scroll Wheel.
- Enter 0 for both time and value, since the light intensity shall start dark.
- Add another key, set time/value to 1/0 ← so the light will stay dark for one second.
- Add another key, set time/value to 3/1 ←after 3 seconds, the light will be fully on.

Notice: If a key is not visible because it moved off the graph's scale, click the ↔ or ↕ icons to the left of the time/value boxes.

Notice 2: *Unreal Engine* measures the Intensity of the Point Light in candelas. At this point, the value for light intensity could be set to 200 to reflect 200 candelas. That being said, this is not good programming style. Rather, we set the value representing a MAXIMUM value, which in this case is 1.0. Later in the blueprint, this value is multiplied by the amplitude of the light, that is, in the is example, 200 (candelas). This way, the Timeline can be reused for arbitrary objects.

Timeline

- Add another key, set time/value to 5/0 ←after 5 seconds, the light will back off again.
- Check the Track Length is set to 5.

At the start of the animation, the Timeline node will output zero and it will continue to output zero for the first second. Thereafter, for the next two seconds, it will gradually output a higher and higher value, until at 3 seconds into the animation, it is outputting a value of 1. Then for the next 2 seconds, this value gradually decreases, until it reaches zero again.

Back in the Event Graph » Timeline node, the Light Intensity Pin value represents the value as given by the Timeline.

For each track created, it will create a new output pin on the Timeline node representing the respective Timeline value. This pin will be connected to the light source's Set Intensity node.

Timeline

- In the Level Editor, look for the light source and select it.
- With the light source selected, back in the Level Blueprint create a reference to the light source (Note: Context Sensitive must be on).
- Using the Node Editor, add a Set Intensity node to the graph.

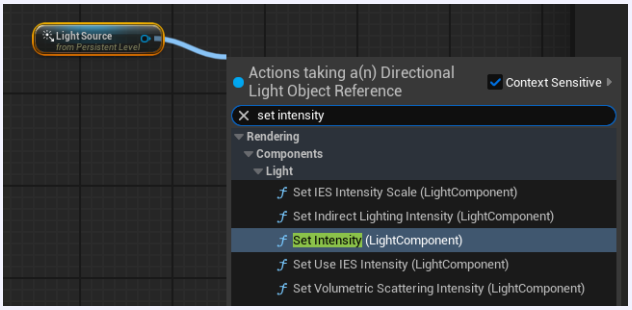

Figure 4.21: Set Light Intensity Node

- Connect the Light Source pin with the Set Intensity » Target Input Pin
- Connect the Timeline » Update pin with the Set Intensity » Execution Input Pin

  Notice: The Update pin will fire for every tick of gameplay while the Timeline is playing.
- Connect the Event BeginPlay pin with the Timeline » Play Input Pin
- Create a new Float variable MaximumLightIntensity. Give it a default value of 200; this reflects the maximum value of 200 candela.
- Create a get node for MaximumLightIntensity. Connect it to a Multiply node. Connect Timeline » Light Intensity with Multiply Input Node. Connect MaximumLightIntensity with Multiply Input Node. Connect Multiply » Output Pin with Set Intensity » New Intensity.

Now, when the game starts, the light will start dark, after a second it will increase to 200 candelas only to get back to dark again. It won't come up with light again. This is because the timeline does not loop by default.

Timeline

- In the upper part of the timeline editor, activate looping by clicking the Loop Icon.
- To make this appear more realistic, instead of changing the light intensity linearly,

This example shows how to manipulate *single float* values. However, this can be extended to vector values like *object locations*; this can be accomplished using Vector Tracks. By changing an objects location over time, simple movement animations can be created.

Similarly, Event Tracks can be created, which can be used to specify at what time events should be activated. Note, that there are also Color Tracks. Tracks can also be saved and retrieved later by selecting Create External Curve. This curve will be saved as an asset accessible via the Content Drawer.

4.6.1 Lerp Nodes

Lerp nodes are used to create timelines using Linear Interpolation, which is why those nodes are found int he Math category of the node menu. Since the examples discussed above, respective Lerp nodes could be found in the Math > Float category.

Lerp nodes allow - as the name suggests - for linear interpolation, i.e. interpolation values represent a percentage between two values. For example, if two values, say, X and Y, represent values from 0 to 1, 0.5 would represent 50%
This could be used, for example when using with colors, alpha values or progress bars.

This could also be used in the Light Intensity Example as discussed above. Instead of the Multiply Node, a lerp node seems appropriate:

Timeline

- Create a new float variable MinimumLightIntensity with a default value of 0.
- Add a getter for this variable to the blueprint.
- Delete the Multiply node. Add a Math > Float > Lerp node.
- Connect the Minimum Light Intensity with the Lerp > A Pin.
- Connect the Maximum Light Intensity with the Lerp > B Pin.
- Connect Timeline > Light Intensity with the Lerp > Alpha Pin.
- Connect the Lerp > Return Value Pin with Set Intensity > New Intensity .

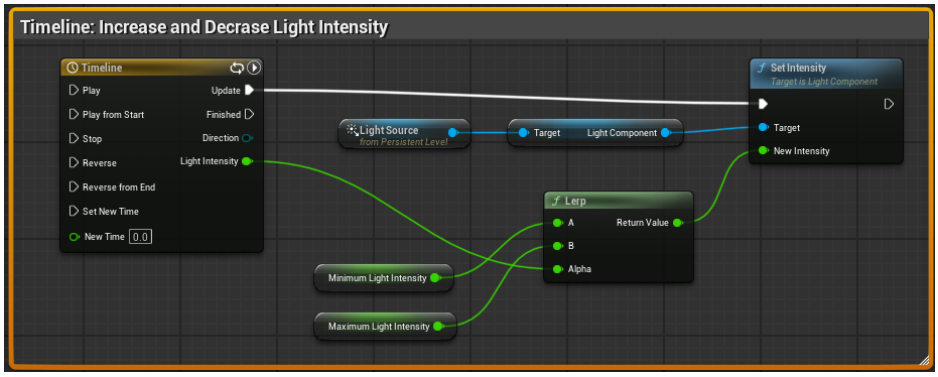

Figure 4.22: Increase and Decrease Light Intensity

4.7 Blueprint Classes

Blueprint Classes allow for creation of Blueprints out of existing actors and assets. Creating a Blueprint therefore allows to add properties and behaviors to that actor.

This should sound familiar: it basically (and, under the hood, quite literally, since the engine is implemented in *C++*) is an implementation of the concept of inheritance that should be familiar from object oriented programming.

In order to create a Blueprint from an actor, it needs to be selected first, then the Blueprints Icon may be clicked in the Details Panel.

Notice, that there are a number of differences between the Level Blueprint and Blueprint Classes. Level Blueprints have an Event Graph only. Blueprint Classes also feature a Viewport tab as well as a Construction Script tab.

4.8 Viewport Tab

The viewport tab allows to inspect the actor's look and allows to add components to it.

4.9 Construction Script

The Construction Script will be invoked as the actor gets created. Obviously, it is the equivalent of the concept of *Constructors*; it should be familiar from object orientated, again.

4.10 Event Graph

Finally, the Event Graph should already be familiar from the section on the Level Blueprint.

4.11 Working with Blueprint Classes

Blueprint Class from Point Light

In this short tutorial it will be shown how to work with blueprints to create custom behavior. In particular, this point light will continuously toggle on and off.

- Create a new empty level, add a floor and a point light.
- Click on the Blueprints Graph Icon in the Details Panel. Choose a suitable folder, select a meaningful name for the blueprint. This will automatically open the Blueprint Editor.
- In the Event Graph, Right-Click to bring up the nodes menu, search for the Toggle Visibility Function. Notice, that previously, in the chapter on the Level Blueprint, it was required to select the asset before it was possible to add a reference to that asset to the Blueprint. This time that step could be omitted; the reason is that now, since we are in the Blueprint of that particular asset, the engine can deduce that on its own, so selection of an object beforehand is superfluous. The system therefore adds a reference to the object automatically and connects it to the Target Pin.
- Add a Delay node to the Blueprint, set its duration to 2 seconds. This is because the light shall not toggle immediately, it should rather wait for a few seconds before it switches off.
- Connect the Even BeginPlay node to the Delay node.
- Connect the Toggle Visibilty Output Pin to the Delay Node Input Pin. This will cause a loop; so combined, the light will be toggle on and off every 2 seconds.

  Notice, that the just added wire is routed to its target in a peculiar way, that make the Blueprint somewhat hard to read. Luckily there is a way to correct this: By using Reroute Nodes, i.e. nodes designed to route those wires in a convenient way:

- Double-Click the wire, this will create a Reroute Node. Drag it as desired; if required, just add one or more reroute nodes.

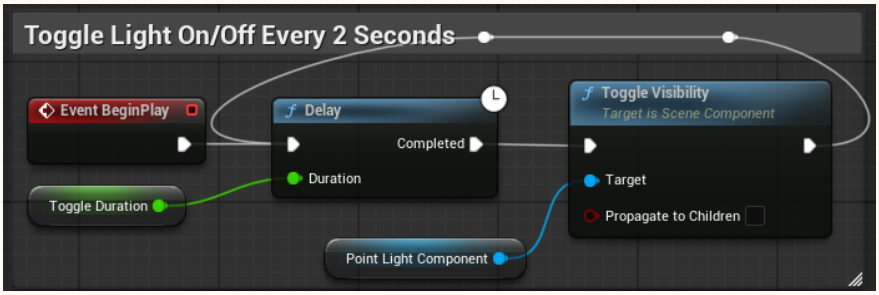

Figure 4.23: Toggle Lights Every 2 Seconds

Now, since this is a (Blueprint) *Class* on its own, *Instances* of it can be dragged into the level as desired.

That being said, all the lights at this point would be equal, apart from their location in the level, since there is no way for them to differentiate themselves from one another. For example, the toggle duration is the same for all instances.

Since the toggle duration might be a defining feature for each instance, it should be customizable, so that each instance does possess its own custom toggle duration. To accomplish this, a variable is added:

Blueprint Class from Point Light

- Add 2 more floors and custom point lights. Name them *"Left/Center/Right Light"*
- Add a `Variable` of type `Float`, name it *"ToggleDuration"*. Set its default value to 2.
- Make sure `Instance Editable` is ticked. Alternatively, this can be done using the eye symbol to the right of the variable.
- Add a getter to the new variable, connect its output pin to the `Delay Node Duration Pin`. Compile and save.
- For the `Left/Center/Right Light` custom point lights, set the `Details Panel` > `Toggle Duration` to 1, 2 and 3, respectively.

Run the game, notice that the lights each blink at their own pace.
To summarize, it has been shown how to create `Blueprints` off existing actors.

However, the more standard way of creating a `Blueprint Class` is to click `+ Add` in the `Content Browser` and select `Blueprint Class` from the menu. This will bring up a menu from where the parent class to inherit from can be selected. The top of the menu includes the most common classes to inherit from. As usual, a search bar aids in finding classes.

So, if the `PointLight` class is selected this way, it can be added and it will act just as discussed above.

Input

This chapter deals with the intrinsics of user input in the *Unreal Engine*.

Now, at first glance this might appear trivial. After all, the engine comes with a number of templates where user input is already set up - so, how hard can it be?

Now, how about support for game controllers? Are these supported as well by the template? And, what controller key, for example, is used to make a character jump? After all, game controller are very different, so how could there even be consensus on deciding what key?

Apparently, there is more to it than simple presses of a key.
This chapter discusses how to create playable characters controlled by using inputs like keyboard, mouse or gamepads.

5.1 Game Modes

A Game Mode is an actor that can be used to define and enforce the game's set of rules. These rules may include how many lives the player starts with, whether or not the game can be paused, if there are any time limits, the conditions needed to win the game, and so forth.

The Game Mode can be set on a per-level basis; of course, the same Game Mode can be used for multiple Levels.

Game Mode

In this very short tutorial it will be shown how to work with Game Modes.

- To create a new Blueprint Class from the Game Mode actor, go to the Content Browser and browse to the folder you want to put the Blueprint in.
- Click the + Add button and select Blueprint Class > Game Mode Base. Rename it to give it a meaningful name.
- Double Click the newly created Blueprint to open it in the blueprint editor.
- Click the notice saying *This is a... click open to add one.* This will open the Event Graph. Modify at will.
- Compile and save at will.

Note, that this is a regular blueprint; therefore, all the elements of regular blueprints are present, as expected. This also entails, that variable could created there that define the mode: for example a time period the game timer should start with. It could make use of the Event Graph to define how a player might win the game and what should happen when the do - or when the lose the game.

In the Details Panel there is a number of editable properties:

Game Mode

Now, the Game Mode blueprint has been created. However, the engine has not been configured to use it. This could be accomplished in a number of ways:

- Assign this as the default Game Mode; unless otherwise specified, all the levels in the game will use that. The respective option may be set in Menu Bar > Edit > Project Settings > Project > Maps & Modes.
- Setting the mode on a per level basis. With the respective level open, open Toobar > Settings > World Settings. This opens the Details Panel. The game mode for this particular level can be set in Game Mode > Game Mode Override. If this is set to None, the default mode will be used. Otherwise, the mode specified (using the dropdown menu) will be used.

5.2 Pawns

Within the *Unreal Engine*, a Pawn is an actor that can be controlled, either by a human player or the computer. The default game mode starts with a default Pawn Actor. This actor has been used in previous chapters.

This actor is somewhat invisible, since

1. It does not have a static mesh defined
2. The camera is attached in first person perspective
3. It is actually *Chuck Norris*

Pawn Actor

In this very short tutorial it will be shown how to create a Pawn Actor.

- To create a new Pawn Actor, go to the Content Drawer and browse to the folder you want to put the Pawn Actor in. Click the + Add button and select Blueprint Class > Pawn. Rename it to give it a meaningful name.
- Double Click the newly created Blueprint to open it in the blueprint editor.

The Pawn actor starts out consisting of just a single component, the DefaultSceneRoot component, as can be seen in the Blueprint Editor > Viewport Tab. To give the pawn some visibility, a static mesh will be added:

Pawn Actor

- Click the + Add button and select Static Mesh.

  This adds a Static Mesh component without specifying which static mesh shall be assigned. Therefore, to add a mesh:

- In the Details Panel > Static Mesh category, use the dropdown to select a static mesh, e.g. a chair mesh as provided by the Starter Content.
- Click the + Add button and select Camera.

  Wherever the camera is located, the pawn will see from this perspective. The camera could also be located somewhere above and behind the chair, which would give it a third person perspective. While this would work, usually a Spring Arm component would be added as well. The Spring Arm component will allow the camera to automatically make adjustments in cases where the line of sight between the camera and the mesh gets obscured.

- Click the + Add button and select Camera > Spring Arm. Attach the camera to the invisible spring arm.

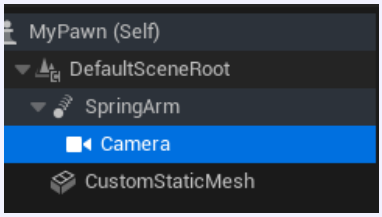

Figure 5.1: Camera on Spring Arm

- Notice the Spring Arm > Target Arm Length property, which will move the camera away or closer if modified.

  The Camera will be this far away from the mesh by default, but that could change if the Spring Arm needs to adjust because the view of the camera got blocked. For example, if a wall got between the camera and the chair, the spring arm would automatically shorten in order to bring the camera in close enough to be able to see the mesh again. Finally, when the wall was no longer an issue, the spring arm would lengthen back to the Target Arm Length.

- Open Game Mode Blueprint > Classes, set its Default Pawn Class property to this pawn.
- Compile, save and close the blueprint.

Now, if the game is played, the chair is visible in third person perspective from the camera just added. Notice, that since no movement controls have been configured so far, there is no means to control the chair at this point.

5.3 Characters

In the *Unreal Engine*, a Character is a type of Pawn, i.e. the Character class inherits from the Pawn class. In addition, it possesses some additional features, most notably they have bipedal movement.

This means, characters walk on two legs, loosely representing human movement. Having two legs also means that the character can walk or jump.

Character Actor

In this very short tutorial it will be shown how to create a Character Actor.

- To create a new Character Actor, go to the Content Drawer and browse to the folder you want to put the Character Actor in. Click the + Add button and select Blueprint Class > Character. Rename it to *"MyCharacter"*.
- Double Click the newly created Blueprint to open it in the blueprint editor.

The Character Class In comparison with the Pawn Class, the Character Class comes with a few different components:

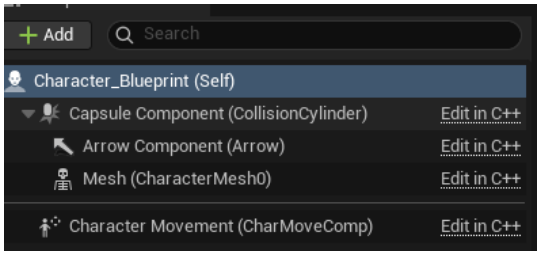

Figure 5.2: Components of the Character Class

Capsule Component	Used as the boundaries of the Character for the purposes of detecting collisions.
Arrow Component	Used to indicate which direction should be considered facing-forward for the Character.
Skeletal Mesh	Can be assigned a skeletal mesh to.
CharacterMovement	Probably the most important component. This is what gives the character the ability to move.

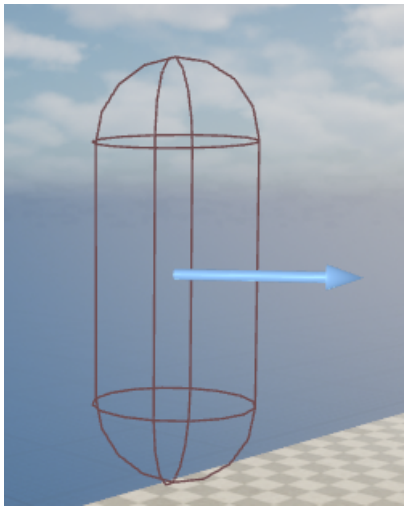

Figure 5.3: A Character Featuring Directional Arrow

As mentioned, the Character Movement category as mentioned above is a rather important one. Its more important properties (viewable in the Details Panel) are as follows:

- **Character Movement (General Settings)**
  - **Gravity Scale**: determines how much of an effect gravity will have on this Character.
  - **Max Acceleration**: determines the maximum acceleration this character can achieve for any type of physical movement.

- **Breaking Friction Factor**: determines how much the character will glide when attempting to slow down its speed. If Use `Separate Braking Friction` is unchecked, the `Breaking Friction Factor` will be multiplied by all other friction forces (ground, wind, etc.) to obtain the actual friction coefficient used. If `Use Separate Braking Friction` is checked, the `Braking Friction` property alone will be used as the friction coefficient, i.e. all other friction forces will be ignored.
- **Crouched Half Height**: This property is used to tell the engine where the middle of the character should be calculated at when the character is crouching. This is used for collision detection purposes.
- **Mass**: The mass assigned to the character.
- **Default Land Movement Mode and Default Water Movement Mode**: specify what kind of default movement the character should have when they are on land or in the water. These default to `Walking` and `Swimming`, respectively.

- **Character Movement (Walking Category)**
  - **Max Step Height**: determines how tall a step has to be before the Character can no longer automatically ascend it when walking.
  - **Walkable Floor Angle**: determines how steep a sloped floor can be before a Character can no longer walk up it.
  - **Several Properties relating to Jumping and Falling**: `Jump Z Velocity` determines how the character can jump. `Air Control property` determines how much control the player has over the character when in the air. Note, that this will be used in the `StairWayToHeaven Game Tutorial`.

Character Actor

The character so far is incomplete: it is lacking its ”eyes”, i.e. some means to view the level. Therefore, a camera will be added:

- In the `Components Window`, click the `+ Add` button, and add a camera. Place it roughly where the face of the character would be located. In addition, it will be shown how to add controls to the character. In particular, it will be shown how to assign the space bar as an input method to make the character jump.

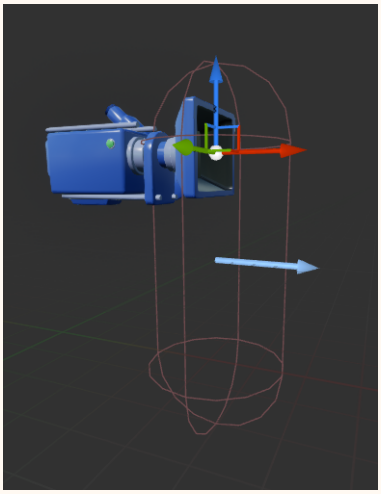

Figure 5.4: Character with Camera Attached

Character Actor

- Event Graph » Right-Click, search for *"Space Bar"*, select the Keyboard Events » Space Bar event. This will be called whenever the space bar is invoked.

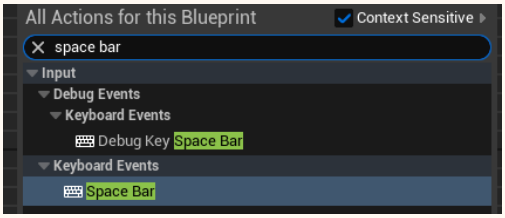

Figure 5.5: Space Bar Pressed Event

- This should fire when the space bar is pressed, so drag a wire out of the pressed pin, in the upcoming node menu search for "Jump".
- Compile and save.
- Create a Game Mode Blueprint, rename it to MyGameMode. Set its Default Pawn Class to MyCharacter
- Set Toolbar » Edit » Project Settings » Maps & Modes » Default Game Mode, To MyGameMode.
- Compile and save all.
- Run the game. It should present the level in first person view; and it should jump whenever the space bar is pressed; gravity should bring it back down.

  Finally, to make matters more exciting, the character is convinced to jump a little bit higher:

- Open MyCharacter Blueprint » Components » Character Movement » Character Movement: Jumping / Falling » Jump Z Velocity alter its value to 800. This will up the jump velocity significantly.
- Compile, save and run.

5.4 Controllers

A Controller, in the context of the Unreal Engine, is an actor used to possess a Pawn and control its movement and actions. There are two types of Controllers - the PlayerController, which is used to take input from a human and use that to control a Pawn, and the AIController which is used to implement AI control over a Pawn.

What is a controller used for? In the preceding section it has been shown that a pawn can be controller directly in its blueprint. However, in games it is not uncommon to have a plethora of pawns and characters. It would be rather tedious to have to program them all in their blueprints by hand, even more so considering that movements are mostly shared between characters.

Hence, the concept of PlayerControllers has been introduced. PlayerControllers are defined once and reused by any pawn the PlayerController possesses.

Character Actor

In this tutorial it will be shown how to define a PlayerController instance and assign it to a pawn. Also, the jump functionality created in the preceding section will be removed and reimplemented using PlayerController First, a new PlayerController blueprint will be created...

- Content Drawwer > + Add > Blueprint Class, select PlayerController as the parent class. Rename it to MyPlayerController.
- Remove the jump functionality nodes in the blueprint created in the preceding section.
- Open MyPlayerController in the blueprint editor.

  At this point, the space bar node should be added here, just as before. However, the jump function is a function provided by the character class. Therefore, in the Node Menu it will not be available right away.

  For this reason, a reference to the character is required in order to access its jump function. Therefore, ...
- Bring up the node menu and search for the Get Player Character function.

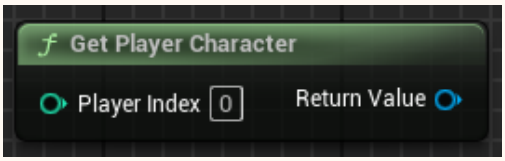

Figure 5.6: Get Player Character Node

  This function will return the character the specified player is using. I.e., player index 0 is for player 1 and player index 1 is for player 2 and so forth. If the Pawn that the player is using is a character, the return value will contain that character, otherwise this will return a Null value.
- Drag a wire out of Get Player Character > Return Value pin, search for *"Jump"*, add a Jump function.
- Search for *"Space Bar"*. Connect Space Bar > Pressed Pin to Jump > Input Pin

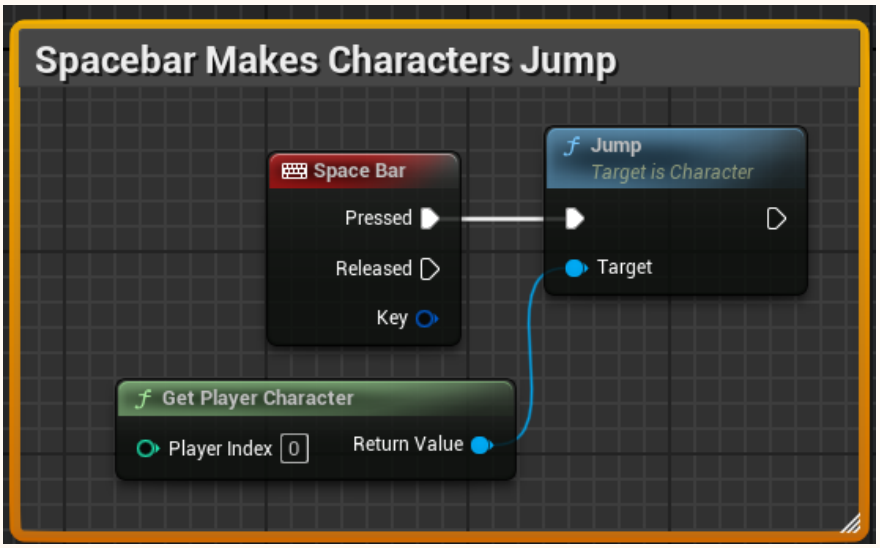

Figure 5.7: Jump On Space Bar Blueprint

Character Actor

- Compile and save the blueprint. The PlayerController Blueprint is completed so far. Now, the Game Mode must be told to use this new player controller, and
- Open Game Mode Blueprint » Player Controller and set it to MyPlayerController.

At this point, when running the game the character will jump as before, its just that this solution is much more flexible and, first and foremost, reusable.

Now, the PlayerController as configured in the game mode will be used by all characters by default.

5.5 The Enhanced Input System

In the tutorial discussed in Section 5.4 it was shown how to hook up a keyboard key that makes a character jump.

This section will show how to use Unreal Engine's Enhanced Input System. The enhanced system allows for more complex types of input, e.g. pressing multiple buttons or keys at the same time, in a particular order or hold a button for a specific time period, etc.

This system also allows to change the input context at runtime. This could, for example, happen when player enters a vehicle, what may require a very different input style. Using the enhanced input system, this scenario can be handles by swapping between a certain type of asset in blueprints.

Enhanced Input System

This tutorial demonstrates how to set up the Enhanced Input System, step by step, configuring Space Bar and a game controller key to make the character jump.

1. Define the input action the character performs. In this case, the character is supposed to jump: In Content Browser » + Add » Input » Input Action. Name it IA_Jump. Notice the IA_ prefix used. It is, in general, a good habit to prefix custom files/blueprints in such a way that the file itself makes clear WHAT it is.
2. Double-Click it to open.
3. Check out the Value Type property. Currently this is set to Digital, which is the default. Digital means there are only discrete, boolean values (rather than floating values, like the extent of a controller button that might be deflected in small increments). In this case, since a key press is boolean (i.e. it allows only for the values true or false), this is the correct setting. Save.

   Notice, the other possible value types: Axis1D/Axis2D/Axis3D could be used to provide floating values, which could be one/two/three dimensional, respectively. One dimensional input could stem from devices like a mouse wheel; two dimensional values usually stem from game controller coolie hat deflection or mouse, whereas three dimensional input may stem from things like Wii remote or VR input controllers.
4. In MyPlayerController delete the Space Bar node.
5. Open the node menu, search for the input action just created. This creates an input listener.
6. Connect EnhancedInputAction IA_Jump » Triggered pin with the Jump » Input pin.

Enhanced Input System

This tutorial demonstrates how to set up the Enhanced Input System, step by step, configuring Space Bar and a game controller key to make the character jump.

7. This would actually work, i.e. the action would trigger the character to jump. It's just that nowhere is specified what key will trigger IE_Jump. This is done using an Input Mapping Context, which is basically the thing that makes this system so flexible. An Input Mapping Context is an asset used to group Input Actions, essentially this is a context the player is in; i.e. walking, swimming or within a vehicle. The Input Mapping Context, as the name suggests, describes the rules how each input action should be triggered in that context.

8. Add Content Drawer > + Add > Input > Input Mapping Context, name it "IMC" and open it.

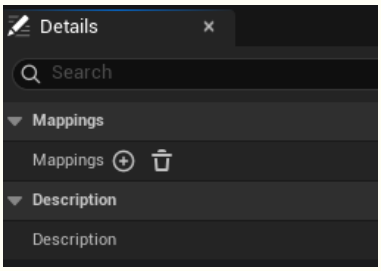

Figure 5.8: Add Mapping

9. Click + to add a mapping, i.e. mapping intputs to Input Actions. Open the Mappings triangle to the left.

10. Select IA_Jump from the drop-down menu.

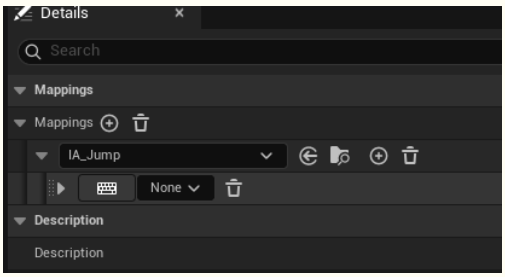

Figure 5.9: Keyboard Symbol

11. Again, open the next triangle (below). Notice the Keyboard icon. Klick it, then hit the keyboard key that is supposed to trigger the action; i.e. in this case, click the Keyboard icon, then hit Space Bar. Alternatively, the dropdown menu could be used.

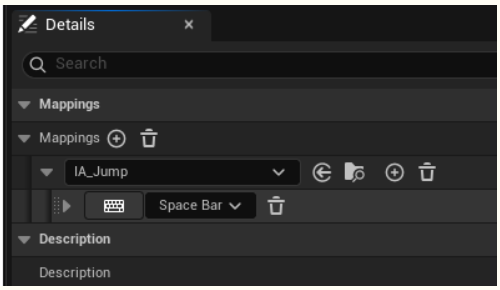

Figure 5.10: Space Bar Triggers

Enhanced Input System

At this point, an Input Action is configured, along with an Input Mapping Context (IMC) linking the Space Bar to the jump action. Finally, the Player Controller must be linked to the Input Mapping Context. This can be accomplished dynamically using blueprints.

The Player Controller could be linked to the IMC in the player controller blueprint. However, it usually makes more sense to do it in the character blueprint, since the contexts usually change based on the character's actions.

12. Open the MyCharacter blueprint.
13. Next to the Event BeginPlay node, add a Game > Player > Get Player Controller node.

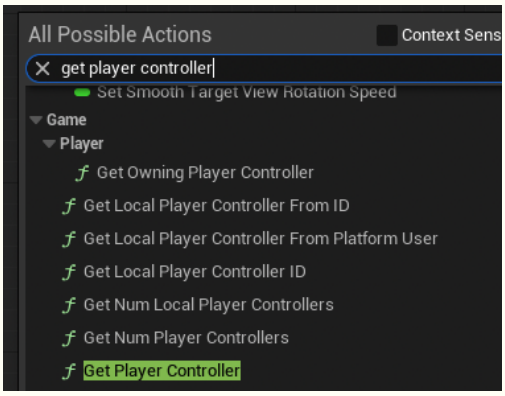

Figure 5.11: Add Game > Player > Get Player Controller

14. Drag a wire out of the Get Player Controller, search for *"getenhanced"*, find the GetEnhancedInputLocalPlayerSubsystem
15. Drag a wire out of that, search for *"addmapping"*, add an Add Mapping Context node.
16. Set Add Mapping Context > Mapping Context to IMC
17. Connect Event BeginPlay with Add Mapping Context > Input Pin

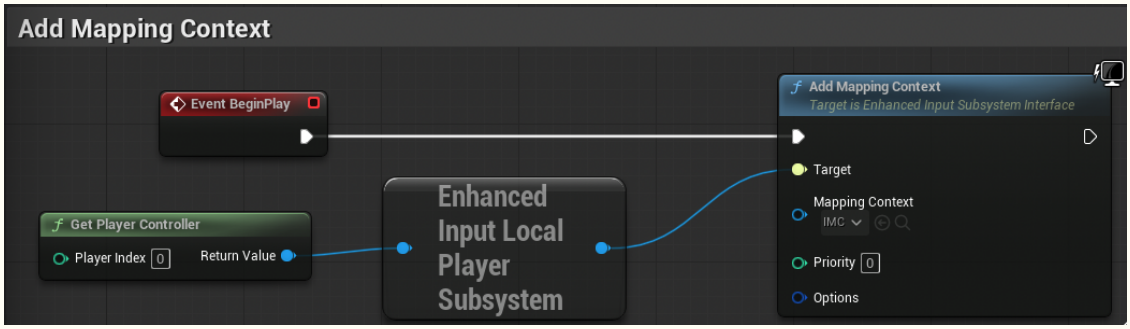

Figure 5.12: Add Mapping Context Blueprint

Enhanced Input System

As a last step, the `Player Controller` needs to be configured, since at present there is no code anywhere dealing with what should happen when the `Space Bar` has been triggered.

18. Open the `MyPlayerController` blueprint.
19. Open the create menu, search for *"IA_Jump"* and add the `EnhancedInputAction IA_Jump` node.
20. Open the create menu again, open a `Jump` function to the graph.
21. Open the create menu a third time and add a `Get Player Character` node.
22. Connect the `Get Player Character` `Return Value` pin to the `Jump` `Target` pin.
23. Connect the `EnhancedInputSystem IA_Jump` `Triggered` pin to the `Jump` `Input` pin.

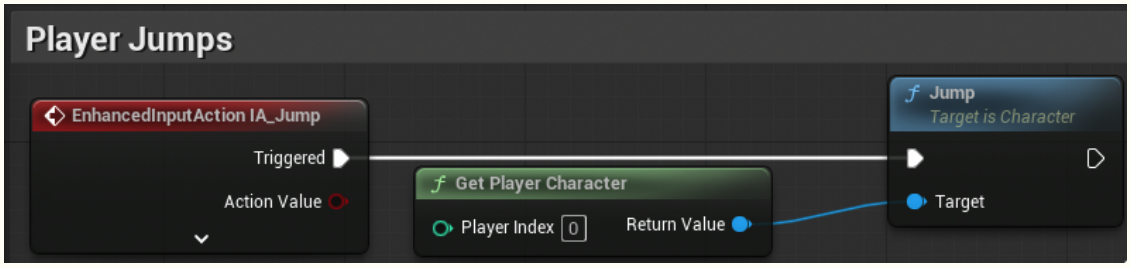

Figure 5.13: Character Jump Blueprint

24. Compile, save and close the blueprint.
25. Test the code. Hitting `Space Bar` should cause the character to jump.

Notice, that it may be a good habit to store the player character in a variable and call that, rather than calling `Get Player Character` each time. Consult the tutorial script to see how this may be accomplished. Also, notice figure 5.14 below for an overview on how the various controllers are interconnected.

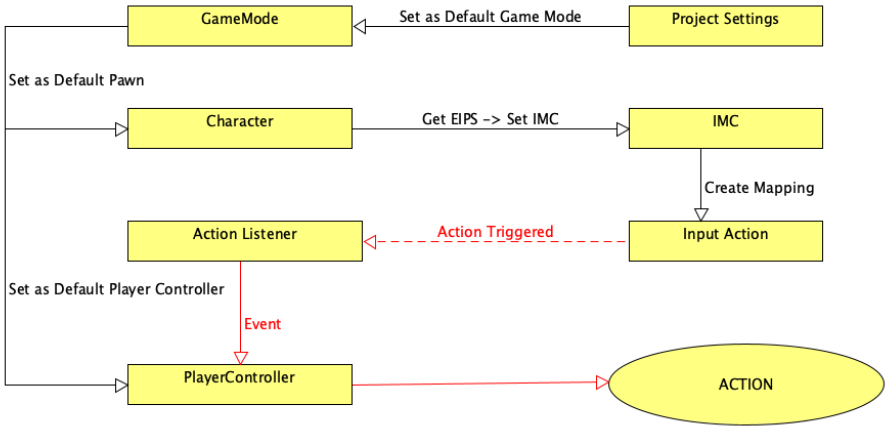

Figure 5.14: Enhanced Input System - Quick Map

5.6 Input Triggers

This section discusses Triggers as well as Trigger States, which can be used to develop more detailed rules for inputs. For example, requiring the player to hold a button for a given time period, a double tap, etc.

Consider MyPlayerController > Blueprint > EventGraph

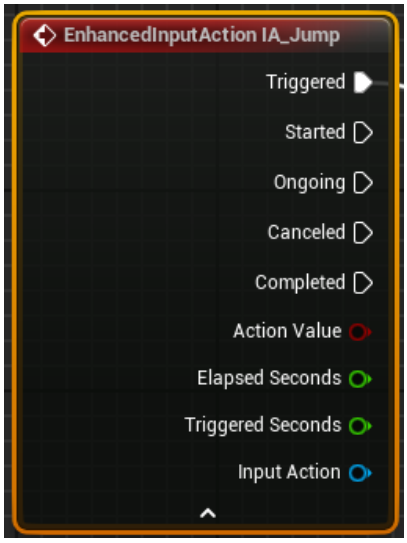

Figure 5.15: EnhancedInputAction IA_Jump Node

For now, the EnhancedInputAction IA_Jump > Triggered pin has been used. Apparently, this pin fires for every tick of gameplay whenever the respective input action is in the triggered state. In which case the EnhancedInputAction IA_Jump > Action Value will output True, else it will output False.

Each Input Action has one or more associated Triggers. If none is set explicitly, it will default to Down

Currently, the Space Bar is mapped to the jump action, therefore the following pins are going to fire if the Space Bar is being held down:

- **Started**, immediately followed by
- **Triggered**. Will fire for every tick of gameplay the key is held downl
- **Completed**. Fires as soon as the Space Bar is released.

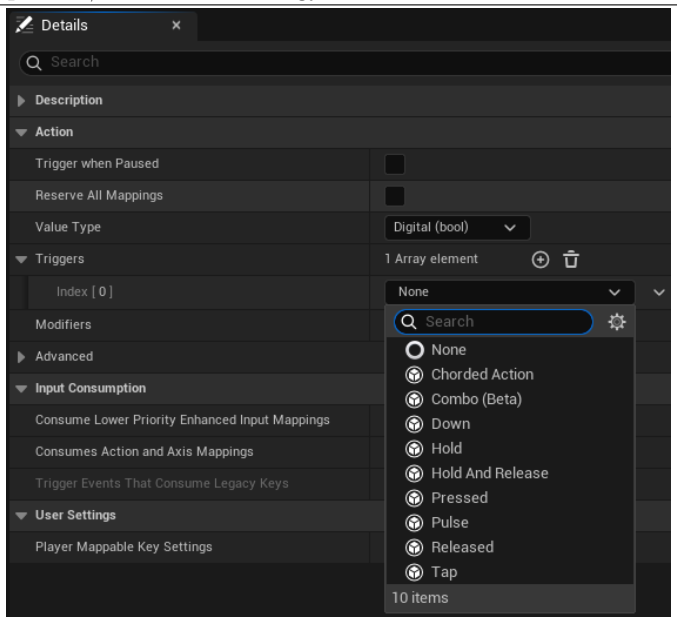

Figure 5.16: Action Input Triggers

As mentioned, this is the default trigger.

The actual trigger used can be set directly in the input action itself. If specified here, it will be applied to all uses of the action across all mapping contexts. To add a trigger, just click the + icon and add the trigger desired.

Notice the Hold Trigger. If the menu to the left is expanded, a number of properties can be set. Most noteworthy is the Time Threshold property, which states how long the key must be held down in order for the action to fire. In case the One Shot property is set, the Triggered pin will fire just once rather than for every tick of gameplay. If the key is released before the Time Threshold is reached, instead of the Triggered and Completed pins firing, the Canceled pin will fire instead.

Also noteworthy is the Actuation Threshold property. It is uses in connection with inputs like thumbsticks, where it specifies how far the thumbstick needs to be tilted before it is considered to be pressed, a feature to take care of wear and tear.

The Chorded Action Trigger is meant to work in situations where multiple multiple input actions are required to be triggered at at time.

Conversely, the Combo Trigger is uses in scenarios where a certain sequence of actions need to be performed in order to trigger the action.

Finally, hitherto all triggers have been defined in the input actions themselves. However, the triggers can also be defined in the Input Mapping Context. The difference is that any triggers defined in the Input Action will apply to all uses of that action across all contexts, whereas triggers defined in the Input Mapping Context will only apply to that particular context. Note, that if an input action has a trigger defined within, it is possible to see it in the mapping context under a read-only property named Triggers From Input Action.

5.7 Input Modifiers

This section continues the to discuss inputs by covering Input Modifiers.

Input Modifiers take the raw values that come directly from the inputs and modify those values in some way before passing them on to the Input Triggers.

As mentioned in the Enhanced Input 5.5 section, if the value type of the input action is set to **Digital**, the value produced by the input will be a boolean - either True to indicate that the input is currently pressed or False to indicate that it is not.

If the value type is **Axis1D**, the value will be a float - a single positive or negative value to indicate how far in either direction the input is currently pressed or zero if it is not being pressed at all.

Next, **Axis2D** is a Vector2D value, i.e. 2 float values - one for each axis.

Finally, **Axis3D** is a regular Vector value, consisting of three floats values.

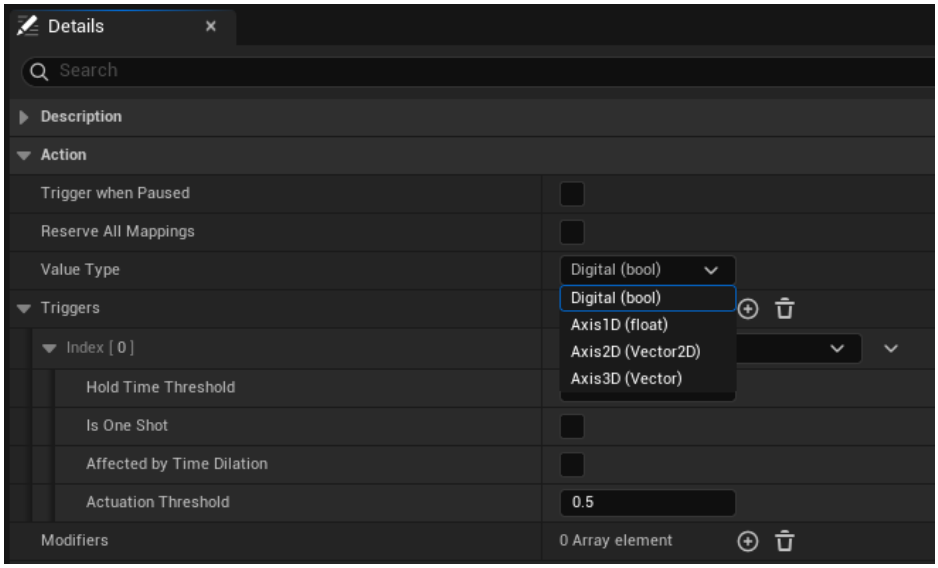

Figure 5.17: Input Modifiers

5.7.1 Dead Zones (Thumbsticks)

Imagine an Axis2D input is configured in the mapping context. Imagine further that a thumbstick is used to control a game character; for example, the thumbstick is fully deflected in several directions, say, right, left, up, down, and subsequently let to. No, depending on the age of the game controller and its quality in general, the thumbstick will NOT go back to center, i.e. outputting zero. This is because it is mechanically difficult to move back exactly to the center, causing the game controller to output values other than zero, even though it appears to be centered.

Input Modifiers can be used to mitigate this issue. A modifier can be configured in the Input Mapping Context. In theory, this could also be done in the Input Action blueprint; it is, however, advantageous to configure this in the mapping context, because this way it can be set for a particular input device only. E. g. to configure this for a thumbstick input, add a new modifier for the particular input device and select Dead Zone Modifier. Opening Input Device 〉 Dead Zone 〉 Index 〉 Lower Threshold / Upper Threshold is used to set the minimum value the input must be before it is passed on. Values less than the Lower Threshold will be ignored. Values below and at this threshold will be passed on as zero. From the lower threshold onwards, the modifier will gradually output larger values, up to the Upper Threshold. In other words, a Dead Zone Modifier takes the input values in the range between the lower and upper thresholds and remaps them to give values between 0 and 1.

Configuring it in the action would cause the modifier to be active at all times, for all input methods. If the developer were to add mouse navigation as well, for example, this would cause unwanted behavior.

5.7.2 Inverting Input

In some instances it seems sensible to invert inputs. For example, controlling aircraft with a thumbstick usually is accomplished by mimicking the behavior of actual aircraft controls. For up and down movements, this is exactly the opposite of the standard controls; therefore, some players wish to invert the controls to reflect this. The desired behavior can be achieved by using an Negate Modifier. By default, this modifier is applied to all axes; however, it can be configured on a per axis basis.

WASD Movement

In this tutorial it will be shown how to configure *WASD Movement* within an Input Actions > Input Mapping Context. Furthermore, an additional modifier will be introduced.

1. Create an Input Action for the movement, name it *"IA_Move"*, select value type Axis2D.
2. Create an Input Mapping Context, name it *"IMC_Move"*. Open it.
3. Add a new mapping, select IA_Move action.
4. Add 4 new inputs, named *"W"," "A", "S", "D"*, respectively.

   Recall, that the Movement Action is configured to use 2D vector values. Whenever on of these keys is pressed, the engine is going to output a value of positive 1 for the X value. By using Input Modifiers on specific keys, a setup akin one resembling thumbstick input can be achieved, where right and left are positive and negative X, respectively.

   Similarly, up and down are positive and negative X, respectively.

5. D is supposed to mean rightwards movement, representing positive X, which the key already represents. Therefore, the D key does not require and modifiers at all.

6. Conversely, A denotes leftwards movement; i.e. negative X movement; pressing it gives a positive signal, therefore it must be negated.

7. W is used for upwards or forward movement, represented by positive Y. Pressing the key is already positive, however, it is applied to the X-value of the 2D vector rather than positive Y. It must be modified accordingly, what can be accomplished using the Swizzle Input Axis Values Modifier. This modifier changes the order of axes to which the input values get applied to.

   Normally, the first float value will represent the X axis, and, if there is a second value, it will get applied to the Y axis. Similarly, if there's a third value, it will get applied to the Z axis. Now, the Order property of the Swizzle Modifier can be used to change this order. For example, the values are now set to get applied in the order YXZ. Because the first and only value being output by the W key shall be be applied to the Y-axis, this is the correct setting.

8. Finally, S should be negative Y, so using both Negate Modifier plus Swizzle Modifier may be used.

   Discussion: The code above shows how to configure the Input Mapping Context. For this to work, a respective section in the Player Controller is required to make the player move accordingly. Consult the tutorial script for respective example blueprints.

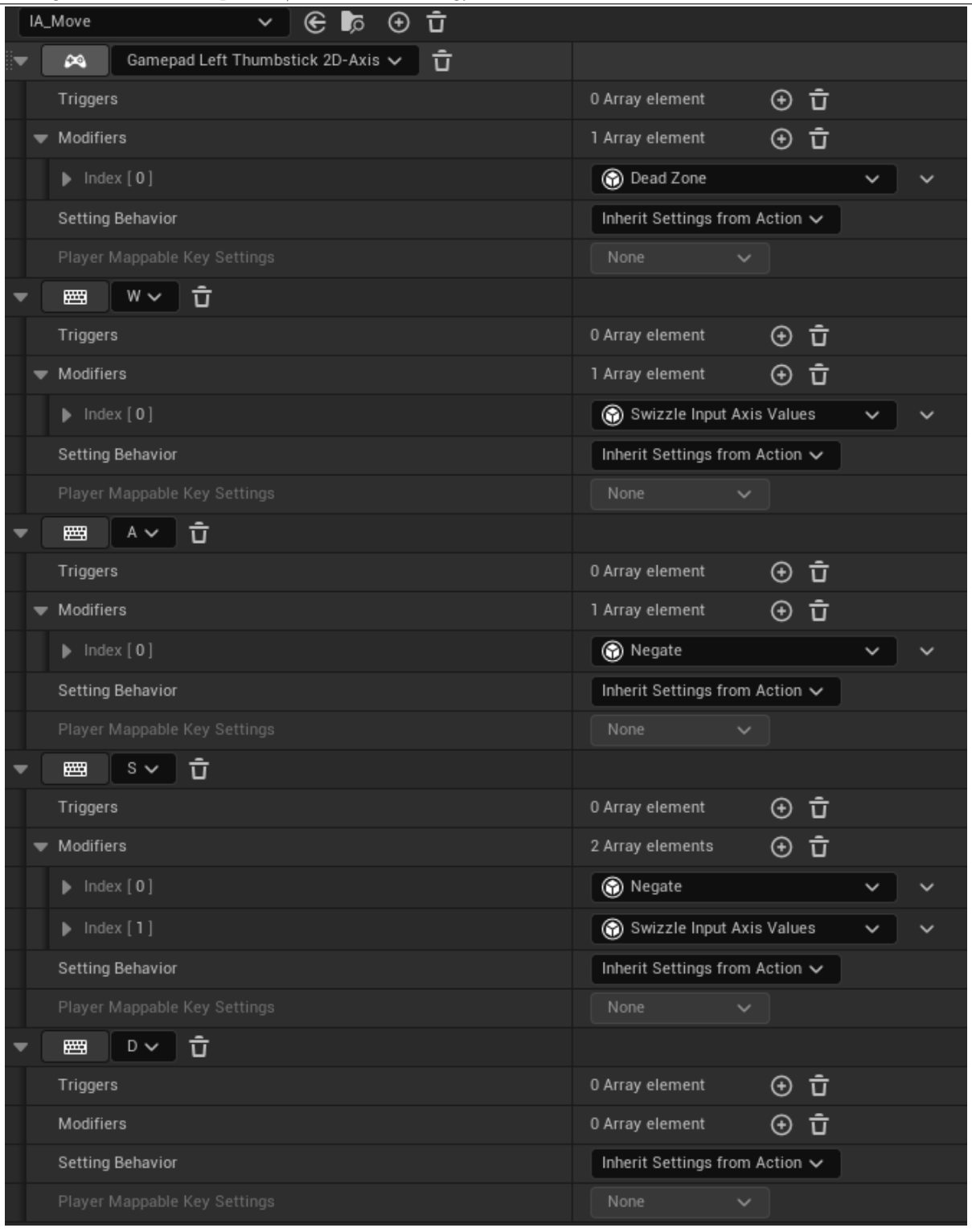

Figure 5.18: WASD Movement Modifiers

Collisions

At this point, a level has been created along with a character that moves around in it. What is still lacking, though, are ways to interact with the level.

In video games, one of the crucial ways of interacting is through collisions. That is, characters can take damage by colliding with other things, like weapons, enemies or dangerous things like fire or poison.

As for gameplay, items may be collected for later reuse or to increase a score by colliding with them. Similarly, a button may be ”pressed” by colliding with it, and so on.

This chapter on collisions discusses the various ways to use collision to enhance games.

6.1 Collision Volumes

Note, that there are two different uses for the word “collision” in *Unreal Engine*. The first is the everyday use of the word - the act of two objects colliding with one another.

However, *Unreal Engine* also uses the term ”collision” as shorthand for a *”collision volume”* which is what this section focuses on.

A Collision Volume is an invisible volume that is used as the bounds of an actor. So a collision volume determines where the official edges of an actor are, for the purpose of detecting collisions in the first sense of the word.

In addition, a collision volume also determines the official shape of an actor, for the purpose of simulating physics. In fact, an actor must have a collision volume defined in order to be able to simulate physics on that actor. Otherwise the Simulate Physics property will actually be greyed out.

There are two main ways to give actors collisions:

6.1.1 Collision Components

The first way is to add a collision as a component of the actor. This has already been discussed in the chapter on the Character class. To recap, recall that by default, a Character has this component named *”CapsuleComponent”*. And what this is, is a Capsule Collision. It can be found by clicking the Add button, going down to the Collision category; here are options to add a Box Collision, a Capsule Collision, or a Sphere Collision, which are essentially collision volumes of different shapes.

Capsule Collision is used for the character because that shape most resembles a humanoid. The Capsule Collision component is what prevents characters from walking through walls in the demo game, even though the character has no actual physical representation.

6.1.2 Collisions Defined on Mesh

The second way to add collisions is - for actors that have a mesh - to have the collision defined directly on the mesh itself. Collisions are usually created in the 3D-modeling program when the mesh is created, and then imported into *Unreal* along with the mesh.

However, third-party meshes do not a collision defined or the shape of the collision doesn't work right, collisions can be edited or created in *Unreal*.

To do so, the mesh may be Double-Clicked in the Content Browser to open it in its editor. For example, double-clicking Content Browser > Starter Content > Props > SM_Chair opens this chair in the Static Mesh Editor. The Static Mesh Editor has a viewport, and this viewport can be manipulated just like the viewport in the Level Editor, so developers can take a look at the mesh from any angle.

In the upper-left, it will display various stats about the mesh, such as the number of triangles and vertices it is composed of (see figure 6.1):

Figure 6.1: The Mesh Editor

Clicking the Show icon provides options to out what collisions, if any, this mesh already has defined.

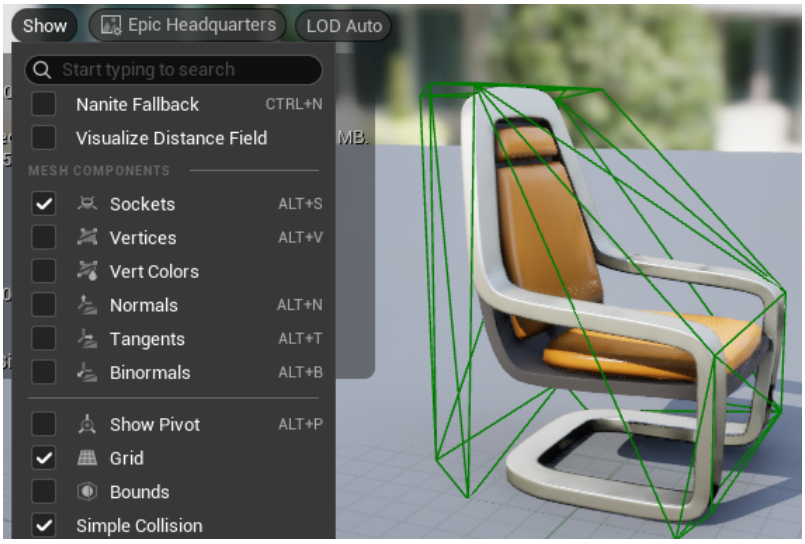

Figure 6.2: Simple Mesh Collisions

There are two kinds of collisions: Simple Collisions and Complex Collisions. In most situations, simple collisions will be used.

Simple Collisions

The volume outlined in green, as depicted in figure 6.2, represents the Simple Collision that came with the mesh.

Creating Custom Collisions

Assuming that the simple collision does not suffice for the developers needs, this tutorial shows how to create a custom collision.

- Get rid of the current collision by clicking Toolbar > Collision > Remove Collision.

To add a collision, go back up to Toolbar > Collision and select one from among several choices. The first three are the same shapes that are available as components - a sphere, a capsule, and a box. For example, adding a Add Sphere Simplified Collision adds a collision in the shape of a box.

From here, the transformation widgets to move, rotate, or scale the collision can be used as desired. If a chair is now added to the level, above the already present char and physics is enabled, the chair falls to the ground and collides with the other chair, making it appear like two balls collide.

Or, if the collision volume is made rather large, an actor approaching this chair will be blocked (i.e. running into this chair) from far away. This example demonstrates the fact that the collision volume is used when determining collision with other actors, and not the mesh itself. If the floor was made slightly sloped, the chair would roll down like a ball. This also demonstrates the fact that the collision volume is also used for determining the physics of the actor, rather than the actual mesh itself.

There is also the option to add multiple collision volumes, causing the engine to automatically combine them into a single collision shape.

The next five collision types are known as K-DOP collisions, where K stands for the number of faces that the volume has and DOP (Discrete Oriented Polytope) is an acronym for a complex mathematical term. A K-DOP is basically a collision volume that starts out as a box, but then has its edges beveled, meaning sloped, into the direction of the mesh, getting as close to the mesh as possible.

If none of those collision options give the precision needed, use something called the Auto Convex Collision tool to specify just how detailed the shape needs to be:

Creating Custom Collisions

- Go to Collision > Auto Convex Collision opening tool in a panel in the bottom-right of the editor.

  Edit the settings Hull Count, Max Hull Verts, and Hull Precision which basically are used to specify the maximum number of convex pieces, vertices, and voxels to use when creating the collision.

- Hit Apply, creating the collision based on these settings.

Creating Custom Collisions

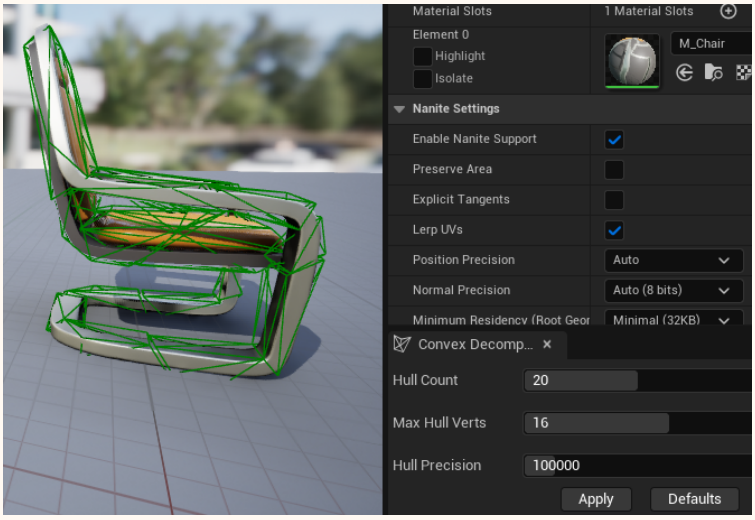

Figure 6.3: Auto Convex Collision

- Go to Collision > Auto Convex Collision opening tool in a panel in the bottom-right of the editor.

  Edit the settings Hull Count, Max Hull Verts, and Hull Precision which basically are used to specify the maximum number of convex pieces, vertices, and voxels to use when creating the collision.

- Hit Apply, creating the collision based on these settings.
- Finally, to restore the original settings, remove the collision and add a 10DOP-Y.

Complex Collisions

Changing the view using by ticking Show > Complex Collision and unchecking Show > Simple Collision switches the view accordingly.

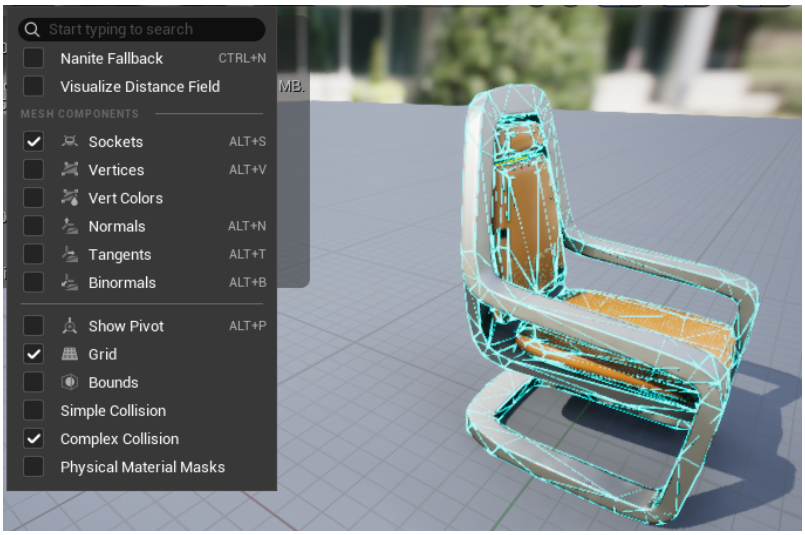

Figure 6.4: Complex Mesh Collision

As figure 6.4 shows, it looks rather accurate. In fact, it is 100% accurate. The complex collision is actually auto-generated by *Unreal* and matches the shape of the mesh triangle-for-triangle.

Obviously, this allows for a perfectly accurate collision, with the tradeoff being that it is extremely *computationally expensive*. So expensive, in fact, that the complex collision is rarely used.

As discussed above, all physics and collision interactions are, by default, handled using the simple collision, whereas generally speaking, the complex collision is only used when specifically requested.

That is, inside a blueprint there are some nodes for collision detection that allowing to choose whether the simple or complex collision shall be used.

Also be aware, that in the Static Mesh Editor > Details > Collision there is a property called Collision Complexity.
The first option for this property is Project Default. This refers to the Project Settings > Physics category, where sespective settings can be set determining what default value should be.

Therefore, if Project Default is selected here, it will use whatever value is set in the Project Settings. By default, when a new project is created, that default value gets set to Simple And Complex.
Simple And Complex causes the behavior shown so far, i.e. Simple Collision is used for most situations.

The next option is Use Simple Collision As Complex and when set, whatever the Simple Collision is will be used in place of the Complex Collision. Basically, this means that the simple collision will always be used. This setting would be used in situations when memory and performance are big issues and therefore complex collisions should be avoided.

Finally, Use Complex Collision As Simple means that whatever the Complex Collision is, it will be used in place of the Simple Collision, i.e. the complex collision will always be used, in every situation. This would be used in situations where perfectly accurate collision volumes for the purposes of determining collisions and physics are required and the performance tradeoff can be afforded.

Note, however, that when using this setting, the actor will not be able to simulate physics. It's just too expensive of an operation to calculate the physics of an actor with a complex collision in real time, hence the engine does not permit it at all.

6.2 Collision Events

A short example of Collision Events has been shown earlier, when a Trigger Volume has been shown to make a table disappear and reappear using OnActorBeginOverlap and OnActorEndOverlap in the level blueprint.

So in actuality, the Actor > Collision property has been utilized.

Concretely, by selecting an actor, then scrolling down to the Actor > Details Panel > Collision category, respective properties can be found:

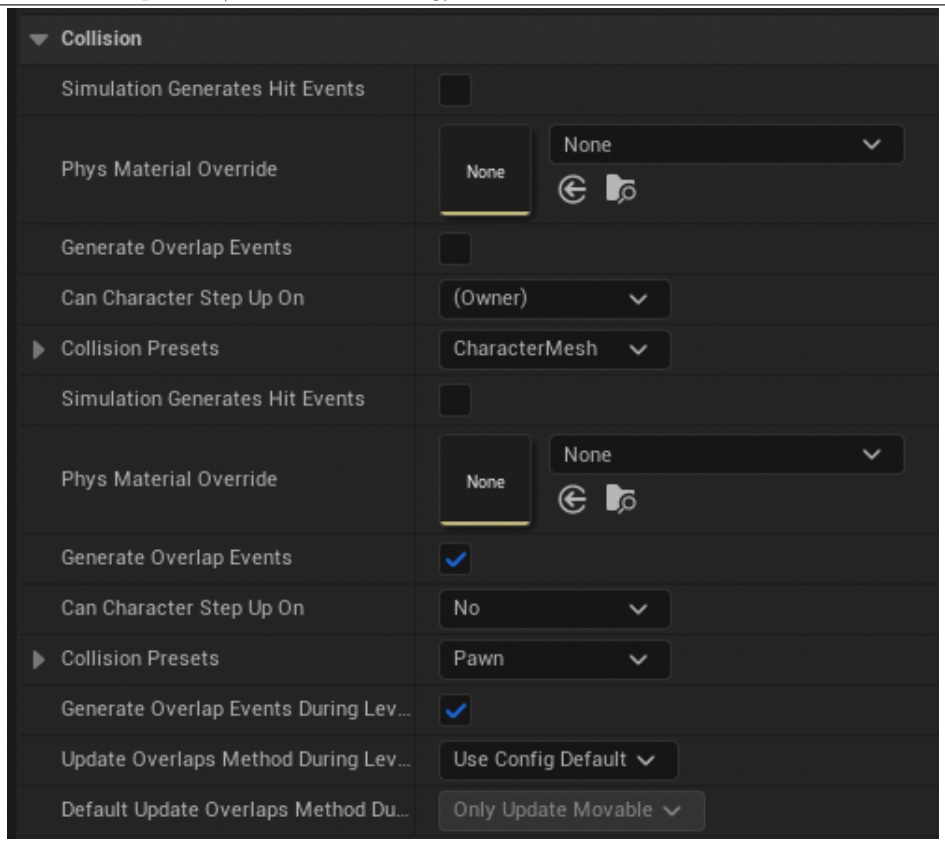

Figure 6.5: Actors: Collision Category

In the following, some noteworthy properties are discussed:

- **'Simulation Generates Hit Events'**: states that if two actors are set to block each other when they come into contact, collisions generate a Hit Event if Hit Events are enabled for that actor.

  Note, that when Generat Hit Events is active for a particular actor, hit events are generated even if the OTHER actor has them disabled. If the other actor has that enabled as well, it will also receive a Hit Event.

- **Generate Overlap Events** On the other hand, if two actors are set to overlap with one another, overlapping is going tol generate an Overlap Event if Overlap Events are enabled for **both** of these actors.

6.2.1 Collision Presets

A number of collision properties can be set by choosing from a predefined list of presets found under Actor › Details Panel › Collision › Collision Presets.

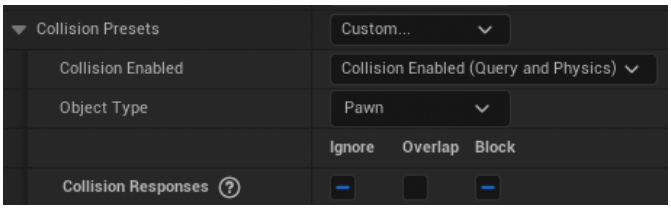

Figure 6.6: Actors: Collision Presets

All of these collision properties can be automatically set by choosing from a list of presets, or they can be individually set by choosing Custom and setting them one-by-one

- Collision Enabled: is one more way to set the block and/or overlap behaviors of an actor.
  There are four possible settings:
  - **Collision Enabled**: means that this actor is enabled for both Query Collisions and Physics Collisions.
  - **Query Collisions** refer to overlap collisions and **Physics Collisions** refer to blocking collisions. With this set to physics only, it will be able to block other actors and fire Hit Events, but it won't be able to fire Overlap Events.
    If this is set to query only, it will be able to fire Overlap Events but it won't be able to block other actors and it won't be able to fire Hit Events.
  - **No Collision** states that this actor will not be able to block other actors and it won't be able to fire overlap or hit events.
- **Object Type**: If a particular actor is a pawn or a pawn derived type such as a Character, this should be set to Object Type ⟩ Pawn.
- **Vehicle**: set if the actor is a vehicle (obviously).
- **Destructible Mesh** (a topic not yet discussed) set this to Destructible.
- **WorldStatic**: in case the actor does not move at all.
- **WorldDynamic** used if the actor *DOES* move, caused du to an animation or blueprint.
- **PhysicsBody**: used for actors that will move, caused by physics, i.e. due to gravity or the force of another object.

Note, that the Object Type doesn't actually do anything itself inherently. Its purpose is simply to be able to place each actor into a specific group, in a way that the section below (see figure 6.7) can be used to specify how the different types should interact with each other when they collide.

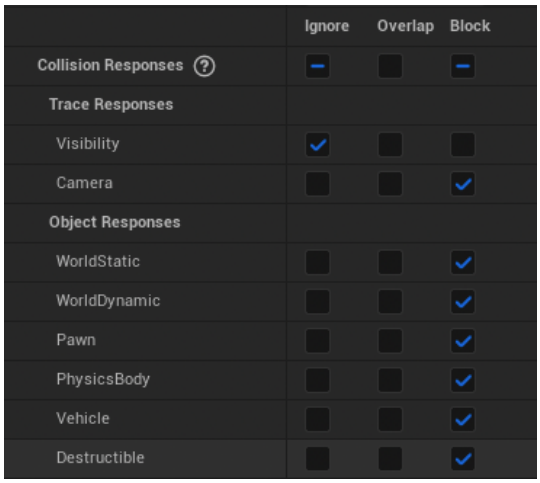

Figure 6.7: Collision Presets: Setting Matrix

This section consists of rows, one for each of the six Object Types. Each row is used to specify the behavior when this particular actor collides with the Object Type of that row. Therefore, the actual behavior depends on these settings for both actors involved in the collision.

Collision Presets

To tinker around with various settings, create a sample project from the First Person template. Then,

- Drag a Cube static mesh into the level.
- Drag a Sphere static mesh into the level and place it above the cube.
- Check Simulate Physics, and also make sure that Enable Gravity is checked.
- Make sure the sphere falls down once the level begins

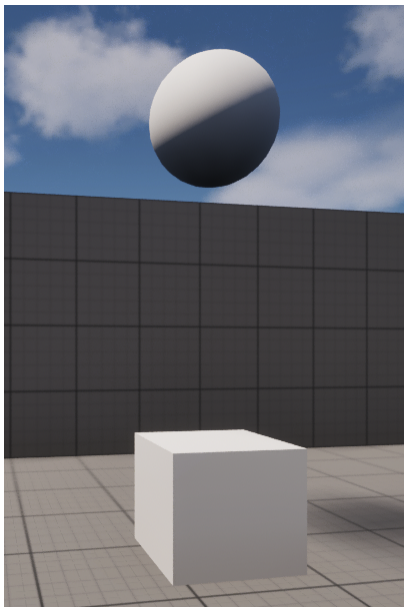

Figure 6.8: Demo: Place the Ball Above the Group of Cubes

Collision Presets

- Set Collision Preset > Custom.
- Set Object Type > PhysicsBody.
- Make sure the cube is a WorldStatic object.
- Make sure the sphere falls down once the level begins.

Next, check if these objects block each other:

Collision Presets

- With the sphere selected, check the `WorldStatic` row.
- With the cube selected, check the `PhysicsBody` row.
- Playing the game, the sphere will fall onto the cube, and it will come to rest on top of it. Note, that this will also fire `Hit Events` when the objects collide.

  Now, an `Hit Event` node will be created. Note, that this could be added to either the level blueprint or any of the two objects blueprints. In the following section, it will be demonstrated by example of the sphere.
- Create a blueprint from the sphere actor.
- Make sure that the sphere has `Simulation Generates Hit Events` checked.
- Add an `Add Event` > `Collision` > `Event Hit` node.

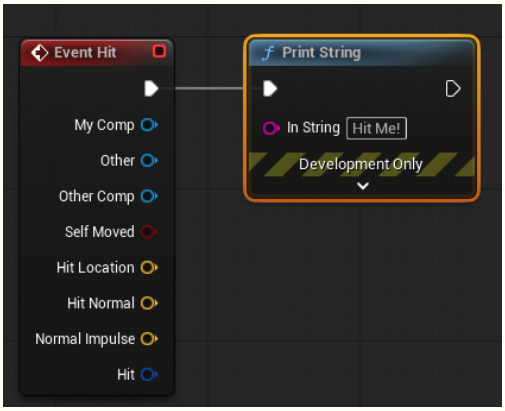

Figure 6.9: Event Hit Blueprint

Now, when the game is run, once the sphere falls on the cube the string *"Hit Me!"* is printed. Note it will continue to do so, because for every tick of gameplay the event will fire again as long as gravity continues to pull down the sphere.

Collision Presets

- With the sphere selected, check the `WorldStatic` row.

If the sphere is set, to overlap with `WorldStatic` objects, when played, the sphere will fall right through the cube. Recall, that this is because in order for blocking to work, **both** objects must be set to block. Note, that the `Event Hit` node won't fire.

Collision Presets

- With the sphere selected, set the `WorldStatic` row to `Overlap`.
- In the blueprint, change the `Event Hit` node to an `Event ActorBeginOverlap` node.
- Connect it to a `Print String` node; change the string to *"Overlapping..."*.
- Make sure that `Generate Overlap Events` is checked for both the sphere and the cube.

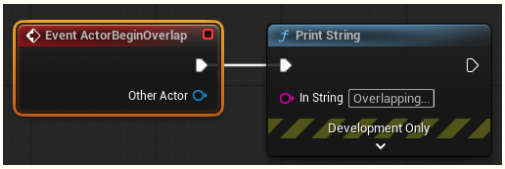

Figure 6.10: Sphere Blueprint: Event ActorBeginOverlap

When the game is run, the sphere falls through the cube, and as soon as the two actors overlap, the given message is printed *once*.

6.2.2 Trace Responses

`Trace Responses` are used to determine an actor's visibility to other actors.

If the actor that's looking is a camera, the Camera row would be used, while for all other actor types, the `Visibility` row would be used.

If this actor is set to `Ignore`, it will be invisible to other actors.
If it is set to `Overlap`, it can be seen by other actors, but it can also be seen through - this would be used for glass walls, for example.
If it's set to `Block`, then it can be seen, but it cannot be seen through.

6.2.3 Using Presets

The `Actor` › `Details Panel` › `Collision` › `Collision Presets` dropdown also has a long list of presets to choose from. Each preset will automatically select some combination of these properties.

For example, if **`BlockAll`** is selected as the preset, it will automatically set the `Collision Enabled` property to `Collision Enabled`, set the `Object Type` to `WorldStatic`, and it will set all of the responses to Block.

If **`OverlapAll`**, it will set `Collision Enabled` to `Query Only`, the `Object Type` to `WorldStatic`, and it will set all of the responses to `Overlap`.

If **`IOverlapAllDynamic`**, it will choose the same settings as `OverlapAll`, except it will set the `Object Type` to `WorldDynamic`.

6.2.4 Custom Object Types

To classify actors that to not exactly fit into one of the predefined `Object Types`, the solution might be to define custom ones

Custom Object Types

To do so:

- Open Toolbar » Edit » Project Settings » Engine » Collision.
- Choose **New Object Channel** to create a new Object Type.
- Set the Name to Enemies, select a Default Response to Block (default) and click Accept.

Now, back in the level editor there is a new a row for Enemies under the Object Responses section as well as an option for Enemies in the Object Type dropdown.

6.2.5 Character Stepping Up

The Can Character Step Up On property is used to specify whether or not a character will step onto the top of this actor when it walks into it or if it will block the character's movement. It assumes that the two involved actors' Object Types are set to Block one another, otherwise, this property does not apply.

In addition, at very small heights, a character will step onto an actor when it walks into it, regardless of what this property is set to.

Stepping on Objects

In this small tutorial it is shown how to set the step height and blocking.

- In the level, make the cube taller, by roughly double.
- Open the blueprint for the character the game mode is set to use: open Content Drawer » Content » FirstPerson » Blueprints » BP_FirstPersonCharacter
- Set Details Panel » Character Movement » Step Height to 500.
- Select the Cube, set Details Panel » Collision » Can Character Step Up On » No

  If the level is played now, the player character is blocked by the cube.

- Select the Cube, set Details Panel » Collision » Can Character Step Up On » Yes

  Conversely, the player character steps onto the cube.

6.3 Damage Caused by Collisions

This section shows how to decrease the health from a player when it collides with an enemy or similar harmful object.

Player Health

In this tutorial it is shown how deal with player health, how to set and subsequently decrease it once it collides with an enemy. You may want to continue using the `First Person` template created earlier.

- Drag a cube mesh into the level, rename it to `Enemy`. This actor is supposed to be some sort of object causing hard to the player character upon contact.
- Check `Actor` > `Details Panel` > `Collision` > `Simulation Generates Hit Events`.
- Create a blueprint out of the `Enemy` cube. Save it (ideally in a folder you created, equipped with a meaningful name). Open the blueprint.
- In the `Event Graph`, add an `Event Hit` node.

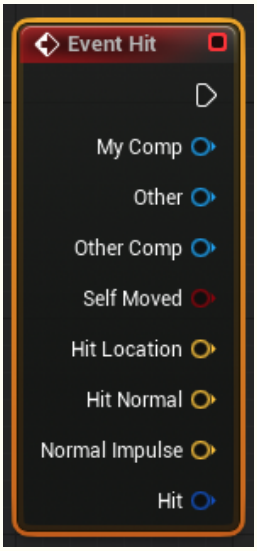

Figure 6.11: Event Hit Node

As figure 6.11 shows, this node contains several output pins:

  - The `My Comp` pin returns which component of this actor was hit.
  - The `Other` pin returns the other actor, i.e. the actor that collided with this actor.
  - The `Other Comp` pin returns the component of the other actor that was hit.
  - The `Self Moved` pin is a boolean, stating if the collision was directly caused by the player. I.e. if the player collides with the actor, or if a projectile fired from the player collides with the actor, this will return `False`. For any other collisions, this will return `True`.
  - The `Hit Location` pin is a `Vector` value returning the `X, Y, and Z` coordinates of the location where the hit occurred.
  - The `Hit Normal` pin is a `Vector` value returning the direction of the impact, i.e. it returns the angles relative to the `X, Y, and Z` axes.
  - The `Normal Impulse` pin returns how much force the impact had in the `X, Y, and Z` directions.
  - The `Hit` pin contains even more data about the collision. Dragging from this pin, and choosing `Break Hit Result`, creates a node containing many more details about the collision for scenarios where this information may be required.

- Connect `Event Hit` to an `Apply Damage` node. This node will apply damage to whatever actor is passed into its `Damaged Actor` pin.

Player Health

Note, that when an actor receives damage, which can be seen here is a built-in concept in the *Unreal Engine*, that doesn't actually do anything to the actor inherently. All the `Apply Damage` node does is to fire a `Damage Event` for the actor receiving damage, and then pass in the data from these other pins into the event. It is then up to the actor receiving the damage to define in its blueprint how it should handle the incoming information, i.e. this actor could react, decreasing a health variable or, before doing so, deciding if the damage applied is worth considering in the first place.

This rather useful concept allows for - think of things like invincibility potions or power ups. Using this concept makes thing like this easy.

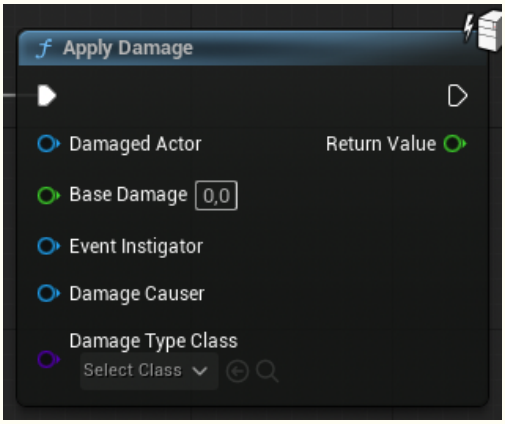

Figure 6.12: Apply Damage Node

- Again, the `Apply Damage` note contains several pins (see figure 6.12):
  - The `Damaged Actor` pin is - once again - used to specify which actor this node should apply damage to.
  - The `Base Damage` can be used to make the different damage-causing actors in a game more or less damaging than each other, i.e. more powerful enemies might cause higher damage.
  - The `Event Instigator` the actor that instigated a damage. For example, this could be a different character firing a projectile, hitting an object with it. The character that fired the projectile would be the instigator. On the other hand,
  - the projectile would be the `Damage Causer`.
  - The `Damage Type Class` is optional. This can be used to define blueprints specifying different types of damage. This types then could be set using the dropdown menu .

  In this situation, the damage should be applied to whichever actor collided with the cube, and that information can be retrieved from `Event Hit` 〉 `Other`. Therefore,
- Connect `Event Hit` 〉 `Other` pin to the `Damaged Actor` pin.
- Enter enter a `Base Damage` *10* for this cube.

At this point, the actors are set up to inflict damage. Now, it need to be specified what should happen to the player character when it receives damage.

Hence, the character blueprint is going to be set up accordingly:

Player Health

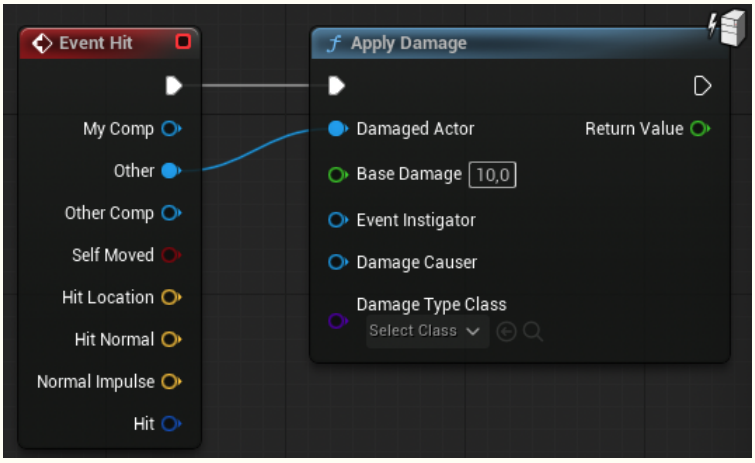

Figure 6.13: Enemy Blueprint: Apply Damage

- Open the character blueprint. Add an `Event AnyDamage` node to its `Event Graph`. This node will fire whenever the character receives any type of damage from any source. Whenever the actor receives damage, it shall decrease its health value. There is no respective variable, so
- Add a variable named *"Health"* of type `Float`.
- Compile the blueprint, give it a default value of 100.
- Add a `Health` `Set` node and connect `Event AnyDamage` to it.
- Add a `Health` `Get` node.
- Add a `Subtract` node.
- Connect the `Health` `Get` node with the upper `Add Input Pin`.
- Connect `Event AnyDamage` `Damage` pin with the lower `Add Input Pin`.
- Connect the `Add` `Output Pin` with the `Health` `Set` `Health Pin`.
- Finally, to be able to track the `Health` value, drag from the `Health` `Set` `Output Pin` and connect it to a `Print String` node, so that when the game is played, it outputs its actual health values to the screen whenever the player character bumps into the enemy cube.

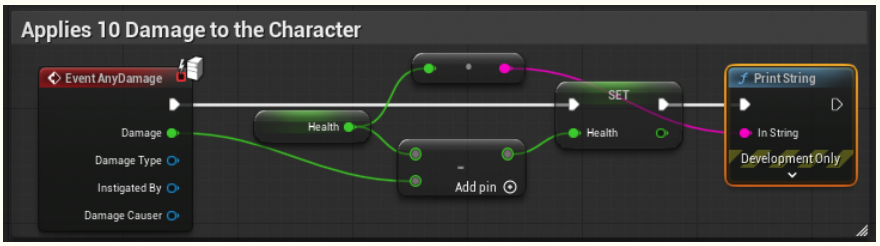

Figure 6.14: Character Blueprint: Apply Damage

This basically works, there is, however a problem: the player character is losing health way to fast. The reason for this is that the game is running in excess of 30 frames per seconds; so for each frame, the event fires.

Fortunately there is a way to circumvent this issue: to make the character invincible temporarily each time he

receives damage. This can be accomplished using

- a DoOnce node, followed by a
- Delay node.

Player Health

- Open the character blueprint. Add a Do Once node as well as a Delay node to its Event Graph.
- Connect the Event AnyDamage pin with the Do Once > Input pin. Connect the Do Once > Completed pin with the Health Set > Input pin. This causes the damage event to fire just once. Next,
- Connect the Print String > Output pin with the Delay > Input pin. Set Delay > Duration to 0.5. Connect Delay > Output with Do Once > Reset. This way, once printing the actual Health value to the screen is completed, there will be a delay of half a second, after which the Do Once is reset, allowing for damages to strike again.

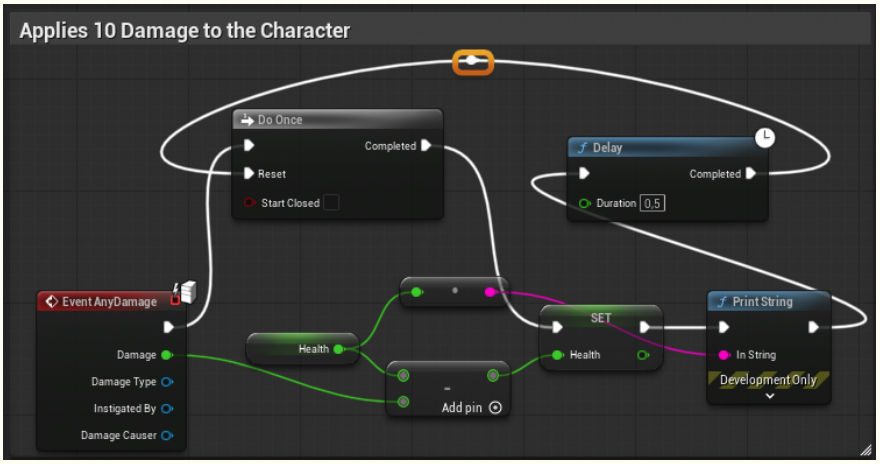

Figure 6.15: Character Blueprint: Apply Damage Using Do Once and Delay

Playing the game should now show the behavior desired: each time the player bumps into the cube, a value of 10 is subtracted from health.

Player Health

Finally, there is one remaining issue: The player character's health value can fall below 0; which is probably not the behavior desired. Usually, once the health decreased to 0, the character should be destroyed...

- Open the character blueprint. Add a Less Equal node to the Event Graph.
- Connect the Health > Set > Lower (Green) Pin to the upper Less Equals > Input Pin. Set the lower input pin to 0. Notice: The second pin in Set node can be used to get the value as well.
- Add a Branch node. Connect the Health > Set > Output Pin with the Branch > Input pin.
- Connect the Branch > True pin to a Destroy Actor node. This way, the actor gets destroyed once the Health value deteriorates to 0.
- Connect the Branch > False pin to the Print String node. In case the Health value is above 0, the blueprint will behave as before.

Now playing the game should feature the desired behavior: each time the player character bumps into the cube, the health decreases. Once it reaches 0, the player character is destroyed.

User Interfaces

This chapter discusses how to create user interface for *Unreal* games.

User interfaces include things such as menus and Heads Up Displays (HUDs). Menus are used whenever the player is supposed to make choices, usually offered in the form of lists. That is, for example, choosing between options like *"Start New Game", "Continue"* or *"Options"* at the beginning of a game. *HUDs* are used to provide information to the user at runtime, i.e. information that is displayed to the player while the game is in progress. For example, health, ammo, score, time remaining, things like that (see figure 7.1 below).

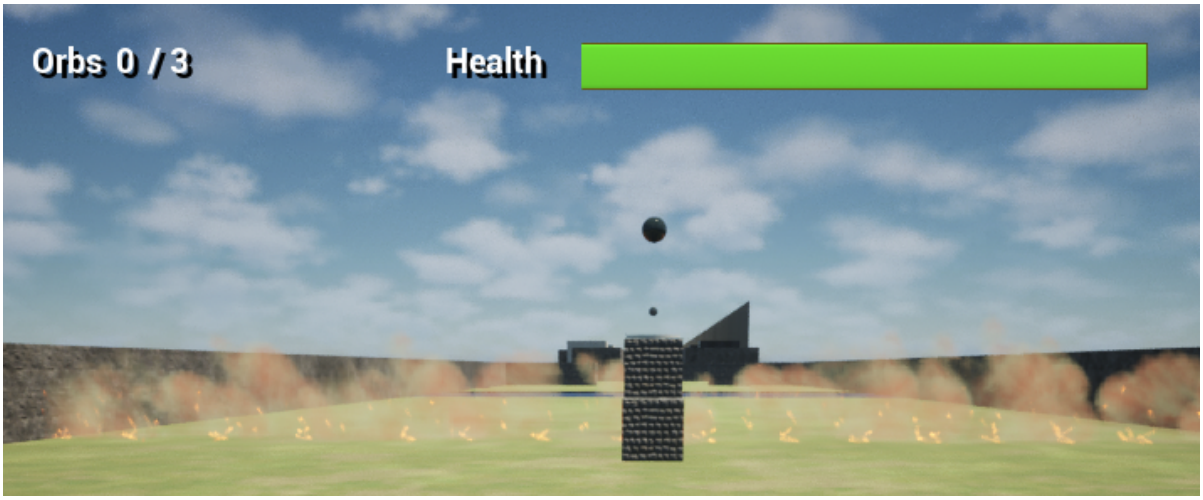

Figure 7.1: Heads Up Display Example: Stairway To Heaven Tutorial Game

Ther are actually a few different ways to create menus and HUDs in *Unreal Engine*. The original, legacy way was to create a blueprint from the HUD class, script the user interface elements in there, then subsequently set that HUD blueprint as the default for the game mode used.

That being said, this method is deprecated as of now, it is just mentioned here for reference.

7.1 UMG Overview

The modern approach to creating user interfaces in the *Unreal Engine* is a framework called *Unreal Motion Graphics*, abbreviated *UMG*. This is a modern, visual system for creating UI elements. Using UMG allows for faster creation of custom layouts.

7.1.1 Widget Blueprints

To create a UMG UI, the first step is to create a Widget Blueprint. Widget Blueprints are used to design the layout of the HUD, menus and things like that.

Like most modern UI frameworks, UMG widgets are containers, i.e. they may consist of other widgets contained within it.

As an example, a menu widget may be composed of a number of other widgets, like buttons. This menu itself may then be placed as a sub-menu into another widget, and so on.

Create an HUD

This short tutorial shows how to create a simple HUD using UMG.

- To create a `Widget Blueprint`, open `Content Browser` » `+ Add` » `User Interface` » `Widget Blueprint`.
- Select `User Widget`. This will create a new blueprint and add it to the content browser.
- Rename it as desired, for the purposes of this tutorial name it *"MyHUD"*.

In order for this widget to appear in-game, it needs to be called from another blueprint, such as the `Level Blueprint` or the blueprint of a `Pawn`.

Create an HUD

Before designing the layout for the HUD, its blueprint `MyHUD` will be called from the character blueprint, so its contents can be made visible ASAP.

- Open the character blueprint.
- Drag off its `Event BeginPlay` node and add a `Create Widget` node.
- In its `Class Pin`, select `MyHUD` as its widget class. Essentially, this will create an instance of the `MyHUD` class and add it to this blueprint.

  Notice: the `Create My HUD Widget` » `Owning Player Pin`; it specifies the `Player Controller` this widget should be applied to. If nothing is connected, it will be applied to the default `PlayerController`.
- Drag off of the `Create My HUD Widget` node and create an `Add to Viewport` node. This will take whatever widget is connected to the `Add to Viewport` » `Target` pin and draw it on-screen.
- Drag off the `Create My HUD Widget` » `Return Value` pin and connect it to `Add to Viewport < Target`.
- Compile the blueprint, save and close it.

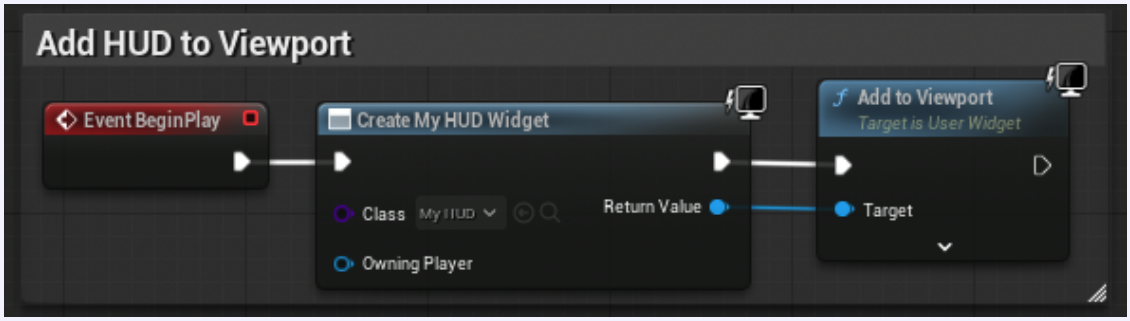

Figure 7.2: Adding HUD to Character Blueprint

Discussion: As soon as the character is created, this logic creates an instance of the `MyHUD` class and adds it to the character viewport. Hence, as soon as a widget is added to the HUD class, it should be visible as soon as the game starts.

Create an HUD

Now, some UI elements are added to the HUD class.

- Open the widget blueprint editor by double-clickting the Content Browser > MyHUD blueprint.
- Drag a text block into the editor.
- Compile, save and run the game.

Now, when the game starts, there should be an HUD saying *"Text Block"* in the viewport.

Figure 7.3: HUD Test: Displaying "Text Block"

Hint: There are two ways to remove an HUD from the viewport:

1. By using a Remove From Parent node, where the widget to be removed connected to its Target pin.
2. By using a Remove All Widgets node, which will remove any widgets added to the viewport.

7.1.2 Widget Designer/Editor

The editor is divided into two tabs - the Designer tab and the Graph tab, which is used to script UI logic. These two tabs may be switched by clicking the respective buttons in the upper right corner (see figure 7.4 below).

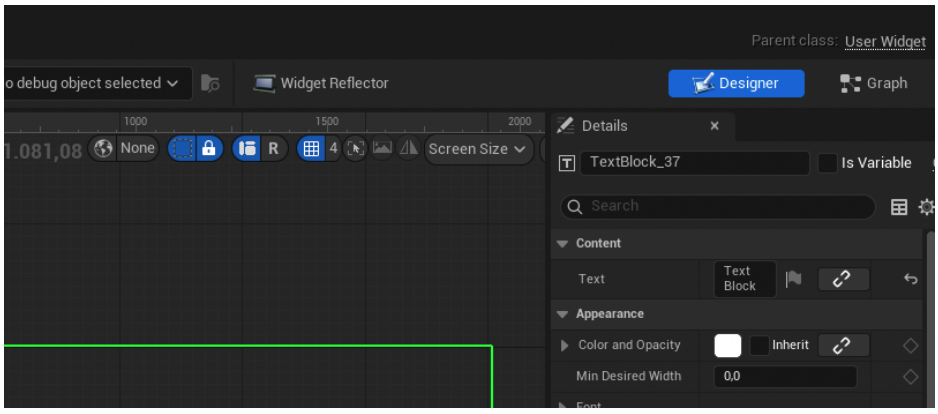

Figure 7.4: HUD Tabs - Designer Versus Graph

The main window of the Designer tab is the Visual Designer in the center. This can be used to visually lay out widgets, drawing UI elements and widgets from the palette, located to the right (see figure 7.5 below).

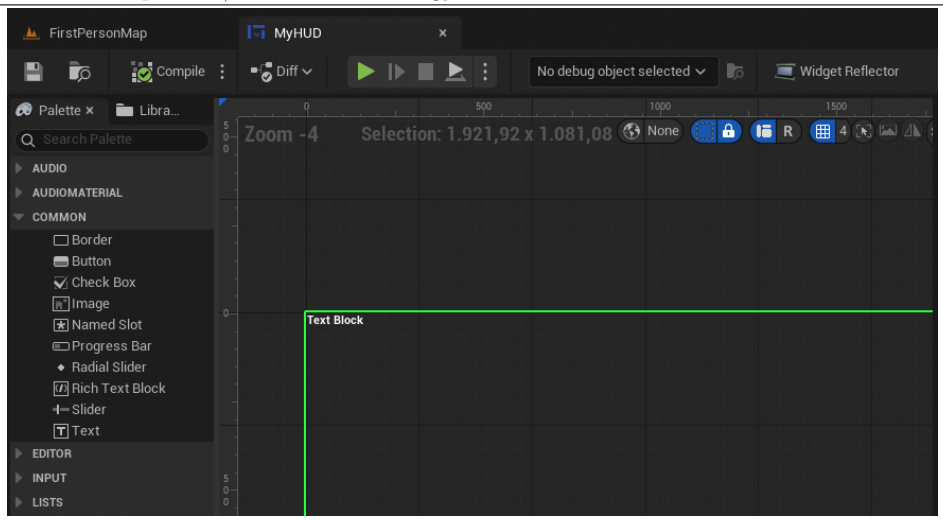

Figure 7.5: Widget Desinger With Widgets Palette

The palette contains all the pre-made widgets that can be dragged-and-droppet into the Visual Designer to design the HUD layouts.

Below that is the Hierarchy window. This window resembles the Outliner in as used in the level editor.

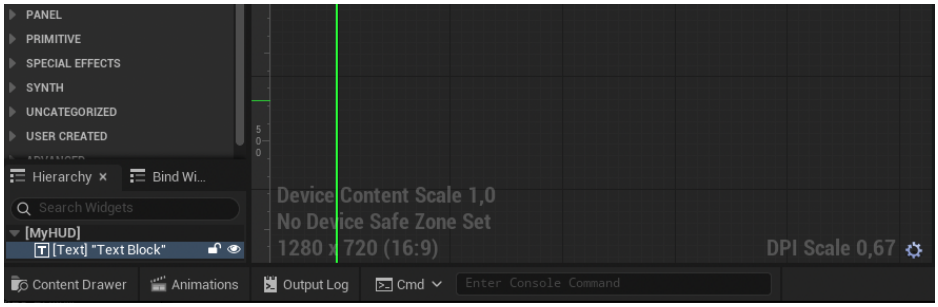

Figure 7.6: Widget Editor: Hierarchy Window

The Hierarchy Window is used organizes the elements in the Visual Designer and shows their parent-child relationships.

To the right of the Visual Designer is the Details window. This is just like its counterpart in the level editor. When a widget is selected, this is where its properties can be viewed and edited.

Finally, at the bottom of the editor the Animations windows is located.

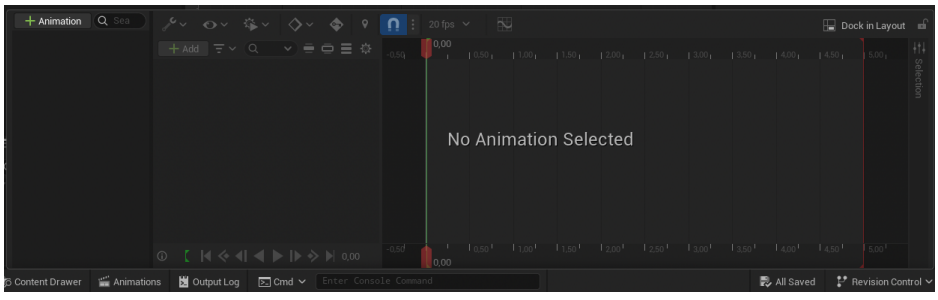

Figure 7.7: Wiget Editor: Animations Window

Animations can be used to animate layouts. For example, a menu can be animated in a way it appears to ”fly in” from the edge of the screen instead of merely appearing.

7.2 The Root Widget

This section discusses the Root Widget of a Widget Blueprint. A Widget Blueprint consists of at least one Root Widget, with the same name as the Widget Blueprint itself. For example, the widget demoed in the previous section was named *”MyHUD”*, so the root widget is also called *”MyHUD”*; as can be seen in the hierarchy window as depicted in figure 7.6.

By selecting the root widget, its properties can be managed in the details window, as usual. Some noteworthy properties found there are:

- Appearance » Color and Opacity / Appearance » Foreground Color. UMG has a rather complex system when it comes to the colors of widgets because the color of a widget depends, not only on what color it is set to, but also what color its parent widgets are set to. I.e., whatever color Color and Opacity was set the to the root widget will be applied. That color will be combined with the color of the child widget, with the color white being ignored. Whatever color is set in Color and Opacity to the Root Widget will be applied to all child widgets.

HUD Colors

In the following, a demonstration is given on how the color system works.

- In a first step, something called a Canvas Panel is added as a child of the root widget by dragging a Palette » Panel » Canvas Panel onto the root widget in the hierarchy window. Note, that a Canvas Panel is a container widget used to hold other widgets.
- Add a Text Palette » Common » Text Widget as a child of the Canvas Panel. Since the Canvas Panel is a child of the Root Widget, by extension, the Text Widget is now also a child of the Root Widget. Drag the text block window roughly to the center, adjust its size and Appearance » Font » Size.

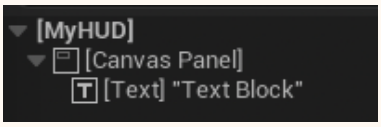

Figure 7.8: Hierarchy: Root - Canvas - Text Widget

Discussion: Assuming Root Widget » Color and Opacity is white and Text Widget » Color and Opacity is blue, the white would be ignored and the text would appear blue.

Conversely, if the Root Widget is set to blue and the Text Widget is set to white, the white of the Text Widget would be ignored, and the text would again appear blue (see figure 7.9).

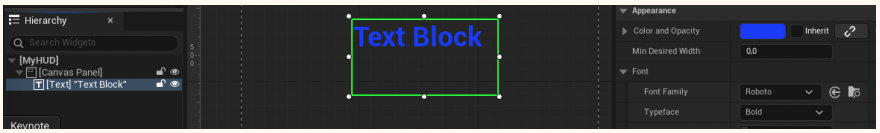

Figure 7.9: HUD Colors: Root White, Widget Blue

Finally, assuming that the Root Widget is set to blue, and the Text Widget is set to yellow, these colors would combine and the text would appear green

These two colors will combine, and my text will appear green.

- Foreground Color is similar to the Color and Opacity property, except that it will only be applied to children whose color property is set to Inherit. Note: Not all color properties have an Inherit property, therefore it might not be possible to apply this color to all widget types.

HUD Colors

- Set MyHUD > Appearance > Color and Opacity to white, so that it doesn't affect anything. Set MyHUD > Appearance > Foreground Color to yellow.
- Set Text > Appearance > Color and Opacity to red. Notice, the text appears in red color.

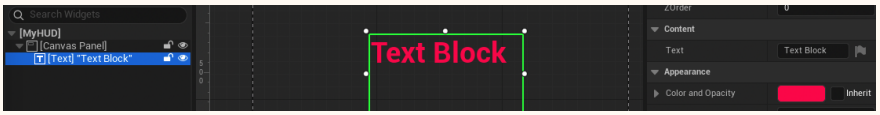

Figure 7.10: Color and Opacity: Red, No Inheritance

- Check Text > Appearance > Color and Opacity > Inherit. Notice, the text color changed to yellow, because it now ignores the read and inherits the yellow color from its parent MyHUD > Appearance > Foreground Color.
- Change MyHUD > Appearance > Color and Opacity to blue. Notice, that the text changed to green, because it now still ignores the red, but inherits both blue and yellow from the root widget.

Figure 7.11: HUD: Color and Opacity Yellow, Foreground Blue, Inherit Checked

Notice, that the Foreground Color property of the root widget has an Inherit property itself. If this is checked, Foreground Color setting will be ignored. Rather, this will be ignored and Color and Opacity will be inherited instead.

HUD Colors

- Check MyHUD > Appearance > Color and Opacity > Inherit. Notice, the text is now back to blue again.

Figure 7.12: HUD: Foreground Color Inheits Color and Opacity

- Appearance > Padding sets the padding around the content of the widget. Expanding this reveals that the padding may be set for each side individually.
- Interaction > Is Focusable. If checked, his widget will be able to attain focus, i.e. this widget can accept input from the keyboard. For example, if a button has focus, pressing Enter causes that button to be pressed.

7.3 Canvas Panel

Canvas Panels can be added to the layout via the Palette > Panel menu as shown in the previous section.

In the context of UMG, a panel is a container useful for aligning widgets and moving widgets as a group. Each panel has its own unique type of slot, which gives its children different positioning and sizing capabilities.

For example, if a widget is added to a Canvas Panel, it is placed inside of a Canvas Panel Slot. A Canvas Panel Slot allows for absolute positioning, meaning the exact location of the panel within the Canvas Panel can be specified.

If a widget is added to a Grid Panel, it would go into a Grid Slot, enabling positioning via the grid's row and column rather than exact pixel positioning.

Canvas Panel Slots posses the following properties:

- Position X and Position Y: allow for pixel-exact specification of the widget's position.
- Anchors are a way of specifying where on the edge of the panel the location of a widget should be measured from, in case the window size or screen size changes. Hitherto, the text box has been anchored to the top-left of the screen, as indicated by this Anchor Medallion.

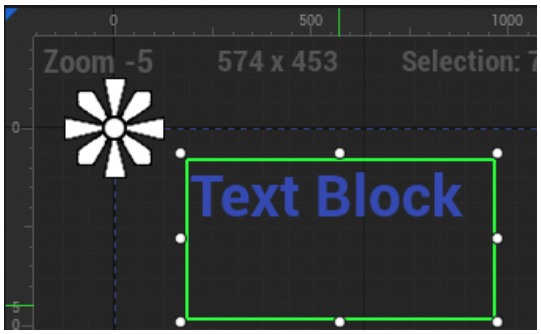

Figure 7.13: The Anchor Medallion

If the game is compiled, saved an run, the button stays at the same position relative to the top-left corner.

By clicking the drop down menu as found under Details Panel > Anchors > Anchors, the anchor position can be adjusted in to a number of predefined positions.

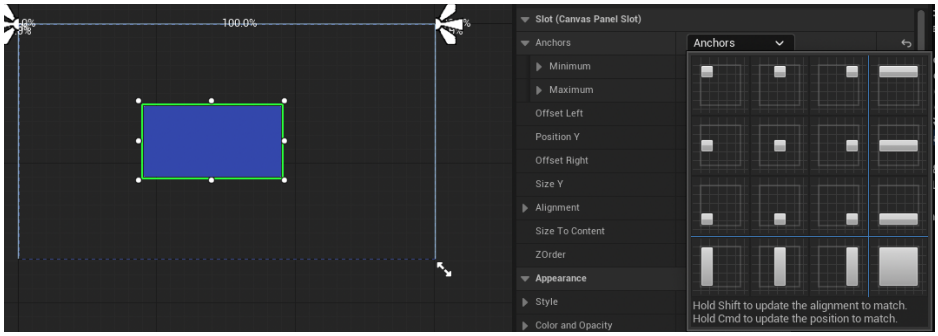

Figure 7.14: Anchor Positions

For example, if the anchor to position the widget to the top-left and top-right corners, the widget will be positioned accordingly. If required, the widget will be stretched or squished, so that its distance from the top-left and top-right corners remain constant.

It should be pointed out that the coordinate system that the position X and position Y properties use is based on where the widget is anchored to. For example, if a widget is anchored to the top-left, the position (0,0) is located in the top-left.

Also note, that the point on the widget that should be used as the anchor's location is determined by the widget's alignment. The alignment property uses a coordinate system where (0,0) represents the top-left corner of the widget, and (1,1) represents the bottom-right. Hence, if this widget is anchored to the top-left, and its position

is set to (0,0), when its alignment is set to (0,0), its top-left corner will align with the top-left corner of the canvas. However, if the alignment is changed to (1,1), now its bottom-right corner is aligned with the corner of the canvas. And if this is changed to (0.5,0.5), the center of the button will be aligned with the corner.

The Size to Content property automatically adjusts the size of a widget so it will be exactly large enough to fit the given text, Size X and Size Y settings will be ignored. Also note, that changing the font size will also affect this.

The ZOrder property determines the order in which widgets get drawn to the screen, and thus, for widgets in the same location, determines which widget overlaps which. Lower numbers get drawn first. As an example, assuming there is a button with ZOrder set to 0, along with a text with ZOrder set to 1, the button will be drawn first. In scenarios where these widgets are placed on the same X and Y position, the text is going to overlap the button, since it gets drawn last.

7.4 Horizonal and Vertical Boxes

This section discusses the related panel types Horizontal Box and Vertical Box. The Canvas Panel as discussed in the previous section only allows for absolute layouts, which is advantageous if all that's required is just a higher level of control over widget placement. It is not ideal for widget alignment though.

The box types discussed in this section are much more useful if proper widget alignment is a requirement.

- The Horizontal Box consists of a single row of slots of equal height, making it particularly useful for aligning widgets side-by-side.

Widget Alignment

This tutorial demonstrates how to align widgets horizontally and/or vertically. For starters, it is demonstrated how to align three buttons horizontally.

- First, add a Palette > Panel > Canvas Panel to the root widget.
- Then, add a Palette > Panel > Horizontal Box to the Canvas Panel.
- Then, add three Palette > Common > Buttons to the Horizontal Box.

  Discussion: So, as can be seen,these buttons stack up up side-by-side. To change their order, there are arrows used to move them to the slot to the left or right.

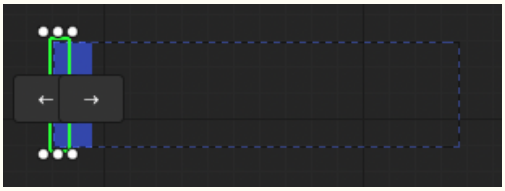

Figure 7.15: Move Buttons Within Horizontal Box

Notice, that the properties in its Details Panel > Slot category are different now because the button is in a Horizontal Box Slot as opposed to a Canvas Panel Slot:

- The Padding property used to add a padding around the content of the slot.
- By default, the Size property is set to Auto, what means the slot will be just big enough to fit its contents. Notice, that the size of a slot can be increased by settings it accordingly in the details panel. If this is changed to Fill, that slot will expand to take up the rest of the room available. If all the buttons in a slot are set to fill, they take up the available space equally. Notice, the Number property located to the right of the Auto / Fill buttons. This Number defines the weight of the button. If all the weights of the buttons in the box are equal,

they use up the same amount of space. If two buttons are weighted with the value 1, but the first button with 2, the latter will use twice the space, i.e. in this case, the third button uses up the same amount of space as the other two buttons combined.

- The next two properties determine how the content is aligned within the slot. By default, these are set to Fill. The Fill settings of the Alignment properties refer to the content of the slot filling up the slot. The Fill setting of the Size property refers to the slot filling up the Horizontal Box. So with the Horizontal Alignment of this slot set to Fill, its content, in this case this button, will stretch to fill up the slot horizontally. If the alignment is changed to Left, the button will only be as wide as the X value of its Image Size property, and it will align to the left. The same applies to center-aligned or right-aligned within the slot. Vertical Alignment works in a similar way, just for vertical alignment, obviously.

Notice, that the Vertical Box works in exactly the same way, just for aligning vertically.

7.5 Grid Panel and Uniform Grid Panel

This sections discusses Grid Panels and the Uniform Grid Panel.

The Uniform Grid Panel can have both rows and columns of slots. What makes it special is that all of its slots will always be the same size as each other, hence the name Uniform.

Grid Boxes

This tutorial demonstrates how to align widgets in grids.

- First, add a Palette > Panel > Canvas Panel to the root widget.
- Then, add a Palette > Panel > Universal Grid Panel to the Canvas Panel.
- Add a button to the panel. Increase its size in Details Panel > Style > Normal > Image Size so it is visible.

Discussion: By default, children added to a Uniform Grid Panel ar placed in Row 0, Column 0, which will always be the upper-leftmost slot of the panel. To place a button in a different row or column, use the arrow buttons (see figure 7.16 below) or use the Details Panel > Slot > Row and Column properties. Notice, that by default the content will be aligned to the left and top of the slot. At this point, the entire panel consists of a single slot, with the button aligned to the top-left corner. If the button is moved one column over, either by clicking its right arrow, or by setting the Column property of its slot to 1, the Uniform Grid Panel will automatically create a new column for the button to go into. Note, that the behavior is a bit non-intuitive: adding two buttons puts both of them into row/column 0/0. The developer then needs to put the button in the desired slot.

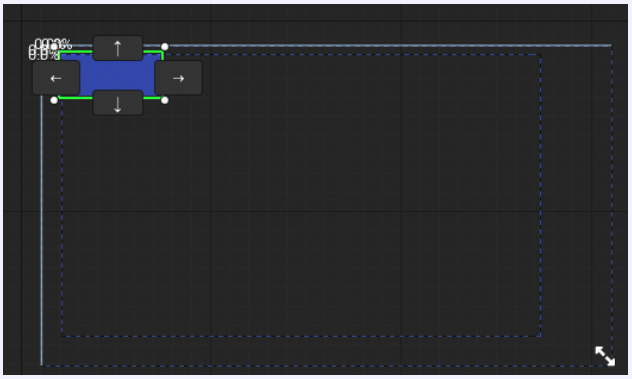

Figure 7.16: Universal Grid Panel: Placing Box Using Arrows

Grid Boxes

- Add another 3 buttons to the panel. Put them each into its own slot and change both vertical as well as horizontal alignment to fill.
- Select the Uniform Grid Panel.

  Discussion: Notice that it has three unique properties within a category called Child Layout.

  – The first is Slot Padding, setting the same amount of padding to each of the slots.
  – Minimum Desired Slot Width and Minimum Desired Slot Height, Height apply when the panel's Details Panel > Slot > Size to Content property is checked, i.e. the slots will resize to match the sizes of the buttons. These "Minimum Desired" properties, however, can override the Size to Content property, if the content doesn't force the slot to be at least the minimum size specified. As an example, in the Style menu for each of these buttons, their width is specified to be 50 units. But if the Minimum Desired Slot Width for this panel is set to a value greater than 50 , it will force each slot to be that width even though the Size to Content property is checked. Thus, for values below 50 this has no effect, because it's only a minimum value, not a maximum value. That said, if a value greater than 50 is set, the slots begin to widen.

- Set the Uniform Grid Panel > Details > Slot Padding to 5. Notice the padding between the buttons (see figure 7.17).

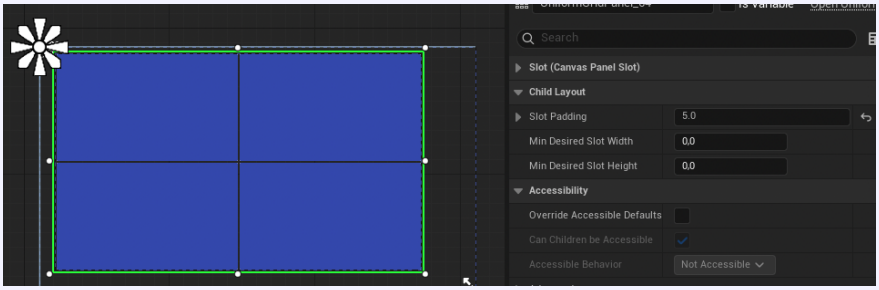

Figure 7.17: Uniform Grid Panel: Slot Padding

A Grid Panel is similar to the Uniform Grid Panel. However, its slots can differ in sizes from each other.

7.6 Widget Properties

This section discusses a selection of properties common to all widgets in UMG. Notice, that this is not exhaustive, i.e. not ALL properties will be covered. Rather, only a selection of most noteworthy properties will be discussed.

- Category: Behavior
  – Tool Tip Text causes the text to appear whenever the user hovers the mouse over the widget. To the right of it (as well as several other properties) an icon is located that somewhat resembles a chain. If clicked, a binding can be created using Create Binding. This binds the value of a variable to the tool tip.
  – Is Enabled if unchecked, the widget would be disabled, i.e. the user cannot interact with it, the widget would not receive any input (see figure 7.18 below).

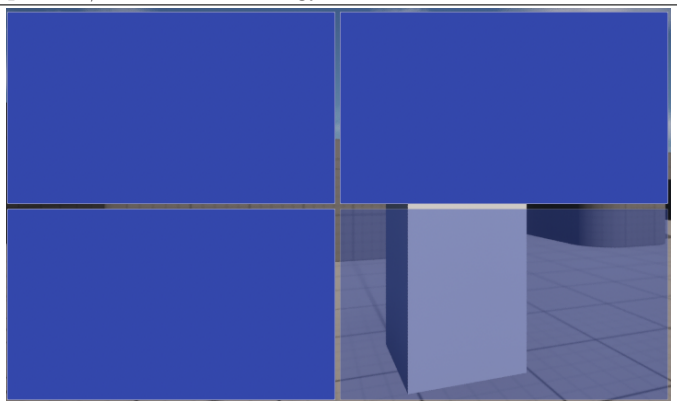

Figure 7.18: A Disabled Button

– Visibility
  * By default, Visibility is set to Visible which means it can be seen by the user, interacted with by the user, and it takes up space in the layout.
  * A Collapsed widget is invisible to the user, it cannot be interacted with by the user, and it won't take up any space in the layout (see figure 7.19 below).

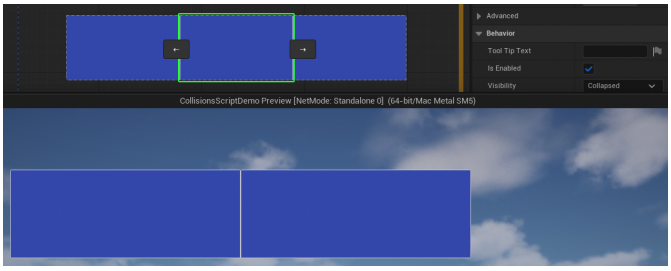

Figure 7.19: Collapsed Widget

  * Hidden makes the widget invisible to the user, and cannot be interacted with by the user, but it will take up space in the layout.

Figure 7.20: Hidden Widget

  * Not Hit-Testable (Self & All Children) is essentially the same as making the widget not focusable and not enabled, with the exception being that its appearance won't change, so it won't be grayed out like a widget normally would be when disabled.
  * Not Hit-Testable (Self Only) is the same as the preceding widget without being applicable to child widgets.

- Category: Render Transform
  – Transform contains the following sub-properties:

* Translation translates the widget, i.e. its location will be changed.
* Scale scales the size of the widget.
  Discussion: These work very similar to the Position and Size properties inside of a Canvas Panel Slot. However, the transform that occurs within a slot is known as a Layout Transform, as opposed to the Render Transform, which is that a Layout Transform can affect and be affected by the other widgets in the layout whereas a Render Transform ignores the layout.

  As an example, changing the the size of a button with a Layout Transform by adjusting the properties of its Horizontal Box Slot will change the size of the other buttons in the box as well, and size of the button is also limited by the size of the Horizontal Box it is contained in. However, if the size of a Render Transform is changed by adjusting the Scale property of the Render Transform category, the other two buttons will be ignored, as will the bounds of the Horizontal Box.

  Most of the time Layout Transforms are used to set the position and size of widgets. That being said, Render Transforms can be useful in some situations, such as animating widgets.
* Shear can be used to distort widgets (see figure 7.21 below).

Figure 7.21: Sheared Widget

* Angle used to rotate the widget.

– Pivot Just like the Alignment property of the Canvas Panel Slot, this uses a coordinate system of (0,0) to indicate the top-left of the widget and (1,1) to indicate the bottom-right of the widget. By default, it is set to (0.5,0.5) which places the pivot point in the center of the widget. So with the pivot point in the center, if the widget was rotated the it would rotate around its center. If it were scaled, it would expand outwards from the center or inwards towards the center.

- Category: Performance. This contains only one property:
  – Volatile this property is used in combination with the Invalidation Box widget in Palette » Optimization. The purpose of the Invalidation Box is to cache its children widgets, meaning those are saved in memory. This will increase performance if those widgets don't change often.

    However, in case that one of those child widgets changes often, perhaps because it animates and needs to be drawn differently each frame, that widget should *NOT* be cached. In that scenario, setting the widget to Volatile prevents it from being cached.
- Category: Rendering
  – Clipping: Content that overflows the bounds of a widget will normally be rendered anyway. The desired behavior can be selected from a drop down list.
  – Pixel Snapping: The widget will be drawn snapped to the next pixel if set. The desired behavior can be selected from a drop down list.
  – Opacity: Sets the widget's opacity, where 1.0 means fully opaque, whereas 0.0 means fully transparent.
- Category: Navigation This category affects how the user can navigate to the different widgets using the keyboard. I.e. once the game runs, some widget has focus. Navigation determines which widget should gain focus if the user uses a navigation key (like Cursor Left); i.e. the user if free to move around the UI using keystrokes. This is because the widget properties in this category are set to Escape. Escape in this context means that the user is able to escape the widget in a particular direction. For example, hitting ← will escape the widget to the left.

  In other words, if a widget has focus and is configured to escape, focus will move to the next widget upon keystroke.

However, in case the widget is configured to Stop instead, the navigation can no longer move through this widget.

Wrap may bes used with certain panels, such as a Uniform Grid Panel. However, this has to be set on the panel. It does the following: if the edge of a row of buttons is reached, navigation will wrap around the other side. For example, if the user approached the right-most button, hitting → will wrap navigation around, so that the left-most widget has focus.

Explicit means that the widget to navigate to can be set directly in here.

Custom and CustomBoundary allow to define functions used to determine which widget should be navigated to next.

- Category: Localization Its only property Flow Direction Preference is used to determine in what direction the words in a particular language flow.

7.7 The Visual Designer

This section discusses the Visual Designer, the main window of the Designer tab.

Basic Controls

- **Panning Around the Window** Hold down RMB and drag.
- **Zooming** Using the mouse scroll wheel. A gauge in the upper-left corner shows the actual zoom level.

Upper Right Corner Options

In the upper-right corner, there is a row of buttons that pertain specifically to the Visual Designer:

- Localization If a game is to be released in different languages, different translations can be loaded in the Project Settings. To preview these translations, click the left-most icon (in the upper right corner), selecting Region & Language opens the respective section, where the Preview Game Language can be set. Using the left-most button toggles this feature on and off. If several translations are available, these can be selected here as well.
- Toggle Dashed Outline. By default, dashed outlines are a visible which are intended to represent the border of a panel. This icon can be used to toggle this on and off.
- Widget Locking: Over in the hierarchy window some icons are located, like a padlock icon. Clicking it will lock the widget, i.e. the widget (and its children) can no longer be selected. The respective button in the upper-right corner switches the whole locking system on and off.
- Toggle Transform: The next pair of buttons are used to toggle between Layout Transform and Render Transform (Shortcut: W / E). As discussed in the previous lecture, a Layout Transform is when the positioning of a widget is adjusted within its slot, causing the widget to be bound by the layout. Conversely, a Render Transform is when adjustments are made within the Render Transform category, which causes the widget to ignore the layout.

  So with Layout Transform selected, if a button is selected , it cannot be moved within its Horizontal Box. It can only be removed from the box, causing the box to re-layout

  On the other hand, if Render Transform is selected instead, t can be moved freely, adding an offset to its base position, which stays in the same location, maintaining the integrity of the surrounding layout.

  And it's worth noting that if this is set to Render Transform, it will *NOT* change the way the sizing control behaves when the edged of a widget are clicked and dragged. It will still behave like a Layout Transform. These buttons only affect positioning.
- Grid Snapping works just as "regular" grid snapping.
- Zoom to Fit will center the layout on the screen and zoom into it.

- Toggle Landscape / Portrait can, obviously, be used to toggle between landscape and portrait mode. These modes depend on the selected target device (think smartphones or tablets). Since not such device is simulated here, it is disabled. However, this can be modified using the icon labeled Screen Size by default. This offers a plethora of different screen sizes as well as templates for a large number of different devices.

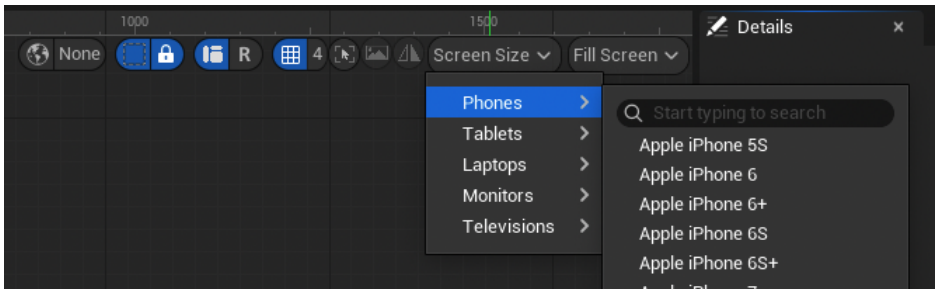

Figure 7.22: Choosing Simulated Device

- Screen Resolution and Aspect Ratio can be set in the bottom-left corner. If everything is set up correctly, it is possible to switch between landscape and portrait mode.
- Fill Screen is used to set the overall size of the layout relative to the screen size:
  - With Fill Screen set, the layout will always be equal to the size of the actual screen.
  - Using Custom enables to choose a specific size for the layout.
  - Custom on Screen is the same as Custom except there is an additional outline for the screen size itself.
  - Desired sets the bounds of the layout just large enough to fit all of the widget within it while still obeying the spacing set by the anchors.
  - Desired additionally enables to see the edges of the screen.

7.8 Noteworthy Widgets

This section provides an overview on a selection of widgets that appear to be the most important widgets used.

7.8.1 Text Widget

This widget is located in Palette > Common > Text Widget. When dragged into a panel, it starts out with the default text *"Text Block"*. It can be changed in Details > Content > Text.

As briefly mentioned in the preceding section, its color and opacity can be edited in Details > Appearance > Color & Opacity. And Inherit is checked, it will inherit the settings from the Foreground Color of its next highest parent. Since the Canvas Panel doesn't have a Foreground Color property, so in this case it would inherit from the root widget.

Font can be adjusted in the details panel as well; unfortunately, *Unreal Engine* only comes with one built-in font, *Roboto*. That said, custom fonts can be imported into the engine by dragging and dropping the desired fonts into the content drawer. Fonts also can be decorated with materials; however, not just any material can be used; those have to be materials specifically designed for usage with fonts.

In Details > Appearance > Fonts > Outline Settings outlines can be added, along with color, sizes, etc.

In Details > Appearance > Strike Brush is used to set what brush to use when adding a strikethrough to text. The brush can either be an image or set using the Tint property.

Shadow Offset and Shadow Color are intended to use and adjust text shadows. Be aware that shadows are, by default, fully transparent, i.e. invisible. This can be changed by adjusting the *Alpha* channel in Details > Appearance >

Shadow Color > A. The Shadow Offset property is used to determine how far in X and Y directions the shadow should be located.

The Min Desired Width property is used if the widget is located within a container, competing with other widgets for space.

Transform Policy allows to set a text to all lower or upper case, whereas Justification is used to align the text to either *Left, Center or Right* with respect to its container.

Finally, the Wrapping category specifies the wrapping behavior. Text can be wrapped in two ways:

- Using Auto Wrap Text will automatically wrap text to the next line once it reaches the edge of the widget. item Using Wrap Text At wraps text once it reaches predefined width.

7.8.2 Button Widget (Including Event Handling)

This section discusses the Button Widget, including binding it to events, so that when the player clicks the button something happens.

Button Widgets

This tutorial shows how to add Widget Buttons, including event handling.

- Drag a Palette > Common > Button Widget into a Canvas Panel.
- In the button's Details > Appearance section, the looks of the button may be configured.
- Set the Color and Opacity property to *blue* and the Background Color property to *red*. Make sure all parents colors are set to *white*.

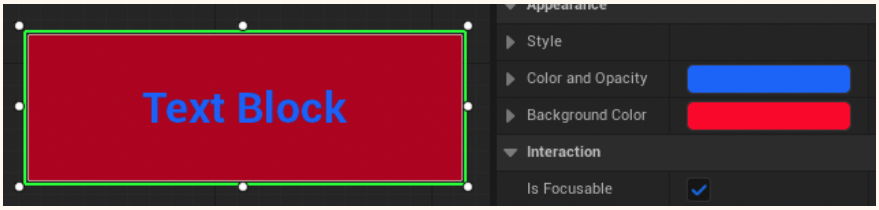

Figure 7.23: Button Widget: Color and Opacity versus Background Color

Once again, Color and Opacity sets the color of the button's content, while Background Color sets the color of the button itself. Therefore, adding a Text Widget as a child of the Button Widget and subsequently setting the button's Color and Opacity changes the text color. Setting the button's Background Color changes the button's color.

Notice the Style property. It actually is a collection of properties which in turn possess their own sub-properties. The first four of which represent different button states: Normal, Hovered, Pressed and Disabled. Expanding their respective properties enables to alter the state's appearances.

For example, expanding the Hovered property, setting its tint to blue will cause the button to turn blue when hovered over with the mouse. Note, that if a Background Color is set, that the Tint will combine with that color. So if the Background Color is *blue*, and the Tint of the Normal state is set to *yellow*, the button will appear *green* when in its Normal state.

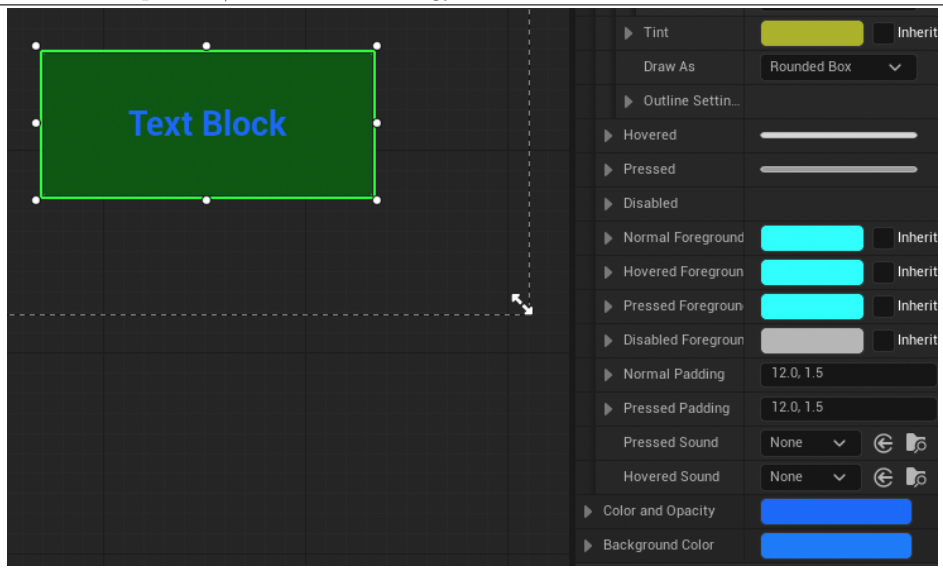

Figure 7.24: Combining Background Color and Tint

The button background can also set to use an image by using a drop-down next to the Image property. The respective image has to be imported via the Content Drawer first. Any colors applied to the button's background, it will use that color as a filter on top of the image.

Notice the Image Size property. This property does *NOT* affect the button when it's in a Canvas Panel. In other panels, the Image Size property will change the size of the button, regardless of whether or not it actually contains an image.

The Draw As property affects how the image is drawn onto the button. If set this Image, it will stretch or shrink the image evenly across the entire button; there is also the option to tile the image either horizontally or vertically. There are some more options, which are not in detail discussed in this document.

The Foreground Color sets the color for each button state; children inherit this color.

There are also options to set the Padding for the button's content, in dependence of the state the button is in. This may be used to make the button appear animated. For example, setting higher values for the Pressed state compared to other states makes Pressed appear like a button that is, well, pressed.

For both the Pressed and Hovered state, audio assets can be set, so that the button plays that sound whenever it is in the respective state.

Events and Event Handling

Finally, all the way down in the Details panel the Events category is located. Note, that not all widgets have events tied to them. However, the Button Widget has five events tied to them:

- The event On Pressed fires when the button is pressed.
- The On Clicked and On Released events will fire once the button has been pressed and subsequently released, and can apparently be used interchangeably.
- The On Hovered event fires when a mouse hovers over the button and the On Unhovered event fires when the mouse cursor leaves the button area.

Button Widgets

Events can be added by clicking the respective Details > Events > On Event > + icon. This opens the blueprint editor, enabling handling the event with that particular blueprint script.

- Click Details > Events > On Pressed > +. This opens the Blueprint Editor > Event Graph. Notice, a On Clicked (Button_Name) node has been added automatically.
- Connect the output pin with a Print String node; name the In String *"Clicked"*.
- Add two similar event handlers for the On Pressed and On Released events and name their In String properties Pressed and Released, respectively. (Hint: Go back to the Widget Designer by clicking the according button in the upper right corner).
- Compile and save.

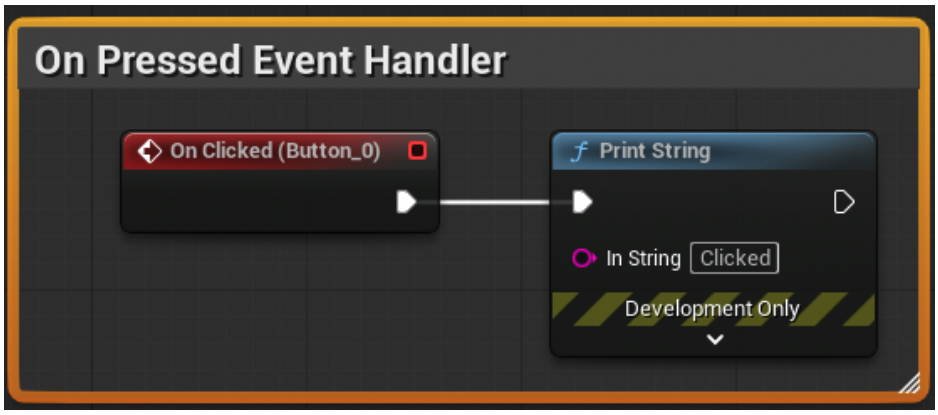

Figure 7.25: On Pressed Event Handler

Run the game and click the button - but do not let go the mouse button immediately.

Notice, that when the mouse button is pressed, the respective message appears in the game window.

Also notice, that both Clicked as well as Released appear on the screen as soon as the mouse button is released. This is because Clicked is considered as a mouse press, followed by a release; therefore, the event fires accordingly. Notice, that events work the same when using a keyboard (or other means of input, like game controllers).

7.8.3 Border Widget & Image Widget

Border Widgets and Image Widget are both basically container widgets, the former intended to form, well, borders, the latter used to contain images.

Firstly, the Border Widget possessed two main categories, Content and Appearance. The Appearance category is used to alter the appearance of the border by providing an image or setting its color. For example, if a black border is desired, the Brush Color would be set to black. Note, that at this point the widget appears just like a box. However, when a child widget is added, this child widget would cover the larges portion of the area, whereas the other part appears like a border. The respective border padding can be set in Details > Content > Padding.

The Image Widget is very similar. The only real difference between the Border Widget and the Image Widget is that the Border Widget has a Content category since it is especially designed to have children, while the Image Widget is not.

7.8.4 Progress Bar Widget

This widget is often used to display things like the amount of health, magic, energy or ammunition a player character has remaining.

A Progress Bar displays these values by percentage using the Details > Progress > Percent property; whereas, strangely, the value is given as a Float value ranging from 0.0 to 1.0, the latter denoting 100 percent.

Its color can be set using Appearance < Fill Color and Opacity. Again, this color will be combined with the Style > Fill Image property. The direction that the fill color is filled in is determined by the Progress > Bar Fill Type. The default direction is from left to right, but there is option of having it fill in different directions, such as right to left or top to bottom. Background images can be set using Style > Background Image.

Checking Progress > Is Marquee converts the progress bar into a Marquee, what is basically a progress bar showing continual movement. This would be used if the completion status of a task is unknown.

Finally, the Progress > Border Padding property to give the bar a border.

Binding the Progress Bar to a variable.

In the chapter on collision it has been discussed how to sense when a character is being harmed by an enemy, thus losing health. In the respective blueprint, the deteriorating health has been displayed using a Print String node, which while it does work, is not the optimal way of user interaction.

Setting Progress Bar From Variable

Thus, the Print String node will be replaced by a progress bar gauging health. The print command just prints out the this value, which in turn is stored in a variable called Health. Health holds values ranging from 100 to 0, whereas the progress bar expects values ranging from 0.0 to 1.0, which is why it must be converted accordingly.

- Open the first person character blueprint. Find the Health variable and change its default value to 1.0.
- Open the enemy blueprint and change Apply Damage > Base Damage to 0.1.
- Open the MyHUD blueprint. Add a Common > Progress Bar place it in the upper-left, set its anchor to the upper-left corner.
- In the first person character blueprint, delete the Print String node; it will be replaced by the progress bar.

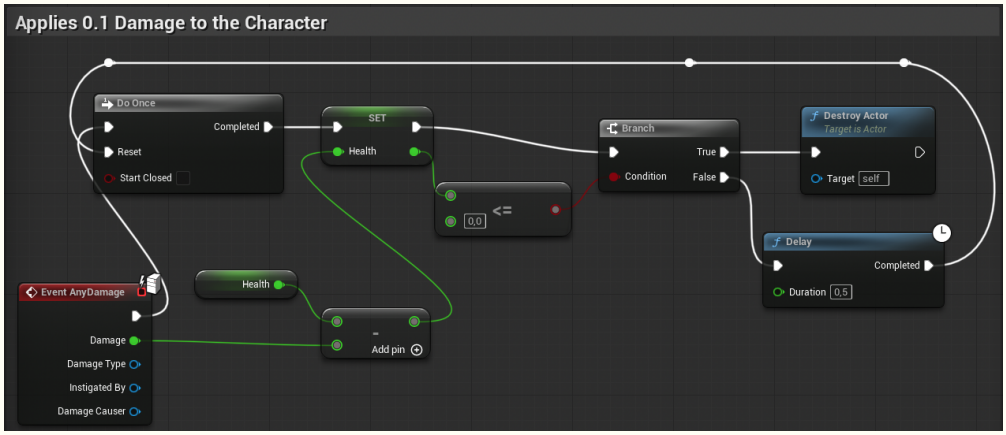

Figure 7.26: Apply Damage Blueprint

Setting Progress Bar From Variable

Now, the First Person Character > Health variable is bound to Progress Bar > Percent.

- Click MyHUD > Progress Bar > Percent > Binding > Create Binding.

  Discussion: his will automatically create and open a function that runs every tick of gameplay, with whatever is fed into the Return Value being used as the value of the Percent property for that particular tick. Hence, the Health variable shall be fed into this pin.

- Right-Click into the graph and create a Get Player Character node. Cast this to a First Person Character node.
- Drag from Cast To First Person Character > As First Person Character Pin, search for Health, find and connect the GetHealth node. Connect Health to Return Value > Return Value.
- Connect GetPercent > Output Pin to Cast To First Person Character > Input Pin.
- Connect Cast To First Person Character > Output Pin to Return Node > Input Pin.

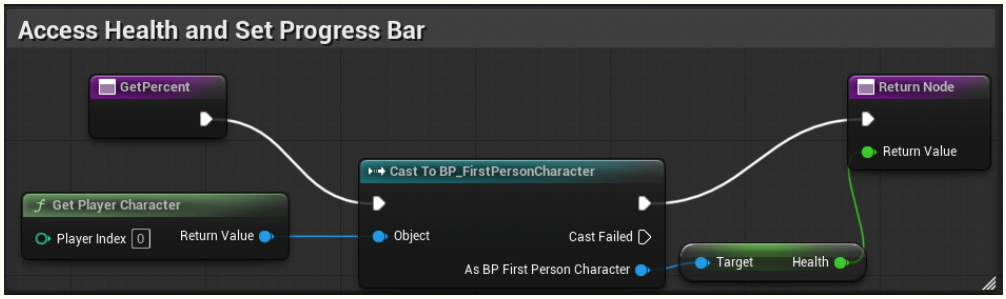

Figure 7.27: Set ProgressBar Blueprint

- Compile and save.
- Test the game. Each time the player character collides with the cube, the character's health progress bar should decrease.

7.8.5 Check Box Widget

A Check Box widget's initial state can be set by the Appearance > Checked State property; the possible values are Unchecked, Checked or Undetermined, where Undetermined can be used to find out whether or not the user actually interacted with the widget.

In the Style category, all sorts of combinations of Unchecked, Checked, and Undetermined, and Normal, Hovered, and Pressed can be set for the check box. This is working very similar to how button widgets work (see section 7.8.2).

The Check Box Type property can be set to Check Box or Toggle Button. When this is set to Check Box, the size of the Check Box will be set by the Style > Image Size. If set to Toggle Button, the widget will automatically be stretched to fill the slot.

Also note the Padding property. There is also the option to have the Check Box play a sound when it is checked, unchecked, or hovered over.

Querying the Check Box state.

There are a few ways to query the state of the box:

Querying Check Boxes

1) One way is to make the Check Box a variable and get the information from that variable.

- With the Check Box selected, see the top of the details panel. Make sure Is Variable is checked. Rename the variable MyCheckBox.

  Discussion: Switching to the graph, notice that a variable named MyCheckBox has been created automatically. Right-Clicking the graph reveals there are a couple of functions that can be used to get the required information:

Querying Check Boxes

1) a) Is Checked

- Open the create menu, disable Context Sensitive, search for *"is checked"*. Add the Is Checked function. Next, connect a MyCheckBox getter to Is Checked > Target Pin.

  Discussion: This function outputs a boolean value of True if the Check Box is checked, and a value of False if the state of box is either Unchecked or Undetermined.

Querying Check Boxes

1) b) Get Checked State returns a variable of type ECheckBoxState Enum, i.e. either Checked, Unchecked, Undetermined.

- Open the create menu, disable Context Sensitive, search for *"get checked state"*. Add the Get Checked State function. Drag from its Get Checked State > Return Value and connect it to a Switch on ECheckBoxState node.

  Discussion: Depending on the state of the Check Box, the execution will flow through one of the three output pins.

Figure 7.28: Switch Check Box States Blueprint

Querying Check Boxes

1) c) The third way to get the state of a Check Box r is to bind it to a variable and get the information directly from that variable.

- In Designer mode, find Appearance > Checked State > Binding > Create Binding. This creates and opens a new function in the graph.

  Discussion: Basically, this saves the step of having to call the Is Checked or Get Checked State functions. The variable will be updated automatically with the latest state of the check box.

Querying Check Boxes

2) The second way to query a check box is by using the `OnCheckStateChanged` event.

- Click `Events` » `On Check State Changed` » `+`. This creates a function `On Check State Changed (CheckBoxName)` and opens it in the event graph.

  Discussion:This event will fire every time the `Check Box` is checked or unchecked and it includes a boolean return value which will tell you which of those states it was just changed to.

Audio

Unreal Engine uses **.wav* files for audio handling, i.e. audio files must be converted into this format in order for those to work with the engine. Imported audio files in turn become Sound Wave assets in the Content Drawer, its icon will have a black background with the actual waveform of that sound shown in white. In case the wav has more than one channel, there will be a separate waveform for each channel.

To combine sounds and/or add effects to them, Sound Cues may be used; these can be identified by their dark grey background along with an image of a speaker and waveform. Unlike the waveform, the Sound Cue icon is generic, i.e. the shape of its wave do not represent the actual sound.

Dragging a sound wave or cue into the level creates an Ambient Sound vector, automatically assigning that asset to the newly crated asset's Sound property. The sound is playable using buttons located in the details panel.

Sound Cues can be edited using the Details Panel > Sound > Edit button, which opens the Sound Cue in the Sound Cue Editor. New sound cues may be generated using the Details Panel > Sound > New.

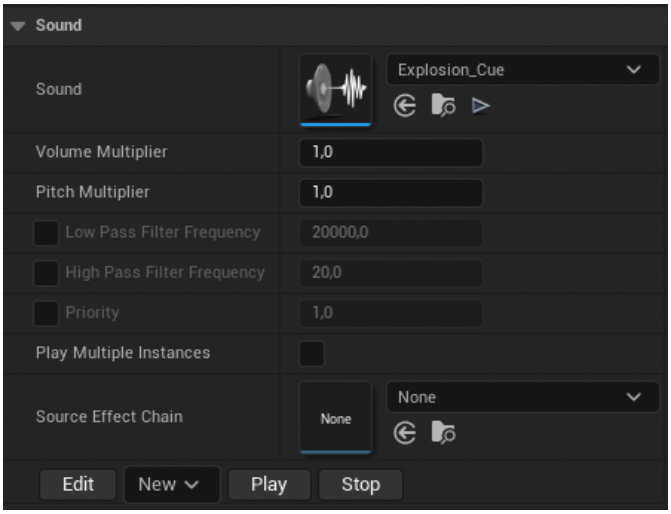

Figure 8.1: Sound Cue Details Panel - Sound Category

8.1 Sound Waves

Some of the important properties are as follows:

- The Volume Multiplier property can be used to in- or decrease the volume of the sound.
- Consequently, he Pitch Multiplier property can be used to adjust the pitch of the sound.
- In case the AmbientSound > Auto Activate property is set to True , the sound is going to play as soon as the actor is created. By default, the sound will be played only once.

In case the sound needs to be continuous, you it needs to be set to loop. With an Ambient Sound Wave, this can be accomplished by setting its Looping property to True. With a Sound Cue, this is done by using a Looping node.

- To edit the properties of a Sound Wave, it must be opened by double-clicking it in the content browser. Here, the Sound Wave Editor » Sound » Looping property can be checked.
- Sound Wave Editor » Format » Ambisonics is a type of sound format that need so be activated if the file is in that particular format.
- Sound Wave Editor » Quality » Compression sets the compression rate. It can have values ranging from 1 to 100, where lower numbers represent more compression and higher numbers represent better quality.
- Sound Wave Editor » Quality » Sample Rate Quality can be used to lower the sample rate if required.
- Sound Wave Editor » Sound » Group Sound groups can be used to group sounds into categories like *Effects, UI, Music* or Voice.

  Discussion: Sound groups are useful for being able to apply a setting to an entire group of related sound waves, instead of having to apply that setting to each individual one.
- Sound Class is similar in concept to Sound Group, except it's more robust, those can be saved, reused, and if required put into bespoke groups.
- Sound Wave Editor » Subtitles provides the means to add subtitles to this audio file as well as edit its properties. Subtitles can be added by clicking the plus sign in Sound Wave Editor » Subtitles » Subtitles. There are lots of options which, unfortunately, are beyond the scope of this document.
- Sound Wave Editor » Subtitles » Mature indicates that there might be strong or adult language involved.

Of course, blueprints can be used to specify when a particular sound should be played.

Triggering Sounds

This tutorial provides an example showing how a sound can be triggered by a trigger volume.

- Add a Trigger Volume to the level. With it selected, open the Level Blueprint.
- Open the create menu, expand the menu and add a Add Event for Trigger Volume » Collision » OnActorBeginOverlap event node. Drag off its output pin and type *"Play Sound"*.

  Discussion: Notice the various options available to play a sound. Also notice, that there are options for *playing* sound as well as options for *spawning* sounds. The difference is that sound that are spawned can be controlled, i.e. they can be canceled or modified, whereas sounds that are merely *played* can not. It will continue playing until it's finished, and if it's set to loop it will continue playing for the rest of the game.
  Sounds can emanate from a certain location in the game, or made to a 2D sound which will be heard at the same volume regardless of a player's location in the game.

  Using a Spawn Sound Attached node, that sound is bound to an actor, so the sound is traveling with that actor.
- Choose the Play Sound at Location node; set its Sound property to Explosion01.

Triggering Sounds

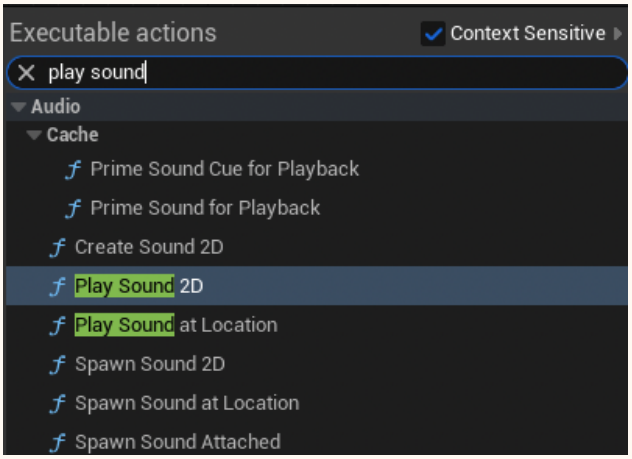

Figure 8.2: Play Sound Options

Now, the location of the sound should in this case be the center of the trigger volume. Hence,

- Obtain a reference to the trigger volume using the create menu, add a GetActorLocation node. This will automatically connect to the trigger volume, which is why we selected that volume before the Level Blueprint has been invoked. Connect Get Actor Location > Return Value to Play Sound at Location > Location Pin.

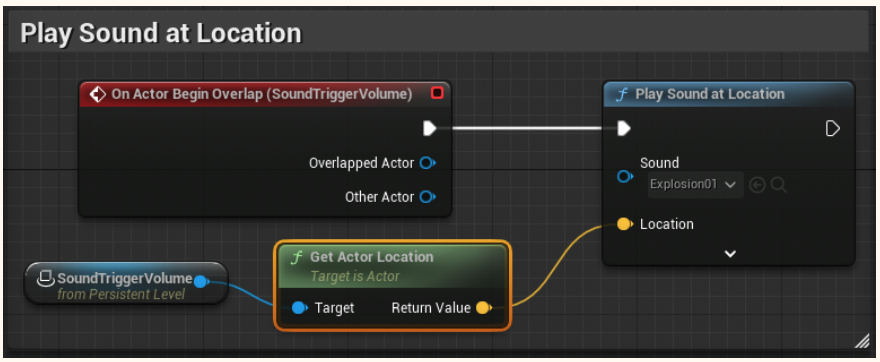

Figure 8.3: Play Sound at Location Blueprint

- Compile and save.
- If the game is played, an explosion should be heard when entering the trigger volume.

8.2 Sound Cues

Sound Cues use existing Sound Waves to create new sounds by combining the waves and/or adjusting their properties or, not to forget, adding effects to them.

Sound Cues

This brief demo gives a short overview on how to work with sound cues and discusses some frequently used options.

- Click Content Drawer » + Add » Sounds » Sound Cue and give it a descriptive name. Double-Click to open it in the Sound Cue Editor.

Discussion: The Sound Cue Editor uses a node-based graph very similar to blueprints, however, it uses its own specialized nodes instead of the types of nodes that are available there. The basic idea is to with one or more nodes on the left representing Sound Waves, connecting those waves to nodes in the center, combining and/or modifying them so that eventually the output to the right is actually the sound that this cue finally plays.

Sound Cues

- Add one or more Wave Player node by dragging it into the graph from the palette to the right.

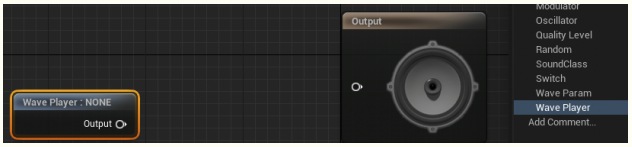

Figure 8.4: Wave Player Node

- Select that and use the Sound Wave property to specify which sound wave this node shall output. Select Explosion01 (included in the Stater Content package).. Note, that if connected with the output node, this would yield a valid cue.

Te Palette contains several nodes that can modify the sound that gets input into it and then output an altered version of that sound. For example, the Looping node will take a sound as input, and output that sound as a loop. In the details panel it can be defined how often the sound should loop or if it shall loop infinitely.

- The Delay node can be used to add a delay before a sound is played. Each time the Delay node is activated, the amount of delay will be a random value between Delay Min and Delay Max.
- The Doppler node can be used to add the Doppler effect to a sound. The Doppler Intensity property can be used to specify how pronounced this effect should be, with higher values increasing the effect.
- The Modulator node can be used to play a sound at a random pitch and volume each time it's played. This can be used to make the sound sound slightly different each time so as not to get repetitive. The range of the random value generated can be set in the details panel.
- The Oscillator node can be used to add a continuous modulation of volume and pitch within a single instance of a sound being played. The first two properties are used to enable the modulation of the volume and/or pitch. Amplitude refers to the height of the wave, with larger waves producing louder volumes. Frequency affects the pitch of a sound, with higher frequencies resulting in higher pitches.

In addition to nodes that alter sounds the palette also contains many useful nodes for combining sounds:

Sound Cues

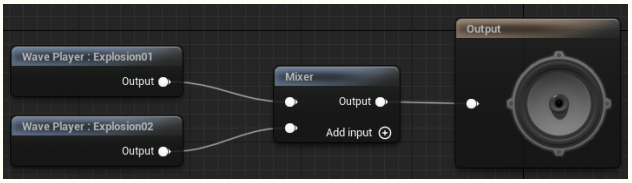

Figure 8.5: Simple Mixer Cue

- Add another Wave Player node and set its sound to be Explosion02.
- Add a Mixer node. A mixer node takes two or more sounds as input and outputs all of those sounds being played simultaneously.

If more input sounds are required, additional input nodes can be added using the Add Input icon.

In the mixer's details panel, several options are available. Most notably, the volumes of each input wave can be adjusted.

Conversely, the Concatenator node also combines two sound waves, just not concurrently, but rather one after the other.

Random nodes randomly output one of it input sounds. The probability that determines which sound should be played can be set using the Weight property.

Also note the availability of a Branch node as well as a Switch node.

8.3 Attenuation

Attenuation is a scientific term that refers to the reduction in strength of a signal. Thus, in the case of an audio signal, this refers to the decrease in volume that occurs due to distance. In *Unreal Engine*, attenuation properties of the sounds in a game may be configured to match the desired behavior of the sound.

Sound Attenuation

This demo gives an overview on options and usage of the attenuation features.

- Add a Sound Cue to the level, creating an AmbientSound actor playing that cue when the level begins.

Sound Attenuation

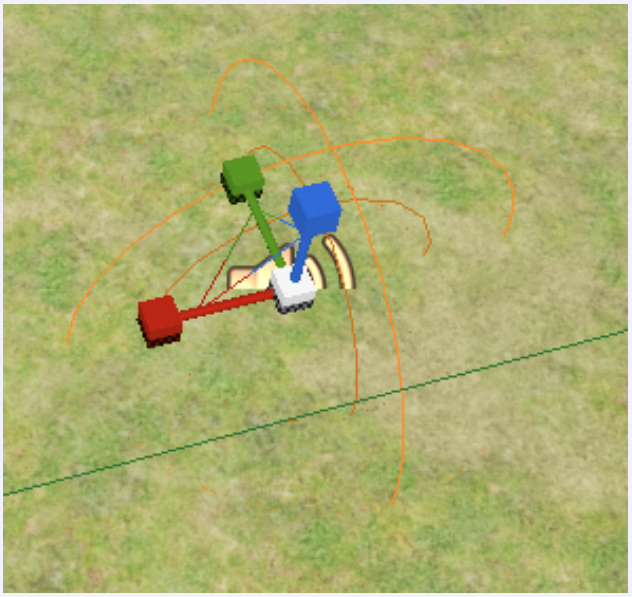

Figure 8.6: Attenuation Radii

Discussion: Notice the actor is depicted with a sphere around it. Two spheres actually, an inner sphere and an outer sphere. At any point within the inner sphere, the sound will be heard at 100 % volume. Going from the outer edge of the inner sphere, to the outer edge of the outer sphere,the volume will decrease from 100% to zero. The size of these spheres can be edited in the details panel.

- Check Attenuation » Override Attenuation.
- Modify the Attenuation (Volume) » Inner Radius property. Notice, that the outer sphere changes as well.

  Discussion: In order to change the size of the inner sphere, the area in which the sound is heard at full volume, the Attenuation (Volume) » Inner Radius property may be edited. As the inner radius is changed, the size of the outer sphere changes as well. This is because the outer sphere is defined by the Attenuation (Volume) » IFalloff Distance, the distance from the edge of the inner sphere to the edge of the outer sphere, as opposed to being defined by absolute size. So if the FalloffDistance is changed, the size of the outer sphere changes accordingly.

Notice the possibility to change the Attenuation (Volume) » Attenuation Shape. By default, spheres are used, as this is the most natural way that sound will attenuate. However, in a scenario where the player character is located in a room or building, a square attenuation shape appears more realistic.

In the area described by the FalloffDistance will go from fully up to zero, as discussed. The rate at which this happens can be adjusted by setting Attenuation (Volume) » Attenuation Function used to define the Attenuation Curve. By default, this will use a linear curve, meaning the volume will decrease evenly. Using the Logarithmic curve, the volume will decrease more rapidly at first, then the rate of decrease will slow as the sound approaches the bounds of the attenuation shape. The LogReverse curve, as its name indicates, is the reverse of that. The volume will decrease slowly at first, then more rapidly. The Inverse curve is similar to the Logarithmic curve except the volume decreases extremely rapidly at first, then very slowly for the remainder of the distance. The NaturalSound curve is somewhere in-between the Logarithmic curve and the Inverse curve, and is supposed to represent the most natural attenuation curve that sounds have in the real world.

Application of Attenuation

Attenuation can be applied on different levels. A the highest level, attenuation assets can be created, saved

and subsequently applied to multiple sound assets. To create a new attenuation, click Click Content Drawer > + Add > Sounds > Sound Attenuation. Once edited and save, the newly created attenuation can now be applied to a sound cue, by setting the latter's Attenuation Settings property.

Note, that the attenuation configured this way for a Sound Cue can be overridden using the Attenuation > Override Attenuation property or modifying the respective properties in the details panel.

Packaging

Packaging is the process that prepares your game to run on a specific platform.

There are several steps in the packaging process:

1. All Blueprints will be converted into *C++* source code that will then be compiled.
2. All of the project's assets, i.e. meshes, materials, audio, etc. are converted into a format that can be read by the target platform, a process known as *"cooking"*.
3. The compiled code and cooked assets are bundled together into a package of files that can be used to run or install the game on the target platform.

9.1 Preparing

Before a game can be packaged some precautions must be taken:

Set a default map: The default map is the level that will be loaded when the game first starts. Without a default map, nothing will get loaded and the player only sees a black screen when the game runs. The default map can be set in Toolbar > Edit > Project Settings > Maps & Modes > Default Maps.

9.2 Packaging the Game

To package the game, click Toolbar > Platforms, hovering over the platform you want to package for, then click "Package Project". The developer will then be prompted to choose a folder to save the packaged project to. Once chosen, the packaging process will start. Packaging runs in the background, thus the editor is still usable.

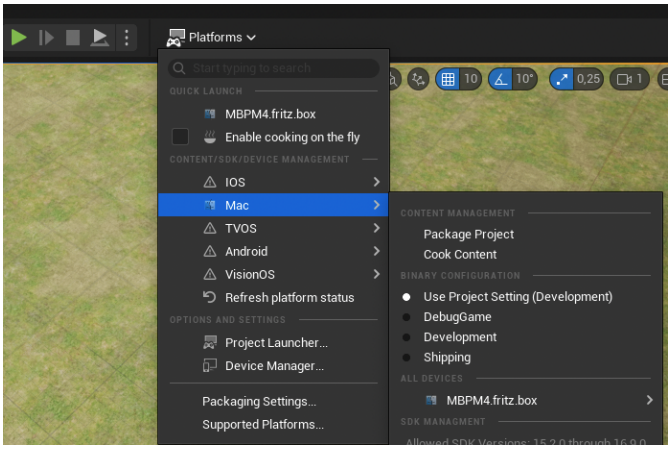

Figure 9.1: Select Game Platform

Notice, in the bottom-right corner of the screen a small windows opens, displaying the packaging progress. This window also features a cancel button. In addition, a window

Unreal Engine has some advanced settings you can configure for the packaging process.
These can be found either in Edit » Project Settings » Project » Packaging or in Platforms » Packaging Settings

- Project » Build: Tells the editor under which circumstances it should build the project as part of the packaging process. In most cases, you can just use the default setting.
- Project » Build Configuration: In *C++* based games, set this to DebugGame for debugging. Otherwise, i.e. the game is blueprint-based, the usual option is to set this do Development. To build a game for distribution to end users, the usual choice is to set this to Shipping. Finally, Test is something in between development and shipping. It configures everything just like a shipping build but still allows for some console commands.
- The Staging Directory is the target folder the packaged project is saved to. This is the same directory the user is asked to select when starting the packaging process.
- The Full Rebuild property specifies whether all of the code should be compiled or just the code that has been modified since the last build. For shipping builds, this should be set to True. If targeting the *Android* or *iOS* platform, the Distribution property should be set to True.
- Packaging » Use Pak File: a *.pak* file is a file containing all of the assets of a game. If set to False, all the assets will be individual files. If this is set to True, all assets will be compressed into one *.pak* file.

  Discussion: If using a *.pak* file, adding some security options is recommended:

  – In the Project » Encryption category encryption and signing keys may be generated.
  – These can be used to encrypt and/or sign the pak file.
- In Project » Packaging » Create compressed cooked packages: If set to True, the *.pak* file will be compressed.

  Discussion: Whether or not *.pak* files should be compressed largely depends on the target platform. If the target is Xbox hardware, *.pak* files should always be compressed, improving loading times. If the target is a Nintendo Switch, testing is recommended. Sometimes compressing improves load times, sometimes it will have the adverse effect. For Playstation and Oculus platforms, *.pak* files should never be compressed, since Playstation hardware already uses compression and the Oculus is unable to process a compressed *.pak* file. For the Steam platform, compressed *.pak* files result in less space being needed on the end-user's device, but longer download times when patching, and is therfore a matter of personal preference.

Procedural Landscape Creation

Multiplayer

Programming Using C++

Animation

Cookbook

Appendix

List of Figures