

The **Cave Engine** Bible

Official Game Engine Documentation



Written by Guilherme Teres Nunes
Uniday Studio - 2025

Uniday Studio – 2025 - All Rights Reserved

Confidential. Redistributing this book is not allowed.

The **Cave Engine** Bible

Official **Documentation** (v1) for Cave Engine **1.2.0**

Written by **Guilherme Teres Nunes**

Introduction

Cave Engine is a Simple, easy to use, 3D desktop Game Engine that is scriptable in Python. You can use it to make any type of games you want and release them commercially.

You can expect a seamless development process with cave, with pretty much zero loading time: it does not require any Shader or Code compilation and asset handling is as fast as possible.

In this book you will find the complete guide to get started with the engine. As well as the documentation of every part of it to improve your understanding of the tool. We expect that you have a basic understanding of game development and Python scripting in order to proceed, but it's not a set rule.

If you have any questions or a comment, don't hesitate to contact us for more information or help.

Redistributing this book is prohibited and using cave engine is subject to its end-user license agreement. Make sure that you all valid legal copy of Cave Engine before proceeding with this documentation.

Table of Contents

Introduction.....	3
How to Contact the Dev?.....	8
1. Your First Time with Cave Engine.....	9
1.1. How to Run Cave Engine.....	9
Troubleshooting:.....	11
1.2. The Project Manager.....	12
1.3. The Editor Basics.....	14
[Tab]: 3D View (or Viewport).....	15
[Tab]: Properties.....	17
[Tab] Scene Graph.....	19
[Tab] Asset Browser.....	19
[Tab] Console.....	21
[Tab] Settings.....	21
[Tab] Timeline.....	21
Hidden Tabs by Default.....	23
[Tab] Timeline Preview.....	23
[Tab] Text Editor.....	23
[Tab] Finder.....	24
[Tab] Statistics for Nerds.....	24
[Tab] Audio Monitor.....	25
[Tab] Joystick Preview.....	25
[Tab] Profiler.....	26
1.4. Manipulating Entities.....	27
1.5. Entity Names and Activity.....	33
Entity Activity:.....	33
1.6. Entity Properties and Tags.....	36
1.7. Entity Components.....	38
[Cmp] Transform Component.....	39
[Cmp] Mesh Component.....	40
[Cmp] Animation Component.....	43
[Cmp] Animation Socket Component.....	45
[Cmp] Camera Component.....	46
[Cmp] Light Component.....	46

[Cmp] Particle Component.....	46
[Cmp] Character Component.....	47
[Cmp] Rigid Body Component.....	48
[Cmp] Physics Constraint Component.....	49
[Cmp] Python Component.....	49
[Cmp] Python Code Component.....	51
[Cmp] Custom Camera Controller Components.....	52
[Cmp] Vehicle and Wheel Components.....	53
[Cmp] Custom Controller Components.....	54
[Cmp] Audio Player Component.....	54
[Cmp] UI Element Component.....	55
[Cmp] Terrain Component.....	59
1.8. Terrain Creation Techniques.....	63
Terrain Texture Paint:.....	65
Custom Terrain Shader:.....	66
Painting the Terrain Masks:.....	68
2. How to Architect Your Game in Cave.....	69
2.1. Understanding Levels and Objects.....	69
2.2. Understanding Scenes and Entities.....	70
Understanding Entities.....	71
Scenes are meant to be UNIQUE (not copies).....	72
What about shared objects?.....	72
2.3. Entity Templates.....	73
What's that?.....	73
2.4. Creating a Scene and Entities.....	74
2.5. Creating and Editing an Entity Template.....	75
2.6. Instantiating Entity Templates.....	76
2.7. Modifiable Entity Template Properties.....	77
3. Understanding the Engine, Editor and Player.....	78
3.1. Cave's Main Loop.....	82
3.2. Understanding Cave's Internal Signals.....	84
3.3. The Start and End methods.....	85
3.4. Understanding all the Main Methods.....	87
[Method] Start.....	87
[Method] First Update.....	87

[Method] Update.....	88
[Method] Late Update.....	88
[Method] Paused Update.....	89
[Method] Editor Update.....	89
[Method] End.....	90
3.5. Understanding the Internal Methods.....	91
[Method] Update Physics.....	91
[Method] End Update.....	92
4. Cave's Audio System.....	93
5. Shaders and Materials.....	95
5.1. Cave Shader Programs:.....	95
Shader Override Ordering.....	95
Uniform Exposition:.....	96
5.2. Shader Naming Conventions:.....	97
Material Uniforms.....	97
Editor: Uniform Visibility.....	98

How to Contact the Dev?

Please, use the e-mail guilherme@unidaystudio.com.br to contact me if you have any questions or comments about the Engine. You may also use our official discord servers.

PLEASE CONTACT ME, without any hesitation, if:

- **THE ENGINE CRASHES.** In this case, try to send me as much information as possible: (1) What were you doing when that happened? Try to give me as much information about it as possible, such as a step-by-step, (2) Did any ERROR MESSAGE appeared? If so, send me the entire thing. (3) Is this crash recurring or it only happened a single time? (4) What are your computer hardware and OS specs? (5) If you think it may help, do you mind sharing the project file with me so I can debug it?
- **YOU FOUND A BUG.** Just like the previous step-by-step, try to send me as much information as possible, mainly HOW TO REPRODUCE the bug, so I can fix it.
- **YOU DISLIKED SOMETHING.** Explain me why and what could I do to improve it.
- **YOU LIKED SOMETHING!** Yep, positive feedbacks are always helpful.
- **YOU THINK THAT SOMETHING IS MISSING.** Explain me what do you think is missing and why. I promise that I'll think about it.
- **YOU HAVE A SUGGESTION.** Let me know! :D
- **YOU FINISHED A GAME/PROJECT USING CAVE!** Don't worry, I don't want to charge you anything. I just would love to see what you've done using it.

It's very important that we keep a good communication, since that's the only way I can improve the engine and make it better for all of us. Thank you for the comprehension.

1. Your First Time with Cave Engine

Welcome to Cave Engine! In this chapter I'll provide an in depth guided tour of the basics of the engine, to cover your first time in it and all the basics you need to know such as project creating, setup, basic controls, behaviors and how everything works for you to create your games. So with no more talking, let's get it started!

1.1. How to Run Cave Engine

The first thing you need to do when setting up Cave is to download the Engine with the provided link. If you don't have a link to it, please contact me for more details on how to download the engine. I'm assuming you have it already, so let's move on:

- **STEP 1:** **Extract the Engine to a Folder** exclusively made for it.
- **TIP:** Do not extract it in a folder with other contents, because it may cause some unknown conflict.

***Important:** There is a known issue in cave regarding some specific User Pathes, so make sure all the requirements bellow are satisfied, otherwise it may not run correctly:*

- **Do NOT** extract the engine on your **Desktop** or inside a **Google Drive, One Drive or Similar Folder**. It's known that this may interfere with the engine.
- Make sure the Path you extracted **does NOT contain special characters**, such as Portuguese, Russian or other language characters: **“ç, á, ã, л, д, etc”...**

I recommend you extracting the Engine at the “[C:/](#)” directory if you're on Windows.

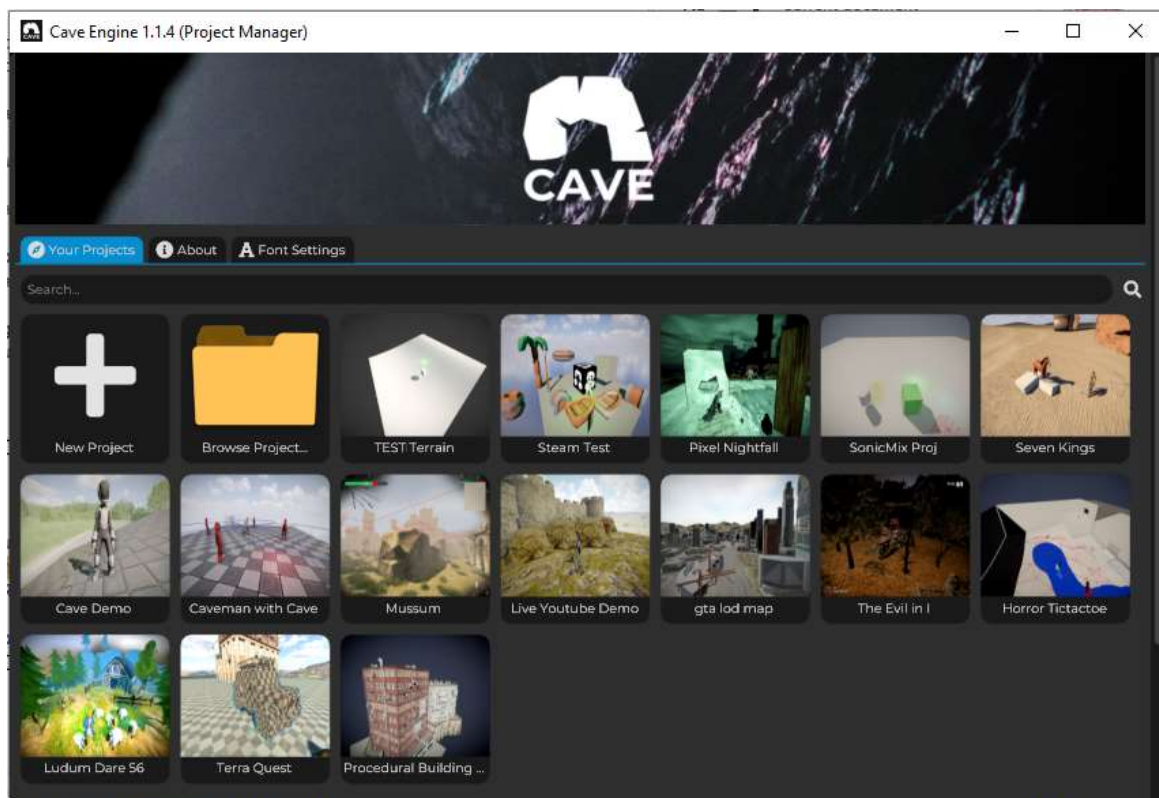
Once you extracted it to the folder, you will probably find the following files inside the folder:

Editor	07/03/2024 03:20	
Lib	01/02/2024 14:07	
assimp-vc141-mt.dll	25/08/2022 12:17	5.727 KB
Cave Editor.exe	09/03/2024 16:12	7.022 KB
Game.exe	09/03/2024 16:11	6.702 KB
python37.dll	08/07/2019 20:35	3.661 KB

The file names may be slightly different or it may not have the DLLs if you're not on Windows, but it's not a problem in this case. What you need to make sure is that it have two Folders (**Editor** and **Lib**) and also two executables (**Editor** and **Game**),.

Note: The Engine NEEDS all those files in order to work and the executables needs to be located in the same folder as the rest of it. Do not delete, move or rename the files.

To Run the engine, simply launch the **"Cave Editor"** file. You can create a shortcut to it in your desktop or pin it to your taskbar if you want to. If everything is all set, you should be able to see the Project Manager Window:



Troubleshooting:

If you don't see the window, it appears and crashes or similar issues, make sure that:

- ✓ The **Path where the Engine is located** is according to the previous Specifications.
- ✓ Your **Antivirus did not accidentally blocked** or excluded any of the Engine files.
- ✓ Your **Graphics Card Video Drivers** are up to date! **(IMPORTANT!)**

Those are the most common issues. If you checked all that and the engine still doesn't open, please don't hesitate to contact us for further help.

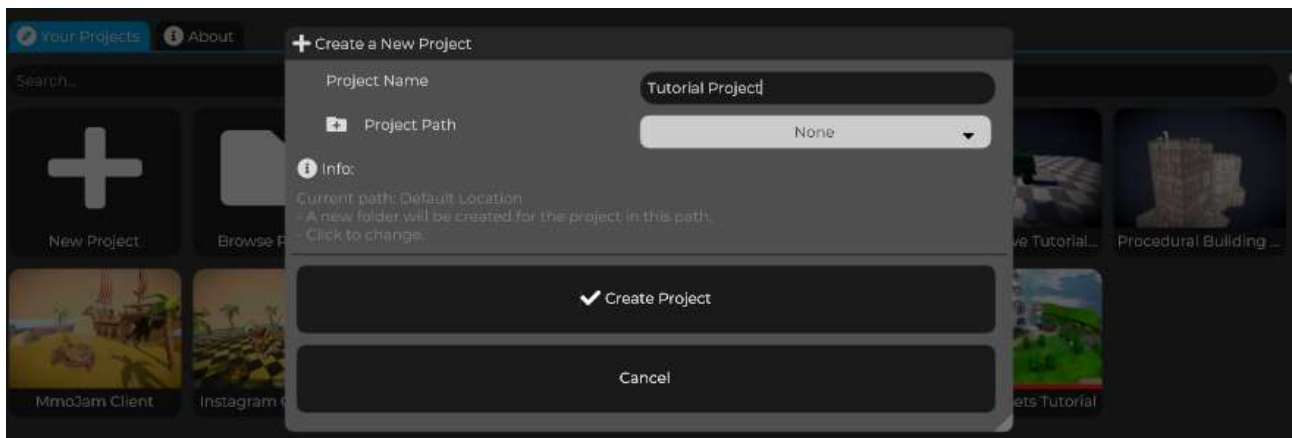
1.2. The Project Manager

Once everything is all set, let's understand the Project Manager a bit better. This is where you'll be able to see all your previous **Cave Project** and Create and/or Locate new ones. This is what the ***"Your Projects"*** tab is meant to do.

In the ***"About"*** tab you can access some extra Cave Engine Details, such as the current Engine Version, links for the Engine License (EULA), documentation, our Discord Servers and so on.

Back to the ***"Your Projects"*** tab, if you have existing projects already, you can hover the mouse over it to see details (such as last modification date, the Engine Version used in it and the Project Path on disk. Right clicking it shows some available actions for the project. If you have a project from a different Engine Version showing, you won't be able to open it in the current one, so a Red bar will be displayed bellow the Project Thumbnail to indicate incompatibility and they will appear in the "Invalid Projects" region.

By clicking on a Project, it will open it. If you click "Browse Project", it will open a File Dialogue for you to locate an existing cave project in our Computer. Finally, by clicking at "New Project", it will open a new project modal menu for you to set some settings:



It is very straight forward: you specify the **Project Name** and click **Create Project**. If you want, you can also change where the Project will be saved on disk by expanding the "Project Path" option.

If “None”, the engine will use its default path, depending on the Operation System you’re on.

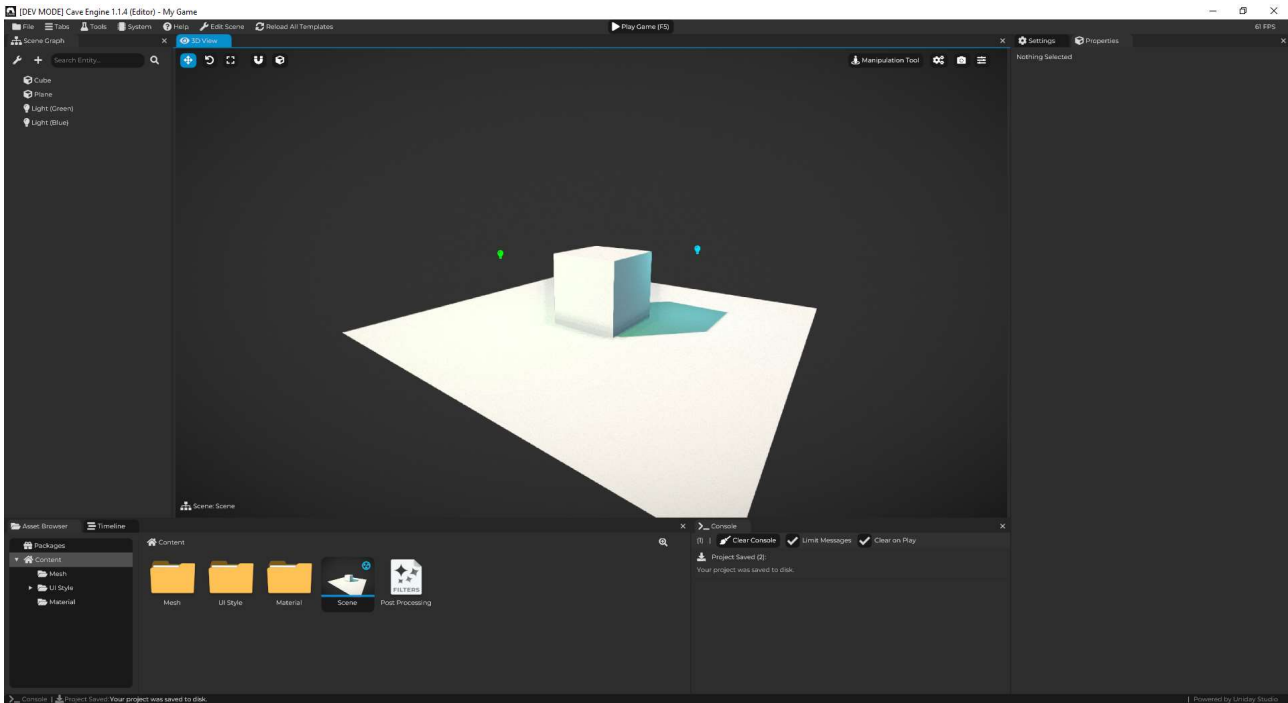
→ On **Windows**, it is “*%AppData%/Cave Engine/My Projects/*”

→ On **Linux**, it’s the same Folder (“*Cave Engine/My Projects/*”) but at **HOME**.

Remember that you can always open where the Project was saved, either by right clicking on it in the Project Manager, or by going to “**File > Open Project in Explorer**” menu once the project is loaded.

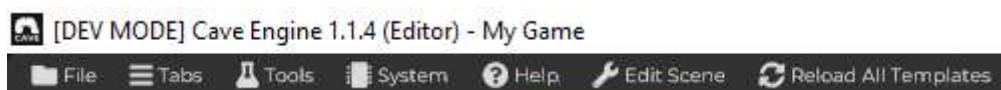
1.3. The Editor Basics

After creating your first project, you should be able to see your **default New Project** and **the Editor Windows** open:



Notice that the Engine is divided into Tabs, and each of those tabs could be rearranged by click and dragging them to anywhere in the Window, allowing you to customize the interface as you like it. The changes made are persistent and shared across projects.

It's also good to know that in the top region of the window, you will find a Menu Bar with a lot of useful options, such as **File**, to save your project, **Tabs**, to reopen any Tab you accidentally (or intentionally) closed and more. The Project Window Title bar (on top) will show you the current version of the engine followed by your project name.



Before we jump into the 3D View and start making some amazing 3D things, let's understand a bit more the main Tabs displayed by default in the Engine and what they are used for:

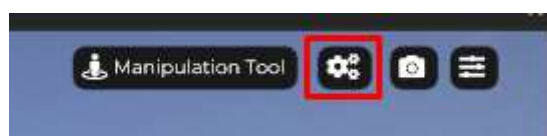
[Tab]: 3D View (or Viewport)



This is where you will see your entire game through it. It will always render the currently opened **Scene** or **Entity Template** (you will learn about all that later). I know you're really excited to start moving around the 3D View, so here is the **basic controls**:

While mousing over the 3D View, click and hold the Right Mouse Button to activate the Mouse look. You will be able to move the camera around as if you were playing a First Person Shooter (FPS) game. While in this mode, you can use W, A, S, D to move around (forward, left, backward, right). The keys Q and E also moves you down and up, respectively.

If you're not happy with the movement speed, simply roll your Mouse Scroll up or down while holding the right button to adjust it. If you want even more adjustments, such as the mouse look sensitivity, the camera acceleration, reduction and also manipulation settings, click in the Viewport Settings icon located in the top right corner of this tab to see the extra configurations:



You can also control the camera like this:

- **Ctrl + Right Mouse:** Zoom in/out (moves forwards of backwards, actually)
- **Shift + Right Mouse:** Moves the View

If you're familiar with Blender, similar camera controls to it could be found in the Num Pad. Here is the complete Viewport controls shortcuts list:

Shortcut	Description
W	Toggle to Move Gizmo .
E	Toggle to Rotate Gizmo .
R	Toggle to Scale Gizmo .
F	Focus the Viewport Camera to the Active Selected Entity.
Delete	Deletes all Selected Entities.
G	Place the Selected Entities to the surface Position currently hovered/pointer by the mouse. If you hold Shift or Ctrl while pressing G, it will also align the Entity with the surface Normal.
Alt + G	Same as just G, except that it first duplicates the Entities and then apply the transformation to the new ones.
Alt + W	Reset Position .
Alt + E	Reset Rotation .
Alt + R	Reset Scale .
Alt + P	Removes the Entity Parent (local, without affecting its Transform).
Shift + A	Opens the new Entity popup menu.
Shift + P	Sets the Currently Active Entity to be a Parent of the other Selected Entities.
Shift + D	Duplicates all the Selected Entities.

Shortcut	Description
Ctrl + D	Same as Shift + D.
Ctrl + C	Copy the Active Entity to the clipboard.
Ctrl + V	Pastes the Copied Entity from the clipboard to the current Scene.

Notice that the Viewport is set to the Manipulation Tool by default, which is the tool that allows you to select Entities and manipulate them. There are other tools, such as the Terrain Tool that works in the Viewport.

Note: *The viewport Shortcuts **are sensitive to which tool you are currently using**. Those are for the main (Manipulation) Tool.*

I'll talk more about the 3D View later... But let's keep going to know the rest of the Tabs first to finish the overall view.

[Tab]: Properties

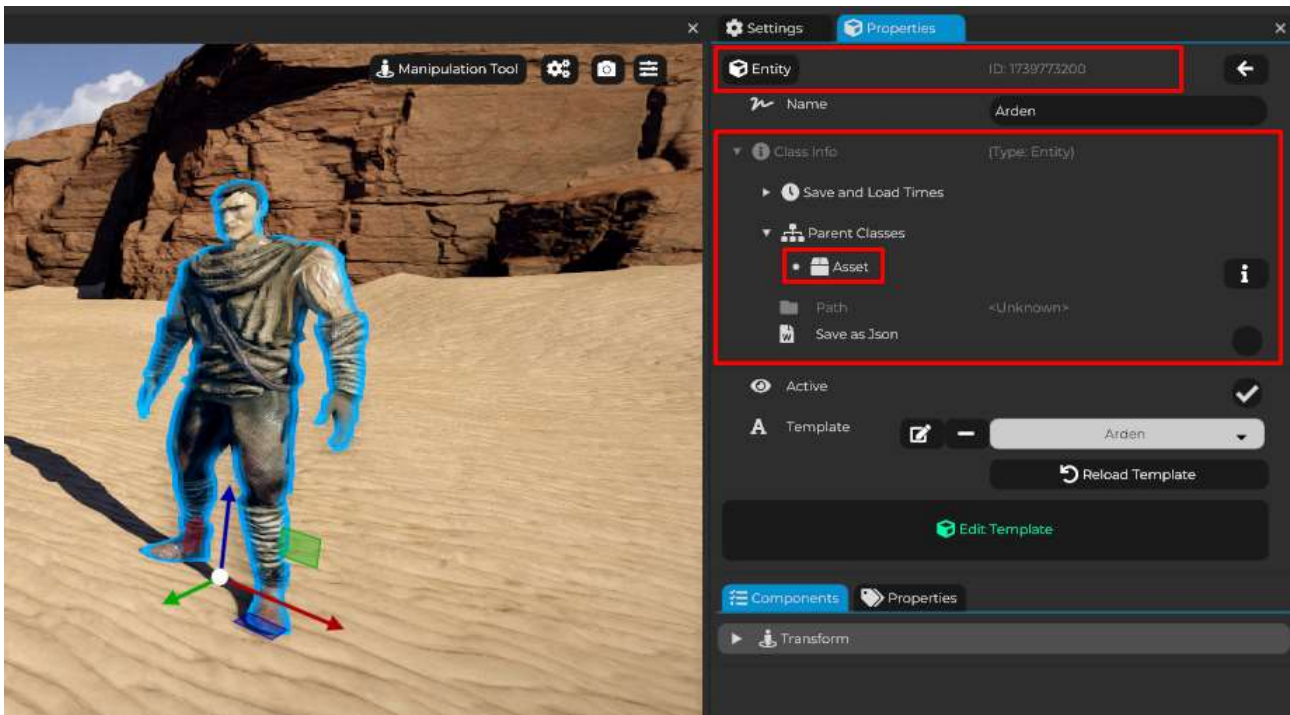
This is the one located on the Right side of the screen by default. As the name suggests, it is meant to show Properties. What Properties? It depends on what you have selected! It shows nothing by default because there is nothing currently selected, but try to mouse over an entity in the Viewport and Left Click it. It will select the entity and show all its properties in this tab. Same thing will happen if you select something in the **Scene Graph Tab** or in the **Asset Browser Tab**, etc.

It also have a handy back button in the top right corner, so if you select something and then select something else by accident, you can go back to your previous selection by clicking in it and keep editing your properties.

This is also a good time to introduce you the Asset concept: Everything you see in the Properties tab is considered an Asset. In cave, almost every object and data format inherits from this Asset abstraction. Being an Asset means that it can be Serialized (saved and loaded) and displayed in the Properties tab for you to modify.

Notice that this is an internal concept, so we are talking about the Editor capabilities here, not necessarily the game options, since this one is project specific and will rely on how you structure things up.

For each selected Asset that appear in the Properties Tab, you can expand the Class info to take a quick glance at its internal formats. For example, if you select an Entity in the 3D world, you will see this:



This 3D Character is a Templated Entity (more on that later) and most importantly: you can see its base class name on the very top of the Properties tab and also its unique internal ID. You can also find and modify the Asset Name.

By expanding the Class Info, you will see its type once again and some debug information, such as the parent class (notice that the Entity inherits from the Asset class, as I said), Path (is Unknown because the Entity is not stored on its own asset file, but inside the Scene or Entity Template) and even how long it took to Save and Load (useful to find bottlenecks if your game's loading time is too long).

After those internal/debug information, you will find the Asset specific settings bellow. Those will vary according to each asset type. The internal data may be useful for you because it helps optimizing the game and also when it's time to start writing Scripts for the Game Logic.

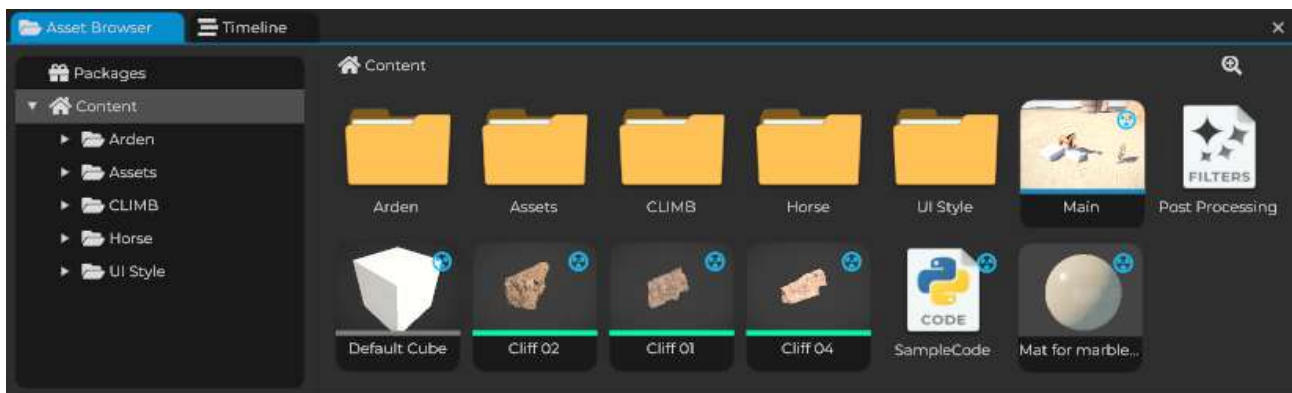
I'll explain what each Asset Type in the engine have as properties and what they all do as we talk about them. But now you know that whenever you want to tweak some settings of properties of something, it will probably be displayed here.

[Tab] Scene Graph

Since you already saw the **3D View** basics, you probably already noticed that it is composed of a **Scene** and **Objects** (**Entity**, in Cave terms) in it. For the currently opened Scene (or **Entity Template**, you'll see more about this one later), all the entities in it will be displayed here. You can also use this tab to establish parenting between the entities by dragging them on top of others, you can click to select a given one and also right click them for a bunch more options.

If you click in the Wrench icon in the top left corner of this tab, it will allow you to **edit** the current **Scene** (or **Entity Template**) in the **Properties Tab**. You can also do that **by left clicking the sky in the 3D View**.

[Tab] Asset Browser



Here is the Brain, the “Memory”, of your Project. It's where you will find all* the Assets that you currently have to compose your game. You will be able to move them around, rename, organize them into folders, import new or deleting existing ones, etc.

**Some Assets are built within other Assets, such as the Entities that, as you saw, are usually inside the Scene Asset. So they don't appear here.*

This tab is split into two: in the left, you will find a Drop Down Tree representation of your folders. It will not show Assets, only Folders, but it's a good easy way to navigate across your project. If you pay close attention, you'll realize that it does

provide some hints regarding what's inside the folders, such as having an expand arrows if it have child folders or be either closed or opened if it have elements inside of it or not. The left side is meant for quick access.

Similar hints are also displayed on the right representation. Talking about the right side, it is where you will find the Assets itself (and also the folders, it's the complete version). Most of the Cave Assets have custom thumbnails with a preview to facilitate your life and also descriptive colors right above their names and bellow the thumbnails.

- ➔ **Light Grey means it's a Mesh,**
- ➔ **Blue means it's a Scene** and
- ➔ **Green, an Entity Template.**

They can also contain indicative Icons, such as the **Blue Radiation Symbol** you can observe in the last image, that means that this given asset or folder is marked as "Dirty", indicating that it have local changes not yet saved to disk. The dirty flag (and most other things we are discussing here) can be accessed through Python.

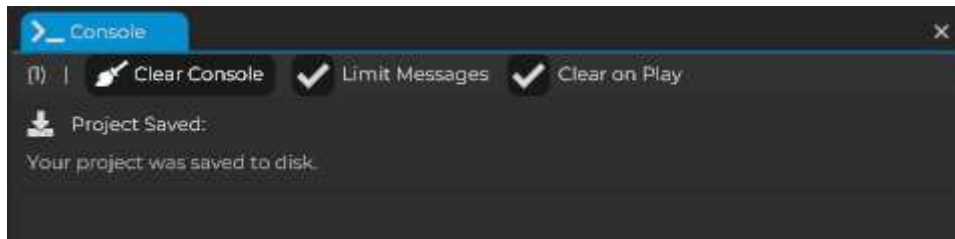
Note: *It's not guaranteed that a dirty asset actually have changes in it, may be a false positive, so don't use this as a rule. But the engine tried to guarantee you that all modified assets are properly marked as dirty.*

TIP: **Ctrl + S** saves all the Dirty Assets, **Ctrl + Shift + S** saves everything, regardless if it's dirty or not. If an asset keeps showing the dirty icon after you save, it is probably because the asset is currently selected, such as the current scene. But that's fine!

In the right side, If you right click an empty space, a folder or an Asset, you will see a popup menu with useful options, such as the ability to rename the Asset/Folder you clicked, dissolve folders, duplicate assets or create new folders, assets and/or import external ones (if you click on an empty space).

[Tab] Console

If you're reading this while experimenting with the engine at the same time (*which I recommend, it's good for learning*), you probably already saved the project once (Ctrl + S). And if you did that, you noticed that something got written to the console:



And this is the purpose of the Console Tab: Show you important or useful information. While a “Project Saved” message is not particularly important, this is also where the engine will communicate to you about errors, warnings, or if something is wrong. So It's important to keep an eye on it at all the time.

When you start writing your own game code, this is also where the **print** outputs will go.

[Tab] Settings

You won't use this tab much, but when you do, it will probably be a special day.

Here is where you will find exporting options to your game to the world!

Not only that, but you'll also find some global Rendering and Game settings, such as the resolution used by the engine (by default it uses the user's desktop resolution, but you can change that if you want), an option to set custom Post Processing Filters, custom Shader for Cave to use and replace the entire Default Shader code, etc.

[Tab] Timeline

Cave Engine have a built in system for **Cutscene** creation (in game animations) that we call “**Timeline**”. It allows you to create any number of cutscenes you want to (by creating different timelines) and play them, including multiple timelines at the same time. On each timeline, you will be able to keyframe multiple Entities,

Components and Component Channels and also change cameras, run scripts, etc. If you open this tab, it will be empty by default because you are not editing any timeline. You can right click the Asset Browser and create a new Timeline in it, then select it and edit the Timeline (or double click it). Then you will be able to start creating your cutscenes. You will see that it could looks like this:



Note: Please notice that this is **NOT** meant for you to replace a DCC (Digital Content Creation) software such as Blender or Maya to create **Skeletal Animations**. The timeline system in Cave is for you to put together already existing animations, so it currently doesn't support individual bone keyframing.

Timelines are not only a Cutscene creating tool, they could be useful to animate real time events and behaviors in your game that happens simultaneously, such as a Helicopter flying by, a building exploding, an enemy campsite appearing, etc. That's because when playing a Timeline (that's done through Script), you can choose if you want it to pause the Scene or not and most importantly: you can play multiple Timelines at the same time!

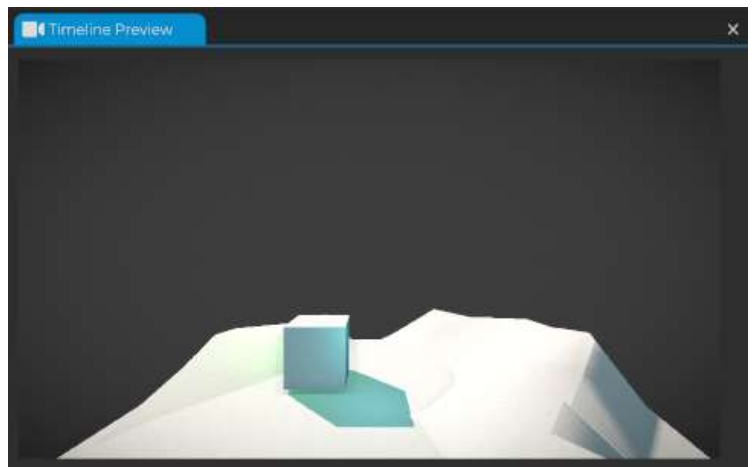
I'll not go much further about this system here just yet, but it's always good to know that it exists in the engine and it can make your life way easier when animating something.

Hidden Tabs by Default

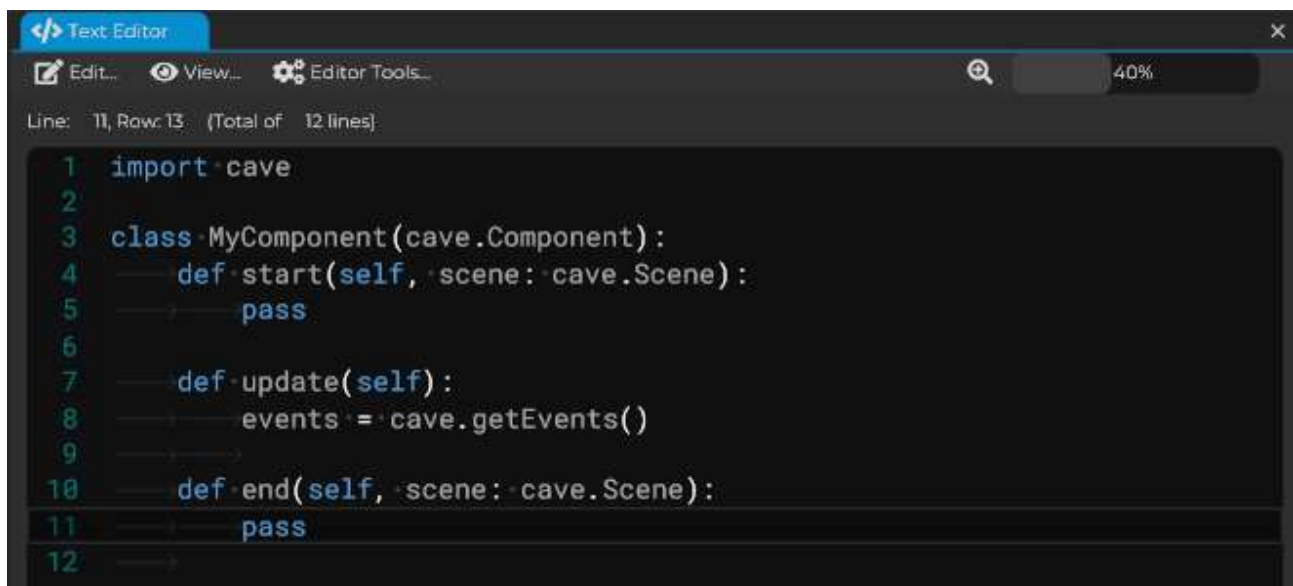
All the tabs you saw until now are enabled by default, meaning that they are all supposed to appear in the Editor right away, as soon as you open your Project. If you click the close button, the tab will become hidden. If you go to the top menu **“Tabs”**, you will see the list of all the Engine tabs and which ones are active or hidden. That way you can reactivate a tab you closed. You will also see **more tabs that you didn't see previously** and we will talk about them now.

[Tab] Timeline Preview

When editing a Timeline (in the Timeline Tab), you may want to add a camera to it. The Timeline Preview Tab exists to show you a preview of what's the timeline's active camera is seeing, making it easier for you to adjust your angles.



[Tab] Text Editor

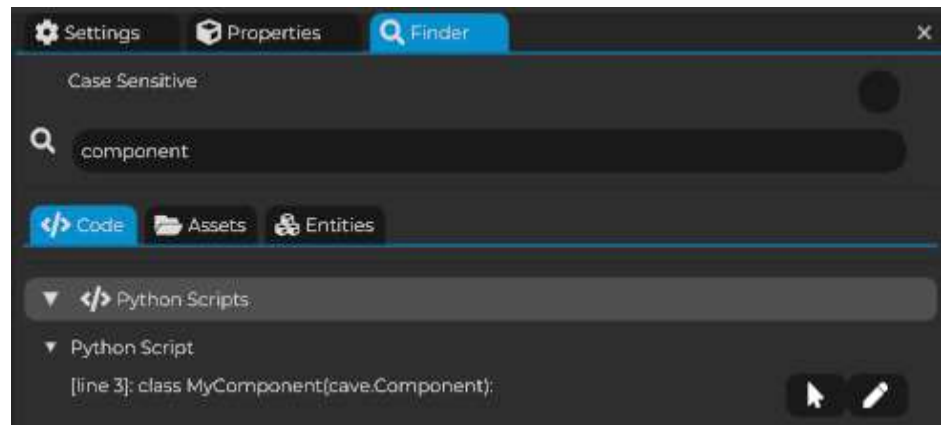


If you create a new **Python Script** in the Asset Browser and double click it, you will be able to edit its contents in this Tab. This Tab opens by default for you when you try to edit any code (Python, Shaders, etc) and closes for you when you're done.

As I said, this is not limited to a Python Script: This same tab will be useful for you to edit Documentation Assets (markdown), Shader Programs (Vertex and Fragment Shaders, in GLSL) and also inline Python Codes that you will find across the engine, such as in the Animation Callbacks and Python Code Components.

[Tab] Finder

The Finder tab could be a hidden gem for you if you've been using Cave and didn't know about it.



Many times through the game development you will find yourself wanting to find a specific asset or code snippet. This task may be hard if your project is big enough and have a complex structure. If you hit this spot, you can simply **press Ctrl + F to open the Finder Tab** (or enable it in the top left “Tabs” menu) and then search for the name you're looking for. It will do a complete search across the Project and show you any code results matching the query. Not only that, it will also search Asset and Entity names.

[Tab] Statistics for Nerds

This tab shows you some interesting statistics about your project, such as how long you've been creating it, how many times you hit Play Game, time spent playing VS editing, etc. As the name suggests, it's some fun statistics for nerds that the engine automatically and silently collects for you.

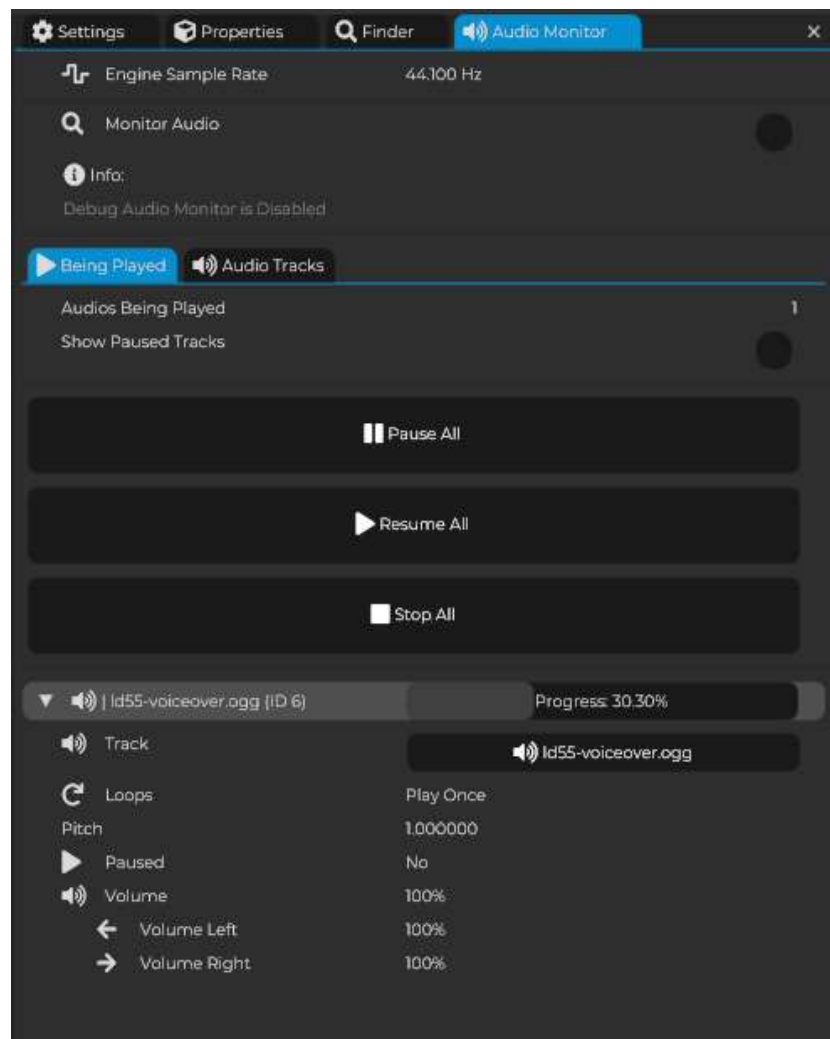
Note: *This information is not uploaded or shared anywhere and it's added to the project's **gitignore** by default, so even if you use git to version your project and work in a team, no one other than you should be able to see it.*

[Tab] Audio Monitor

When working with Audio, it may be hard to debug the game and understand exactly what's going on, so this is where this tab comes to play.

It will show you two important information:

A list (with details such as progress, loops, etc) of all sounds currently being played by the engine and a list will all audios you have in your project with individual details such as size, sample rate, etc.



This information effectively helps you to better understand what is happening with the audio of your game and also allows you to optimize and improve where it's necessary.

[Tab] Joystick Preview

If you're working with Joystick (Controller) support for your game, this Tab will be handy, since it shows all connected and recognized Joysticks and all its important information and Active Buttons. It allows you to see what's supported for the given Joystick, its key names and even test most of its features.

[Tab] Profiler

If you double click the **FPS (*frames per second*) Counter** in the top right corner of the Windows, it will open the **Profiler Tab** for you.

The Profiler tab is a complete toolset for you **to debug the performance of your game**. Not only that, but also allows you to profile the **memory usage**, the **save and load time** for you to be able to optimize the loading speed of your game, provides an option to **emulate a low frame rate** to test your game into harsh conditions, and also an option to **apply common optimizations** in batch for your entire project at once.

It is split into many tabs and sub-tabs. In the main **Profiler** tab you will see the average time spent into **physics, logic, and rendering**, and a tree and/or a bars view to better understand the bottlenecks of your game. So if you're running low on frame rate, this will be the place to look at.

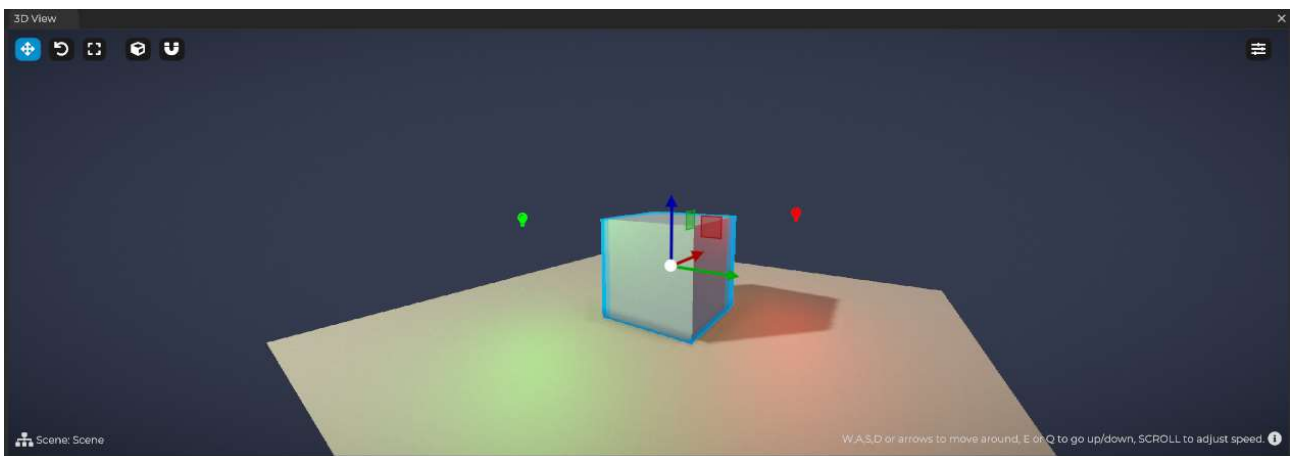
It has options to force GPU sync, making the profiling a little bit more accurate, and also to pause the frame capture to track individual frames for performance issues. Alternatively, you can save the **Profiler Capture** as a **log** file located in the **Logs/Profiler Captures/** folder inside your project directory, and this log file can be opened and explored into **Google Chrome's tracing system**. (open Chrome and type **chrome://tracing**, then hit enter).

1.4. Manipulating Entities

Now that you have an overview of the Editor Tabs, let's go back to the 3D View and the other related tabs (mainly the **Scene Graph** and the **Properties Tab**) and start playing around with the Entities. This is the best part, because it's the fundamentals that will allow you to create anything you ever wanted to! All the projects you want to make, they will all be constructed with this. So let's get it started!

In Cave Engine, the 3D world where everything takes place is called a **"Scene"**. You can have multiple scenes in your project (*again, we'll see how it works later*) and on every scene, you have a collection of **"Entities"**. Think of an Entity as your objects, but not only that, as the lights, a trigger, or even a folder to organize other projects: since an Entity can have **"Child Entities"**. The Entities are also composed by what we call **"Components"** and we'll see them in details very soon.

When you create a new Project, you will see that a Scene is created and opened for you by default: so the world you're seeing in the 3D View, that's your Default Scene. You already know how to move the camera around in it in order to see what's going on. Now, if you **Left Click the Mouse while hovering an Object**, it will select it for you. You can also left click it in the Scene Graph, assuming you know its name. Notice that, once selected, it the object will be highlighter in Blue and also display a Gizmo:



If the Entity have Children, they will also be highlighted in Blue. The gizmo is the main way of moving the objects around in the 3D world. If you're familiar with

Blender 3D or another 3D program, you probably already used those gizmos before and know how to use them. If not, here is how they work: You can **mouse over and left click any of those 3 arrows that appeared in order to move the object on the given axis**. If you also **hold Left Ctrl** while moving it, it will toggle snapping for the action.

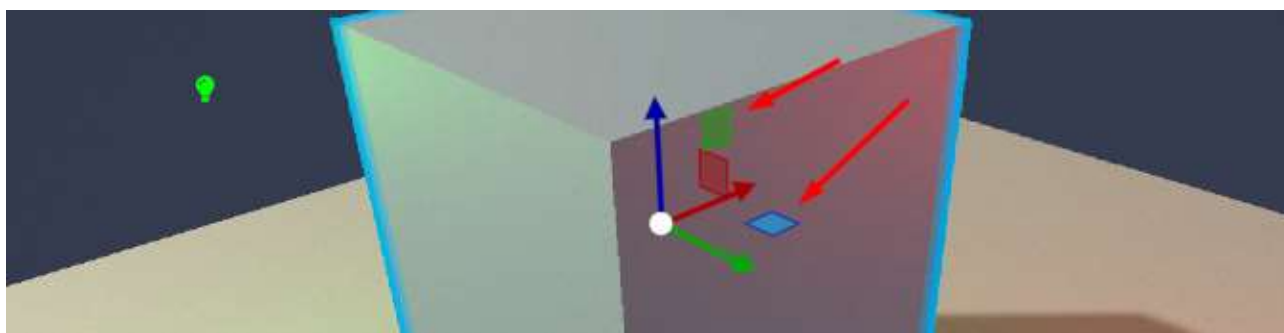
The 3D axis are:

- **Axis X (Red):** Usually the “side axis”
- **Axis Y (Blue):** The “up axis”
- **Axis Z (Green):** Usually the “forward axis”

The color convention in cave is a bit different, because most other programs use Red, Green and Blue to symbolize X, Y, Z axis, respectively, but Cave does swap the Blue and Green colors. The reason is that it hopefully makes it easier to remember that the Y axis is pointing at the Sky, which is typically Blue, helping you to identify that it should point up, generally speaking.

Of course, that's not a rule: if you're creating a space game, there is probably no “up” axis. But generally speaking, it's recommended to stick with this pattern whenever it's possible, since most engine's built in systems had that in mind when developed (such as the physics).

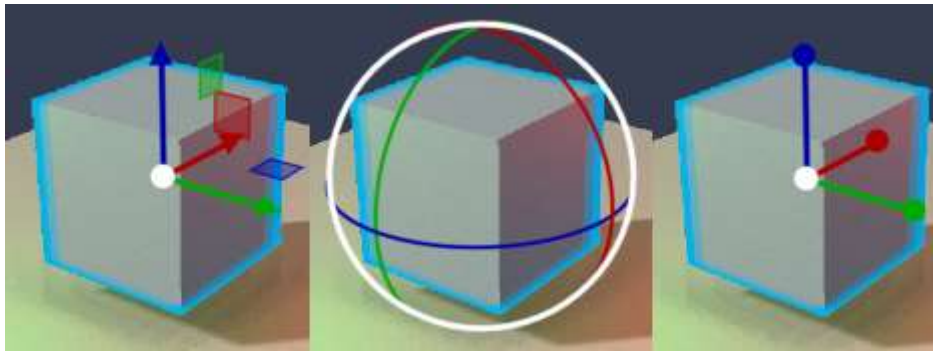
Continuing our exploration, you can also notice that other than just the arrows, it will also have some colored Planes near them:




Those planes, if you click and drag while mouse hovering them, will allow you to move in all axis, except the colored one. Last but not least, **dragging the White**

Circle in the middle allows you to “free move” the object according to the camera’s position.

The move Gizmo (the one you just saw) is not the only one: we also have the Rotate and the Scale Gizmos. They work as the exact same way as this first one, except that they do have different formats and obviously manipulates different Transform elements of the Entity. This is how each of them looks like:



You can toggle between those 3 gizmos by using the shortcuts: **(W) Move, (E) Rotate, (R) Scale**, or by using the 3 buttons in the top left corner of the 3D View Tab ().

If you are not happy with a Position, Rotation or Scale of the selected Entity, you can press **Alt + W**, **Alt + E** or **Alt + R** to reset them, respectively. And if for some reason you lost the entity in the scene or it is just too far away for you to manually move the camera over there, **Press F** to set the camera to a position where it is visible and Focused.

One last important shortcut to move Entities around is the **“G Key”**: With an Entity selected, mouse over a position in the 3D View world and **Press G**. It will immediately move the entity to that given position you were hovering. That covers the basics of Transforming an Entity in the world.

***TIP:** You will notice, once you learn parenting, that if an Entity has Children, moving it will also move the children accordingly.*

TO DUPLICATE a given entity, you can simply select it and press **Shift + D** or right click it in the Scene Graph Tab and select “Duplicate”. You can also **Hold ALT while**

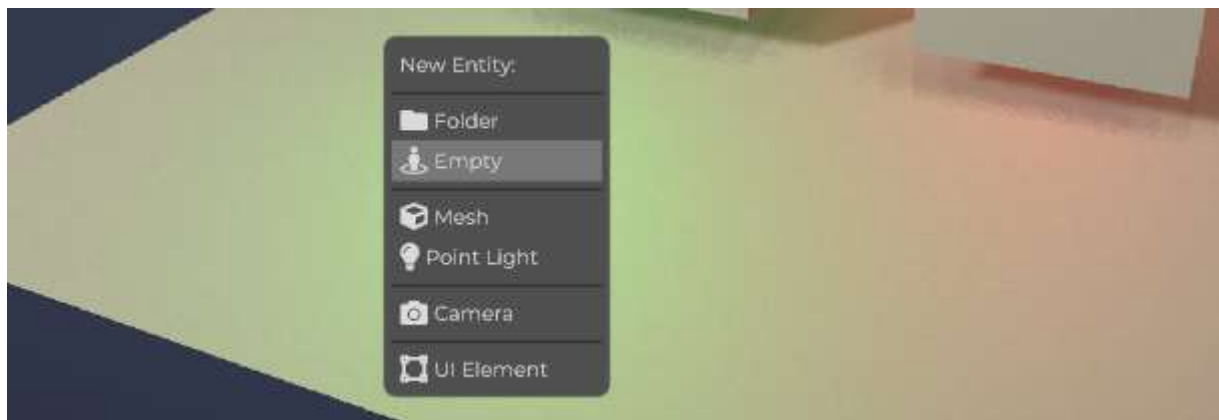
interacting with a gizmo (like the move arrow, for example) to leave a Copy of the entity at that given Transform (position, rotation and scale) and keep going. That is particularly helpful when creating levels (Level Design).

TO DELETE a given entity, simply select it and **press DELETE** on your keyboard or right click it in the Scene Graph Tab and select “Delete”.

***TIP:** Moving, Duplicating or Deleting an Entity also does the same with its Children.*

TO ADD A NEW ENTITY, you can press **Shift + A** while hovering the 3D View or simply right click an empty space in the Scene Graph. The difference between those two approaches is that if you do it by the Shift + A shortcut in the 3D View, if you were hovering a position with your mouse, the new entity will be added in this position instead of the center of the world (that is the case when you add it from the Scene Graph).

Notice that in both cases, a popup menu will appear with some default Entity Constructions for you to add and speed up your development process:



Those types have no internal differences other than the components they add in the Entity by default. Meaning that you can create a Folder (*that is the only one in this list that is an Entity with no Components in it*) and manually compose your entity as you want to (*but this approach is slower, of course*). Those default notations means:

→ **Folder:** An Entity with no Components. It will not be visible in the 3D World and will not have Gizmos. Useful for organizing the project.

- **Empty:** An Entity with a Transform. It will be visible in the 3D World as an Icon. Useful to mark places in the world or to start constructing a custom entity structure as you wish.
- **Mesh:** An Entity with a Transform and a Mesh Component with the Default Material and a Cube attached to it. It will be visible in the 3D World as a Cube.
- **Point Light:** An Entity with a Transform and a Light Component. It will emit light to illuminate the scene.
- **Camera:** An Entity with a Transform and a Camera Component. As the name implies, it specifies from where your world will be rendered at during the game.
- **UI Element:** An Entity with an UI Element Component. Since it does not have a Transform, it won't appear in the 3D world, but it will appear on top of it, as part of Cave Engine's Game UI System. We will talk about this one later.

By selecting one of those options, it will create what you want in your scene and select it for you. **Notice some interesting things regarding that:**

- An Entity in Cave **can** have **NO Transform**.
- Only **Entities with a Transform will appear in the 3D World**, since it serves exactly to give them a position in it.
- Later on you will see that, since Cave composes its entities by components, you can have anything in between the previous default options if you use your imagination.
- Cave Engine **does have a complete UI System** for you to create the Interfaces for your games. We will explore it later!

If you right click an entity in the Scene Graph and go to the **"Add Children"** menu, you will see this exact same menu as well. In this case, it will add the entity directly as a Child of the Given Entity.

Still talking about Entity manipulation, there is one very useful thing: You can press **Ctrl + C** and **Ctrl + V** in the 3D View to copy and paste, respectively, an Entity! Same

actions can be found by Right Clicking an Entity in the Scene Graph (to copy it) and in the New Entity pop up (to paste it, but it will only appear IF you have an entity in the Clipboard). This is specially handy if you want to copy and paste an entity from a Scene to another.

TO PARENT AN ENTITY to another, select the Entity you want to be a Child, then, in the Scene Graph, drag and drop it on top of the other Entity that you want to be the Parent. **TO REMOVE a parent**, right click the Entity in the Scene Graph and select “Remove Parent”.



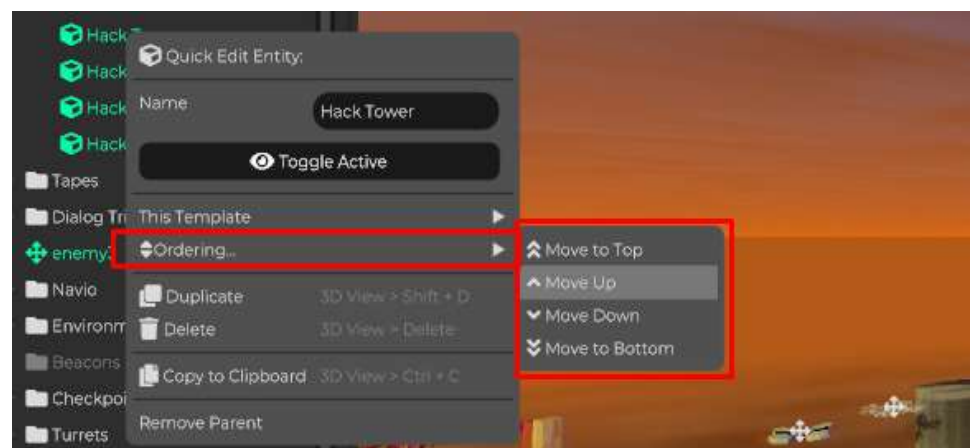
By doing that, you can create any type of parenting structure you want, as can be see in the image.

A common approach when creating game in cave is to use “Folder” entities to organize your scene, and then making all related entities a child of this folder. You can be very creative when it comes to that.

But this is not the only function of the parenting system: child objects moves and also gets added, duplicated and/or deleted along with their parents, meaning that it will impact a lot how you structure and make your games. It also affects the UI Element Component’s anchoring points as you will see later on.

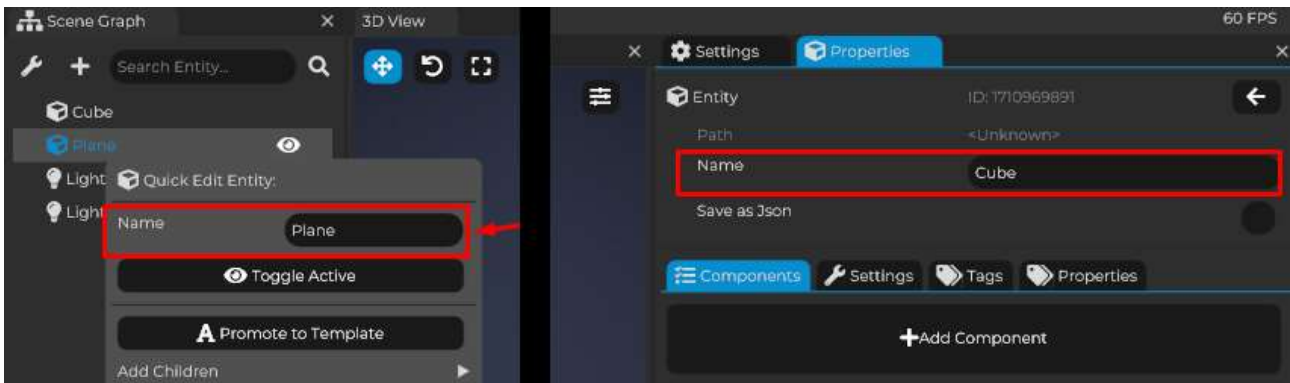
One last thing that it's important to know about the Entity hierarchy is that the order of which each entity appear may be important to you, either by organization but also to define what gets executed first and so on. You can mouse over an entity in the Scene Graph, right click it and do to this options bellow to reorder them as you wish.

Please note that or ordering is bound to the entity scope, meaning that it will always be relative to its Parent Entity ordering.



1.5. Entity Names and Activity

At this point, you probably noticed that Entities can be named. It's very simple to rename entities: simply right click on it in the Scene Graph and you will be able to do it. Alternatively, you can also select the Entity so it appears in the Properties Tab and rename it over there. Check the two options below:



Different from other engines, **Cave Engines DOES NOT use the Entity names as Unique Identifiers**, meaning that you can rename any entity at any time without affecting Cave's internal behaviors and you can also have multiple entities with the same name. This is also true for the Engine Assets in the Asset Browser, so you can expect that later on when playing around with them. And what does it use to uniquely identify Entities, Assets and other data formats? Each of them have a **Unique ID number**, that is displayed on the top right corner of the Properties tab by default.

Those IDs are always different from entity to entity, asset to asset and can be used to query things via script in game. It's also possible to use the entity names, just know that it may have multiple ones with the same name. You will not going to find a lot of practical case, as the engine user, where the ID will be used widely, but it's still important that you understand that the engine itself works that way behind the scenes.

Entity Activity:

If you look closely, when you select an entity it will have **an Eye Icon near to its name in the Scene Graph**. If you click on it, the eye will be slashed and the entity

will disappear from the world. The same result will happen if you right click it and select “Toggle Active” or if you go to the **Properties Tab**, then the sub tab “Settings” and toggle the “Active” checkbox in it. By clicking again in the Eye Slashed, it will appear again.

What you’ve done here is disabled and then enabled back the Entity.

A Disabled Entity still exists in the Scene, can be accessed, edited and queried by scripts, but does not interfere with anything in the active game world. Meaning that it does not have physics, does not run any logic and is also completely ignored by the rendering process (so it will not appear on screen). You can also freely deactivate or activate an entity in game using code, just notice that, since a disabled entity does not run code, it would not be able to reactivate itself (meaning that another entity needs to do it).

This feature is specially useful if you’re making something in your game that needs to appear at specific moments, let’s say an enemy encounter for the final boss battle that will take place at the beginning of the map: so when the player first approaches the location, there will be nothing in there... but when it’s time to have the encounter, it all appears. You can do that by placing all the enemies and even scene props and other useful stuff inside a folder entity (for example) and leaving it disabled.

The difference between Disabling vs Deleting an Entity DURING the game (*we call this “killing” the entity*) if that if you delete and re add the object, it will have its original state built back. If you simply disable it, it will remain in memory and any changes made before disabling it will persist. So if you have an Enemy in game and it loses a sword during combat, if you disable it and then re enable it, it will probably still have the sword missing (*depending on how you programmed that, of course*).

Advanced Tip: Disabling an Entity does triggers the **Component’s end(...)** method. The same with Enabling it: it will trigger the **Component’s start(...)** method.

When it comes to Adding vs Enabling an Entity DURING the game, it comes more to a practicality than anything else: Let’s use that previous enemy encounter as an

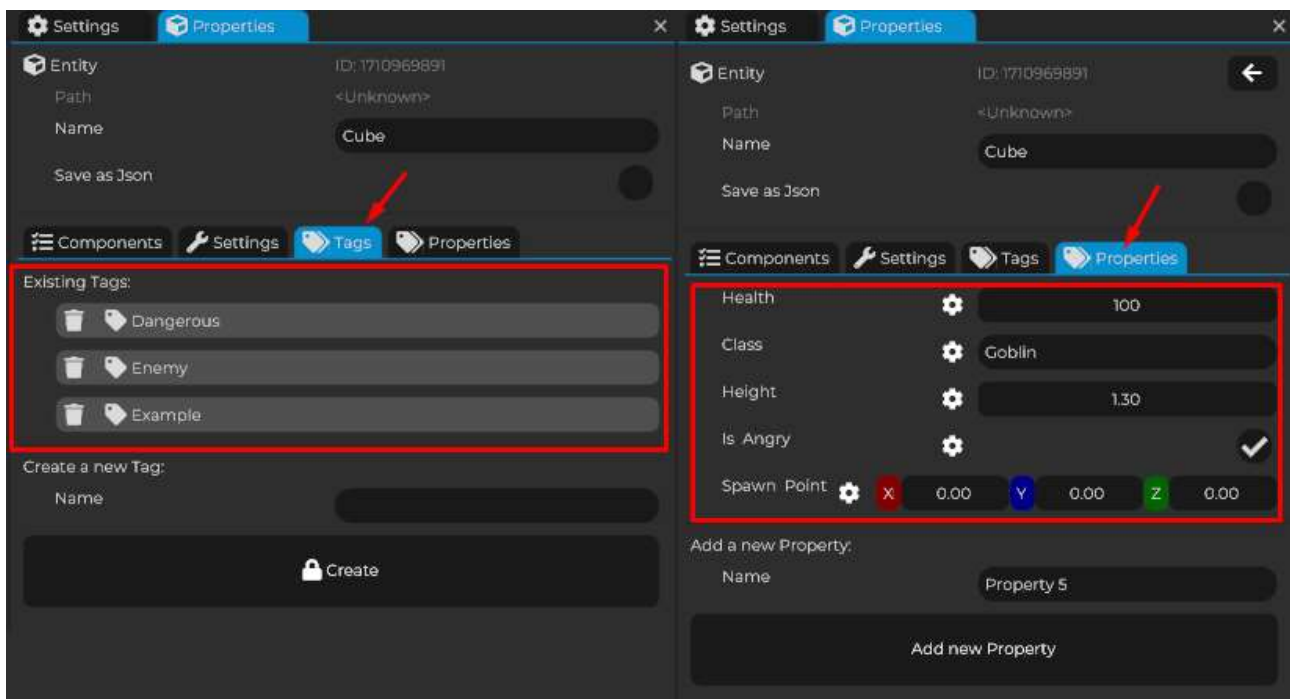
example. You will discover soon that adding Entities in game can be done by hand, manually building the entire thing via code, but mainly using **Entity Templates**, that we will see in detail soon. As a simple explanation just for the sake of this example, the **Entity Templates** are *“pre built entities”* that you want to use and spawn a lot across the game, perfect for Enemies, Items, Props and even the player itself. It’s amazing and well recommended that you use this feature a lot once you learn it. But for this specific example (the enemy encounter), it will not make a lot of sense to use it, since you will only ever have this specific encounter once in the entire game and it is very dependent of the rest of the map it is located on. You can use a template for that, but it will probably just make your life harder. :)

Last but not least, it’s worth mentioning that the Entity Activity can also be used to temporarily disable parts of your map while editing it, to make it easier for you to visualize what is going on.

1.6. Entity Properties and Tags

You may want to store some custom metadata into the Entities, either to tag them as an Enemy, Player or something or to store a custom value, such as their Health Points, etc. For that, Cave Engine provides you two interfaces: the Tags and the Properties. We will discuss both in this section.

To locate them, select an entity and, in the Properties Tab, open the Tags or the Properties sub tab:



In the two images above, you can see both sub tabs opened. I've added some custom Tags and Properties in it before taking the Screenshots to serve as an example of what are they meant to be (*you can see them inside the red outlines I've added*).

The Tags are simply strings and can have whatever name you want. But each tag does not contain a value assigned to it. Behind the code, the Tags were directly implemented by the engine in pure C++ and are usually very fast to query and check, compared to the other option (the properties). For that reason, you will later find that even the Scene have built in APIs to query entities with a given Tag, for example. So if you need a fast way to identify some Entities, the tag is the best option for you.

You can add a new Tag in the Tag sub tab by typing the tag name in the “New Tag” field and clicking on the “Create” button. You can delete an existing one by clicking on the “TRASH” icon next to the corresponding tag you want.

The Properties are literally a Python Dictionary that each Entity have. You can add any string keys to it and assign any value you want for them. Health points, the character’s class or race name, custom status such as a boolean to indicate if it’s angry or not, it’s up to you. If done via code, the value can have any type you want. Via interface, Cave allows you do have some basic formats, such as integers, floats, booleans, strings and vectors.

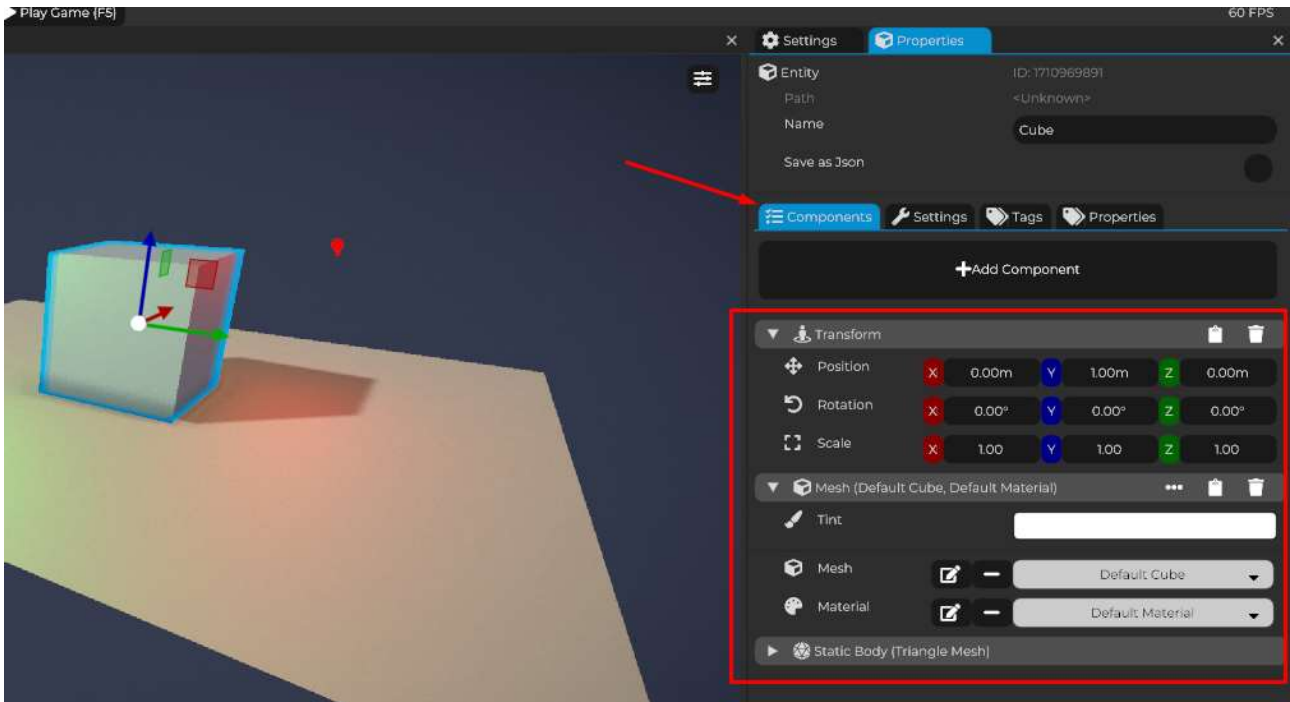
You can add a new Property by typing the prop name in the New Property’s “Name” field at the bottom of this sub tab and clicking on the “Add new Property” button. You can either delete an existing Property or change its data type by clicking on the Gear icon next to the corresponding prop you want.

Advanced Tip: All the Properties of an **Entity Template**’s Root Entity will be displayed as overwritable props to all instances of that given Template.

Note: Since the Properties are Python code, it tends to be slower than the builtin engine options, such as the Tags. While it’s fine to do most things with them anyways, using the properties to perform scene queries, such as grabbing all entities with a given prop in a scene with dozens of thousands Entities, is not encouraged. Unless it’s a very punctual thing.

1.7. Entity Components

Now we are approaching one of the most important parts of the engine. Because as you probably already may have noticed, everything in Cave's world is Entities composed by Components. Let's get the Default Cube we have on your Default Project's main scene for example. If you select it and then go to the "Components" sub tab (in the Properties Tab), you will see something like this:



On your instance of the engine, each component will probably be closed by default, but you can mouse hover and left click them to expand, showing its contents.

The default Cube will have a Transform, a Mesh and a Rigid Body component, as you can see respectively in the screenshot. Notice that they are not named exactly like that tho. That's because Cave do prioritize some custom component naming in order to facilitate for you to find what you want. So the Mesh component will display its mesh and material names in its name and the Rigid Body, if it's a static or a dynamic body as well as the collision type.

But if you mouse hover any of those components, you will see their original names. This will be specially handy for you layer to know how to call each component in order to retrieve them via Python.

When expanded, the component will have a Delete (Trash Icon) and Copy (Clipboard Icon) displayed on its top right corner with their corresponding functions. Some components may also have extra icons, like the Mesh Component that does have an icon with three dots in it. Those extra icons exposes some extra functionalities that are not the same for every component. The Mesh, as an example, provides an option for you to create a rigid body component from it (to add physics) if you click in it.

TO ADD A NEW COMPONENT, click the “Add Component” button on top of this Editor Region or simply right click on an empty space and the same new component popup will appear. This popup can be seen in the image on the right.

Notice that we have a lot of Component types. They tend to be self explanatory, but of course I'll go throughout all of them and provide you a bit more information during this Documentation.



Let's start with the most used ones and keep going from there:

[Cmp] Transform Component

One of the most important Components: it stores 3D information regarding an entity, more precisely its **Position, Rotation and Scale**. Many other components directly rely on this one, such as the Mesh, Rigid Body, Camera and pretty much any other component with some 3D functionality.

This component also handles parenting hierarchy transformation, to make sure that a Child Entity have its position, rotation and scale relative to its parent.

It's important to know that, while you're seeing the Rotation in the Editor as [*Euler Angles*](#), the engine internally uses [*Quaternions*](#) for that. You can click on those two names to be redirected to a Wikipedia page on each topic, if you're new to them.

Via code, you will find both options (Euler and Quaternion) available for you to use as you wish.

The reason why Cave uses both is because while Quaternions usually provides a more flexible and precise way to handle 3D rotations, they are harder for us to comprehend and manipulate by hand, so an Euler representation is often used instead. So you can easily rotate the entity using Euler Angles when possible and rely on Quaternions to do more advanced and flexible movements.

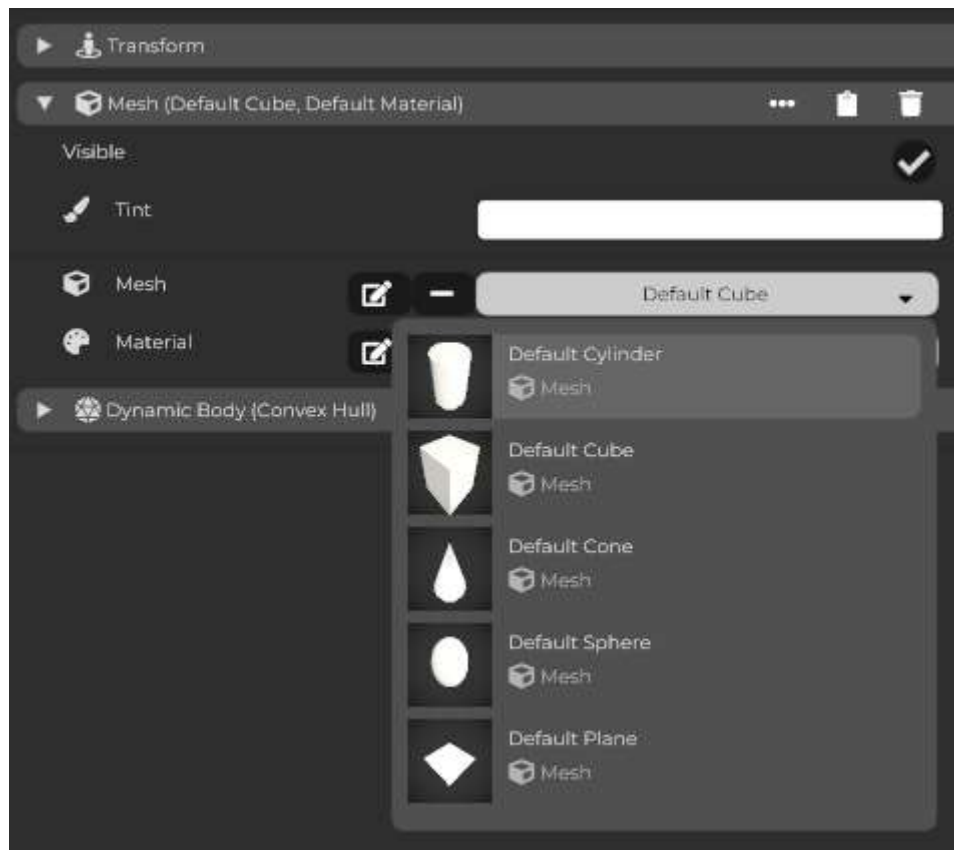
Advanced Tip: *Via code, you can also find a lot of useful methods in the Transform Component to manipulate vectors, positions and other things. It also provides a matrix representation of the entire Transform. It's worth checking the python stubs for the complete API, once you reach this part, of course. :)*

[Cmp] Mesh Component

The Mesh Component serves **to Render a given Mesh and Material at that Entity's location** (*guess what? It relies on the **Transform Component** to know that!*). It is one of the main routines in Cave responsible for drawing things on screen.

It only have three variables for you to define: its **Tint, Mesh and Material**. Since this is likely your first time with cave, let me explain what they are and how it is used. Starting with the most simple one, **the Mesh Component's Tint** is a local **RGBA** variable that allows you to specify a color that gets multiplied by the final material output. This is useful to set custom temporarily colors, like making the enemy glow red as a visual indicator of damage, when the player hits it.

The Mesh and the Material are Cave Assets, meaning that they are located somewhere in the Asset Browser. Before we talk about them, let's take some time to observe a common pattern in the Engine: You will notice that in Cave, **every place where you can assign an Asset will have a similar interface**, with the property title and a light colored drop down menu containing the Asset Name (or None, if none). If you click on it, it will expand, showing all assets from that given type in your Project:



Some Assets, such as **Meshes**, **Materials** and **Textures**, have a custom **Thumbnail** to facilitate finding them. If the list is big enough, it will also contain a search bar on top for you to search for a specific asset name. When you have an asset set to it, you will also find a minus icon next to it for you to remove the asset (and assign None) and a pen icon to edit the Asset. The Edit icon is a great shortcut to allow easy access to its properties. Just remember that when editing an asset like that, unless it is local, your changes will be applied to all other instances using it.

It's Good to Know: Cave Engine calls those Properties where you can assign Assets as **"Asset Handlers"**. They have a specific type (e.g. Texture, Mesh, Material, etc) and you can programmatically set an Asset to it using its Name, Unique ID or the asset instance directly. You will interact with them a lot, not only when Editing the game in the Editor, but also when writing your custom game code.

Now that you have a basic understanding of the Asset Handlers, let's understand **what are the Cave Meshes and Materials**, used in this component.

The Mesh Asset is a container for 3D mesh data. It stores vertex positions, normals, tangents, UVs and animation data (layer weights and IDs) and also the triangle

index list. The Mesh also store hints related to its default corresponding material, transform and other useful things. Last but not least, Mesh Assets also have optional Level of Detail (LOD) and distance culling information, used for the Engine to optimize the Rendering. You will see more about all that later, but that's the basics for now.

Advanced Tip: *Cave Engine's Meshes uses indexed lists to represent its faces and it always uses the [Triangle Strip format](#) (click to open Wikipedia).*

The Material Asset is a container that stores a [Shader](#) and a collection of [Shader Uniforms](#) to specify how a particular mesh will be rendered. This will directly influence how the engine calculates the Meshes final position for each vertex and also how the light is calculates per pixel rendered to the screen.

Cave provides a default Shader for you to use, where you can set Albedo colors, textures and so on, but it's always good to know that if you want to, you can write your own custom ones using GLSL code. But that's a bit more advanced, so I'll not go into details about it yet, since this is just the introduction.

Notice that while the Mesh Asset does have a Hint regarding its preferred Material, it does NOT directly attaches to a given Material, meaning that in Cave, the connection between a Mesh and a Material is very week and often **solely defined by the Mesh Component**. This approach is very different from most other engines and have its own set of pros and cons.

A positive thing about this approach is that it keeps things separated and with a unique purpose, allowing reuse of the resources in many ways, such as using Meshes as Physics Colliders, Particle Spawner shape or whatever your imagination can bring up with. It's also easy to replace the materials used to render a given mesh.

But an important compromise to keep in mind is that, due to its nature of the engine, **meshes does not support multiple materials**. So if you import a mesh from Blender to cave and it have more than one material assigned to it, you will notice that the engine will split that mesh into multiple ones (one per material slot).

With that in mind, **if you have a multi material mesh and wants to use it in cave**, you will need to create multiple Mesh Components, one per material, and assign the corresponding mesh to it. In practice, this is not a huge deal, it's just a practicality regarding the way the engine works.

***Important:** if you edit a Mesh Component through code and change its mesh or material, you may notice that nothing will change. That's because the Mesh Component will only submit itself to the internal engine rendering systems when it starts, so newer changes won't be acknowledged. In order to acknowledge them, you need to call the **Mesh Component's reload()** method after the changes.*

[Cmp] Animation Component

Given an Entity with a Transform and Mesh Component in it, if the Mesh assigned to the Mesh Component have animation data, you can add this component in it too in order to specify and execute Skeletal Animations. It takes an Armature and an Animation Asset and will run it, assigning all the appropriated deformations to the Mesh in real time.

While the interface of this component in the Editor may be simple, the component itself is not: it provides A LOT of functionalities for you to animate your characters the way you want to via Code. We commonly call this component the **Entity's 'Animator'**.

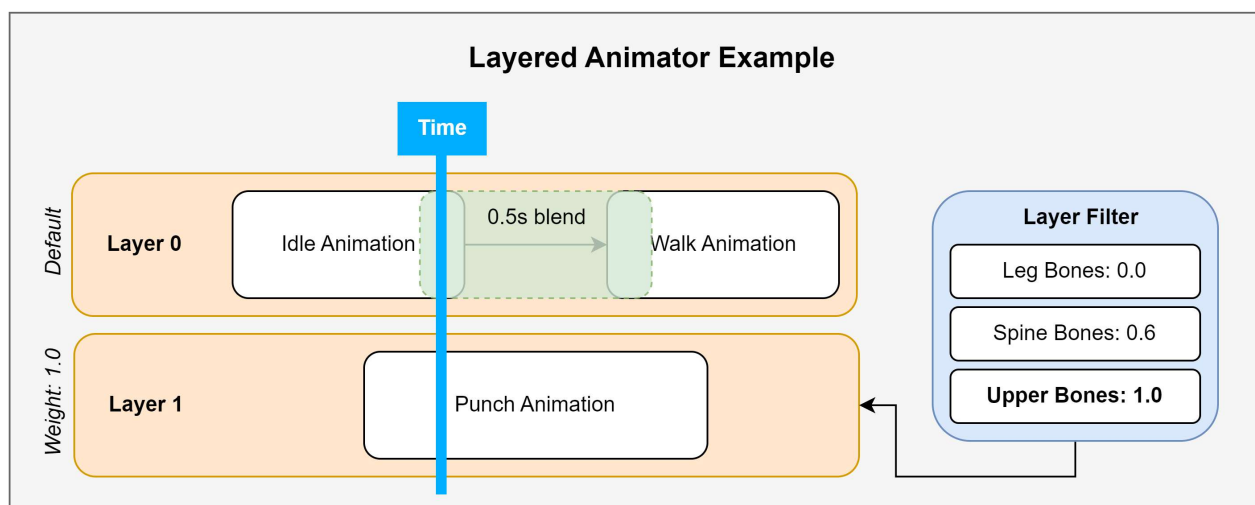
Programmatically you can get it and play any animation you want, given an Animation Asset, providing all sorts of extra parameters and controls, such as the animation speed, its blend, if it should be player in loop or not and others. To understand a bit better how to use it, it's important to first comprehend how it handles animations.

The Animator have a Layered Stack system, where you can use any layer amount you want to and each layer executes an independent animation. The layer is referenced by its ID with **zero meaning the first one, then one, two and so on...** If you play an **"Animation A"** on **layer 0** and then changes it to play the **"Animation B"**

instead, you can provide a **Blend Time** (*in seconds*) for it to interpolate between those two, properly blending it and delivering a smooth outcome.

When using multiple animation Layers, each layer pose is evaluated and each of them (except the first one) is blended with the previous layer pose given a **Layer Weight** variable. So if layer 0 is playing a “walk forward” animation and layer 1 a “walk right”, if the layer 1 weight is set to 0.5 (*think about it as 50%*), it will first evaluate the “walk forward” pose, then the “walk right” and blend those two by 50%, so it will be half the forward walking pose, half the right walking pose (possibly meaning a diagonal pose). This process is additive, meaning that this final interpolated pose will be used as the first animation to interpolate with later 2, if in use, and so on.

In addition to the Layer Weight, you can also add **Layer Filters**, to specify **specific weight on a per Armature Bone basis**. For example, let’s say that you are creating a third person game and you want the character’s legs to play one set of animations, such as idling, walking, running or jumping, and the upper body to play another set, such as punching, aiming or shooting. You can use the first layer (*layer 0*) to play the leg animations, then set a **Layer Filter** for the second one (*layer 1*), specifying a 0.0 (meaning 0%) weight for the leg bones and slowly increase the weight across the spine bones, until it reaches 1.0 (meaning 100%) for the upper body parts (arms, head, shoulders). Then you execute the punching animation on layer 1 and it will work exactly as you may expect it to. Check this scheme:



You can see in this example scheme a perfect use of Cave's Animation Component functionalities. As described before, the character is executing its lower body animations on layer 0 and the upper body ones on layer 1. Notice that it was idling, then the player decided to punch, causing the Punch Animation to start playing on layer 1. Then, during the punch, the player decided to start walking, triggering the Walk Animation on layer 0, with a half a second blend. So at the point marked by the "Time" pointer, three animations are affecting the pose: on layer 0, most of it is the Idle Animation, but already started blending to the Walk Animation. On layer 1, the Punch Animation is in progress, blending with the previous one by 1.0 (the layer weight) times each individual Bone Weight assigned by the Layer Filter.

Keep in mind that the Layer Filter is optional and if not provided, will assume a 1.0 weight to all bones (times the layer weight). In the scheme, I've also simplified the look of the Layer Filter to fit on screen. Ideally, you will be setting each individual bone weight or using Cave's APIs to recursively assign a weight to its child bones.

Tip: *The complete Python API for this Component (and all others) can be seen in the Python Stubs files that comes with the engine.*

One last important thing to mention about the Animator is that it also allows you to **provide a callback function (Python) to be executed right after its armature evaluation process**. With this callback you can get the final pose and do any manual modifications you want, such as move or rotate a bone or even implement an Inverse Kinematics solution for foot placement and so on.

As you can see, the Animation Component provides an extensive function set for you to animate your game characters in any way you want to.

[Cmp] Animation Socket Component

Still talking about animation, let's say that you have an animated character and now wants to place a Sword in its right hand. You can create a new Entity for the sword, with a Transform and a Mesh Component, then make this entity be a child

of the animated character Entity and then add an Animation Socket Component to it (*to the sword entity*).

With this component, once you have a setup like we just exposed, you will be able to select a specific bone for the mesh to be “parented” on and also provide offset values to correctly position it where you want.

[Cmp] Camera Component

The Camera Component, as the name suggests, specifies a camera using the Entity’s Transform. It have all the main options you can expect from a Camera, such as its Field of View (FOV), clip start and end, which means the minimum and maximum distance that the camera will render, and other useful properties.

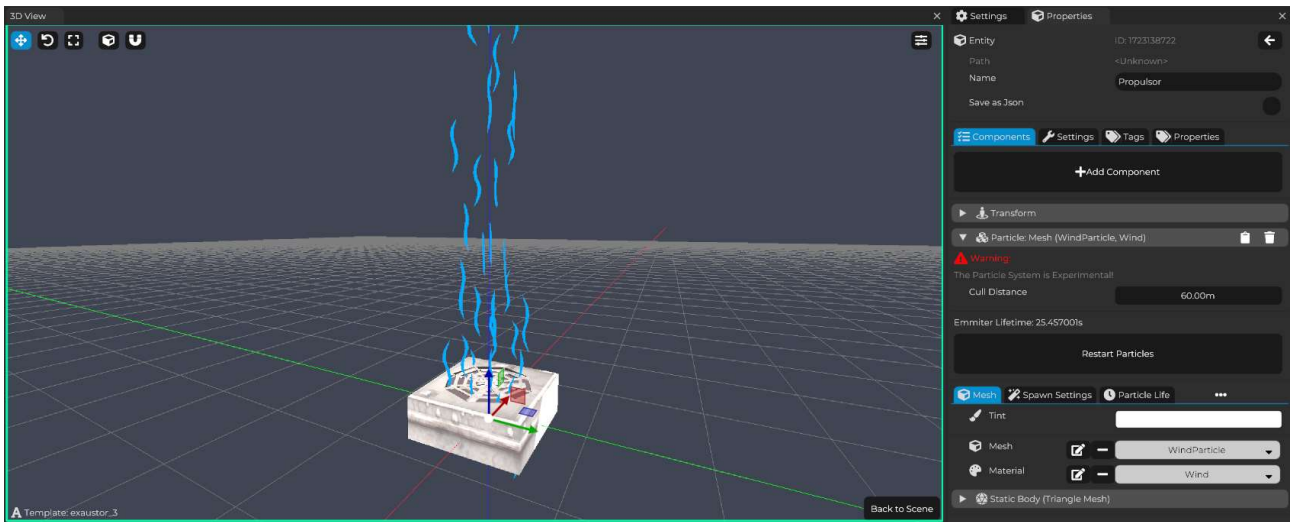
You can specify a custom Camera Lerp factor in it too, allowing the camera to slowly move towards its final transformation instead of clipping to it. It’s also possible to expand a Preview in the Component’s UI, to see exactly what the camera is seeing.

[Cmp] Light Component

The Light Component creates a point or spot light in the world, that gets passed to the rendering Shaders in order to mimic lighting in the scene. It provides options for you to adjust the Light radius, color and intensity.

[Cmp] Particle Component

If you want to create particles, visual effects or simply have a lot of visual only (with no individual physics or logic) instances of an element like a mesh, the Particle Component will be a good fit for you. At the time I’m writing this Documentation, it is still experimental, so some things may change in the future, but it is already good to be used in a lot of cases. We use it a lot in our games here at Uniday Studio:



In this image you can see a usage example of it: we used the Particle System to simulate “wind particles” flying out of an air vent. It was necessary for it to be very visible in our game, because the played had a lot of interactions with it.

[Cmp] Character Component

The character component allows you to create a custom character type physics for your game. It differs from the other physics components such as the reached body because this one have a specific physics fine-tuned specific for character type objects such as a person.

It allows you to control the radius, height and offset of the shape, which is always a **capsule like shape** to mimic the regular silhouette of a standing or crouching human. It has settings such as fall speed, gravity, gravity direction, jump speed and max slope, which is the maximum inclination that the character can climb It also provides you the regular collision masks that the engine uses to filter collisions.

You will see more about collision masks when we talk about the game engine physics system.

[Cmp] Rigid Body Component

The Rigid Body Component is **the main physics component** that you will need and use throughout your entire game. **As the name suggests, it serves to identify and give it a physical shape to any rigid object.** And by rigid, it means any object that can be collided with, casting and/or receiving collisions and not only that, but also objects that can react and respond to collisions.

An unmovable object is considered by the engine as a **static object (static rigid body)**. And this can only change if you enable the **dynamic** checkbox in the component settings. Once the rigid body is marked as dynamic, it will have a mass and also react to the collisions with other bodies and resolve those collisions. Meaning that if the object is about to clip into a wall, for example, it will resolve this collision to not clip it, and it will also fall with gravity.

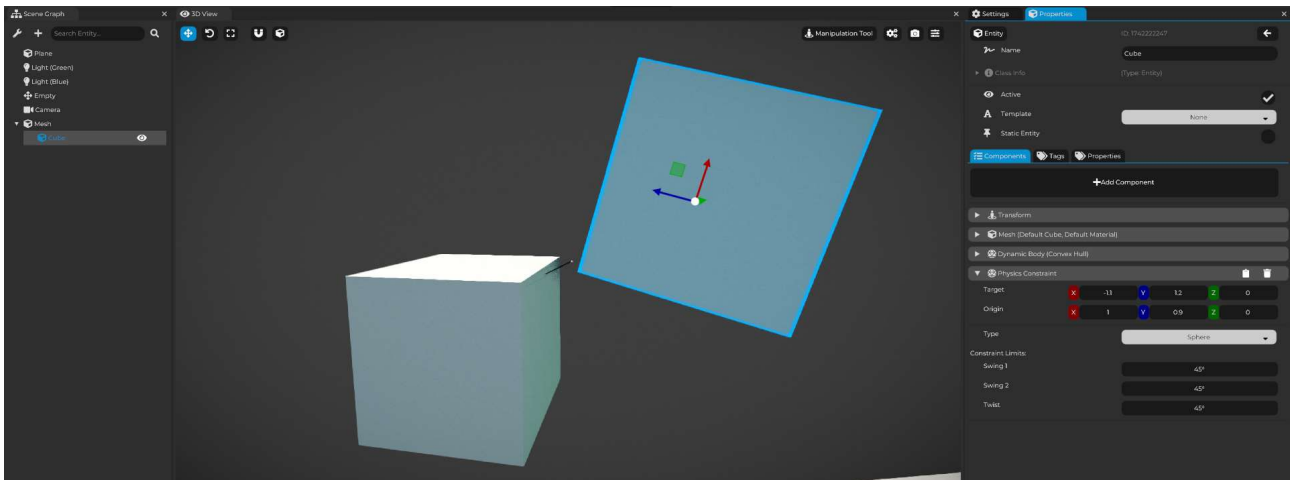
A rigid body can also be marked as ghost, meaning that it will still detect collisions and those will be available for you via code, but it will not react to the collisions. So just like a ghost, it will fall and clip through walls, roof, the ground and any rigid body in the scene. This is very useful, for example, to create triggers in the scene to detect when the player or something crosses certain point in the map.

Angular factor serves to determine if a given rigid body will rotate when resolving the collisions or not. You can adjust this on a pair axis basis and one means that it will rotate and zero means that it will not rotate on that axis.

You can also **determine the shape of the rigid body by passing a mesh asset to it.** If no mesh is provided, it will use a box shape by default and you can adjust the offset and scale of this box in the component settings. **If you provide a mesh,** it will use the mesh to calculate the collisions and provide you three options for this: (1) Use the bounding box of this mesh, (2) the convex hull of this mesh, which is a version of the same mesh without any concavities in it or (3) the triangle shape of the mesh itself. It's worth mentioning that due to how physics works in video games, **a dynamic rigid body cannot have a triangle mesh collision shape.** So even if you provide a mesh for it to collide, it will either be a box collider or a convex hull which is the default.

Last but not least, you can define **collision masks** for the rigid body, we will talk about the masks later. Also a distance culling for it, which is an optimization option to temporarily disable the physics for rigid bodies that are far away from the player's camera.

[Cmp] Physics Constraint Component



A Physics Constraint Component allows you to **constrain a given rigid body into another rigid body**, which is the one that the parent Entity of the current Entity have.

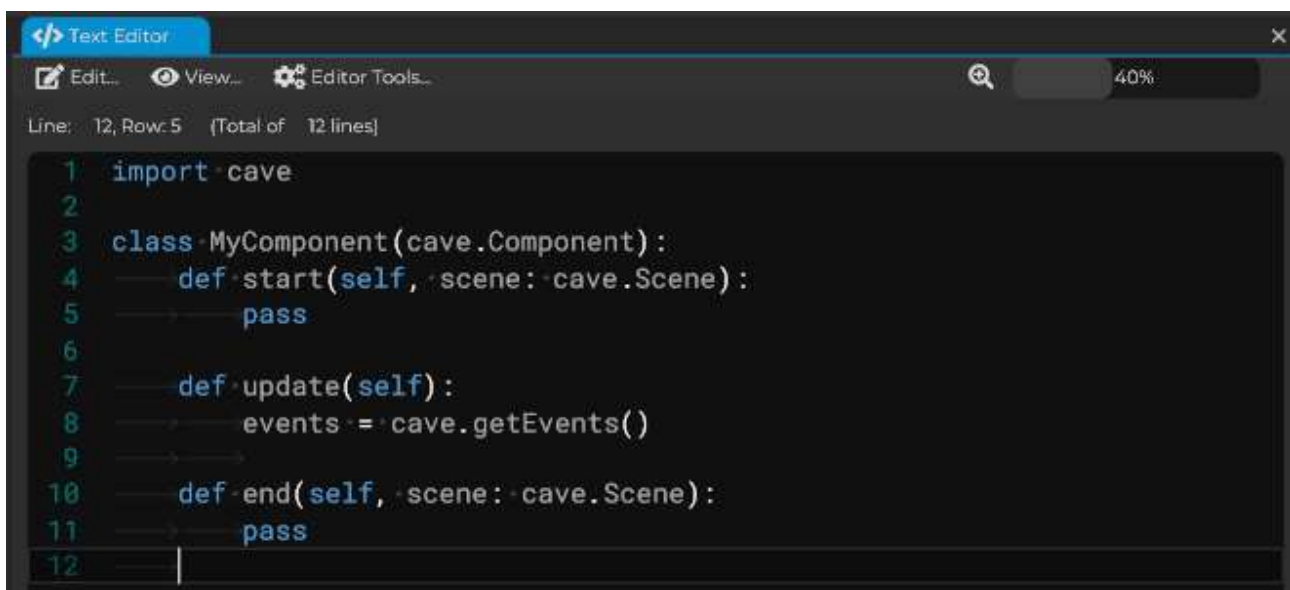
You can set the **origin** of this constraint, that it is relative to the parent's transform, and also the **target** of it, which is relative to the entity transform itself, the type of the constraint and limiting factors such as the swings and twist. This component type is very useful to create more advanced physics systems such as dynamic bridges and even character ragdolls.

[Cmp] Python Component

The Python component is the main component that allows you to write custom logic to your game. You can do that by creating a Python script in the asset browser and then attaching it to this Component or simply creating the Python Component for the Entity you want and then using the “New Script” button in the Component's UI to automatically create this script for you and set things up.

Python components, since they are created through script, which is an asset in your game, can be reused across your project to give logic to different entities in different situations. So it's a very modular approach to create your games. It is also a good option if you want to edit the scripts using an external tool such as VS Code, which is supported by the engine, since Python scripts are exposed in the file system and can be opened by third-party programs.

This is how a newly created Python component script looks like in the text editor tab:

A screenshot of a text editor window titled 'Text Editor'. The window has a menu bar with 'Edit...', 'View...', and 'Editor Tools...'. Below the menu bar, it shows 'Line: 12, Row: 5 (Total of 12 lines)'. The editor area contains the following Python code:

```
1 import cave
2
3 class MyComponent(cave.Component):
4     def start(self, scene: cave.Scene):
5         pass
6
7     def update(self):
8         events = cave.getEvents()
9
10    def end(self, scene: cave.Scene):
11        pass
12
```

Notice that **MyComponent** inherits from **cave.Component** and that it does have three default methods: **start**, **update** and **end**, but it can also optionally have other methods that you can add, such as the **pausedUpdate**, **firstUpdate**, **editorUpdate** and **lateUpdate**.

While writing the code inside the component, you can access the owning entity of this component by simply using the **self.entity** variable.

Later in this book you have a better understanding of what each of those methods are and when they're called by the engine. But just to get you started, **the start method** is called every time the component is initialized or activated and **the end method** is called every time the component is deinitialized or deactivated. Every component in the engine, not only the Python component, do have a reload method that reloads the component by simply calling its end and then the start method again.

The update method is called every frame if the scene is not paused. If the scene is paused, **it will call the pausedUpdated instead**. The **editorUpdate** is called every frame while the game is in editing mode. This method is meant for the bug only and it will never be called in the exported game or even when the game is running.

IMPORTANT: *If you pay close attention, you will realize that the Python Component, that you actually add to the entity using the game engine's editor, only serves to link against the actual Python component that you create in the script as a code. **So there is an indirection going on here**, meaning that the Python component you create through code is not directly added to the entity, but instead it is wrapped around the actual Python component that you see in the engine's interface. **In practice, this will not represent a lot of differences to you**, but it's very good to know. **There is only one small difference** that you will notice and this will happen when you try to access a custom Python created component from an entity. If you use the regular **entity.get(...)** method, that is meant to allow you to get components from the given entity, it will not be able to find the Python created one. For that, you need to create to use the special **entity.getPy(...)** method or set the flag for the regular method to also search for Python components.*

For more information regarding the Python API, please read the API documentation.

[Cmp] Python Code Component

Just like the Python component, the **Python Code Component** allows you to create and execute Python scripts for your Entities. **Except that this one is meant for simpler code and non reusable code.**

So if you want, for example, to create a coin that rotates around its axis for the player to collect, this logic to rotate around the axis is simple enough to not justify

create an entire new script with a component class in it. So you can use the Python code component to straightaway add this code directly to the update method.

This approach is much more straightforward, which can be very good, but as a trade-off, it also loses flexibility. Since now, you can no longer share the same code with multiple entities without having to duplicate it.

It's important to mention that any variables that you create and define in the **start** or any other method of the Python code component **are kept across the different method scopes**, meaning that you can initialize a variable in this start method and use it in the update method and expect things to work out.

Since the Python code is meant for smaller, in place, logics, the Cave Engine also understands that you may want to have those smaller logic being spread across the map, copied thousands of times. For example, **in the coin example that I mentioned** that the player can collect, **you may want to have thousands of rotating coins spread across the map**. Because of that, the engine provides an extra **optimization** step for the Python code component in particular.

Once the optimization is enabled, it allows you to have automatic **distance culling for the code**, meaning that after certain distance from the camera, that component will completely stop calling the methods that was supposed to run every frame and this will only be restored on the distance criteria is met again. Or you can select the Frames Skipping optimization, that allows you to run logic but instead of every frame, you run it every so often, effectively skipping a lot of frames and saving performance. For example, if you have a button that checks if there is something such as a player on top of it, you may not want to run this every frame, 60 times a second. Maybe running it once or twice a second will be enough.

[Cmp] Custom Camera Controller Components

We have some custom pre made Camera Controller Components:

→ **First Person** Camera Component

→ **Third Person** Camera Component

→ **Top Down** Camera Component

→ **Mouselook** Component

They're meant to allow you to create the types of camera controls a bit more easily. But of course you can still make them yourself from scratch using Python if you want to.

[Cmp] **Vehicle** and **Wheel** Components

The vehicle and wheel components are meant for you to create vehicle physics for your game. In order to create such physics, the very first thing you need to do is to create an entity for the vehicle chassis, that contains the body of it, and add both a **rigid body component**, with the dynamic checkbox enabled, and the **vehicle component** to make this rigid body act as a vehicle. Then you add a child entity per wheel that you want it to have, position them in the correct place and create a wheel component. The wheel component doesn't need a rigid body to work with. The only rule is that it needs to be a direct child of the entity that contains the vehicle component.

With that set up, you can control each individual wheel radius, settings and suspensions in the wheel component, and also the vehicle engine forces and steering in the vehicle component. Front wheel and back wheel are differentiated by the checkbox "**is front wheel**" that the wheel component have and only front wheels can have steering, but there's no limit of how many front or back wheels a vehicle can have.

***Tip:** You don't need to create the actual visible meshes for the vehicle and also for the wheels in the same Entities that you used to add the vehicle and wheel component. If you want, you can create a child Entity for the vehicle and also for each wheel to be able to proper adjust the mesh used to handle them. This is a good tip because sometimes the wheel mesh that you created in Blender, for example, are facing the wrong direction, so you want to rotate it. You will find that the wheel component will not allow you to do this in the same Entity, so you can create a child Entity instead.*

For more information on how the vehicle system works in cave, please check the demo project for the vehicle scene and showcase.

[Cmp] Custom Controller Components

Cave Provides some custom controller components for early testing:

- ➔ Player Controller Component
- ➔ Vehicle Controller Component

As I said, this is meant for early testing your game or for very simple use cases. That's because both the player controller and the vehicle controller only provides a basic W, A, S, D movement and doesn't have any options for you to extend any further than that.

The reason for this is that such logic is easy enough for you to do in a few lines of Python code. So it doesn't justify making those components heavily extendable since this can end up making your life more complicated than it was supposed to be. If you want to have an example on how to create such controllers yourself using Python, check the demo project.

[Cmp] Audio Player Component

The audio component is meant to allow you to play a specific audio in the simplest possible way by simply adding this component to an entity. The audio can be 3D or not, you can adjust if you want it to loop or not and also specify the volume and pitch.

A 3D audio will fade off as the camera gets further away from it and it will also alternate between the left and right speaker if the user playing the game have a stereo audio system simulating the 3D effect. Currently, Cave only supports stereo audio up to two channels and it does not support Dolby Atmos 5.1 or further (surround sound).

The audio player component is perfect for background music or sounds or specific sounds that get executed on when an entity owning this component is added or activated to the scene, such as an explosion. If you want more control over the audio you are playing in your game, you can also play them through Python using **cave.playSound(..)** function.

[Cmp] UI Element Component

Cave Engine do have a built-in **Game user interface (UI) system** that is both simple and yet very versatile and scalable at the same time. It is meant for you to be a trivial to understand and provide all the tools that you need to extend it. As we will see in a moment with some examples.



In the image above you see an example of the cave user interface system in the main menu of our game “Crow”. These menu have many sub menus and items, different shapes and buttons and settings and it is all responsive and automatically adjusts themselves according to the screen size, meaning that different players with different monitor resolutions will be able to have a good experience with the menu of the game. **This is all thanks to Cave UI system.**

Each UI element has a **Transform** tab that takes care of **its position and scale** in the screen. This is also relative to its parent UI element. The Transform also have the layer, to allow you to order what gets rendered first to the screen. Last but not least, it also provides a checkbox for you to decide if you want this UI element to work as a scissor or not. If the UI is a scissor, it will cut the drawing of all its child UI elements, allowing you to create more complex menus such as drop downs.

Child and Parent UI elements is known by use the entity child and parent system that you already know how it works and how to use it in the scene graph tab.

The UI element also has a **Behavior** tab that allows you to define how it should look and behave over different events, such as if you want to allow the mouse to hover the button and if you do, if you also want to allow the mouse to click on the button. Each of those two options have their corresponding actions and callbacks for you to add custom logic on hover, on click, etc. You can specify a custom **styling** for each of the element state such as the base color, the color on hover and the color when pressed. This color is not only limited to a single value, it can be an image, have blurred background, alpha and even [9-Slice Scaling](#).

In a real-life project, you probably will have dozens, maybe hundreds or even thousands of buttons spread across your game. So it's a good idea to create a **default Style Asset** in the Asset Browser to pre-specify the style of the buttons and reuse them across the game. This way you will have a unified way to change the style not only for one button but for all the buttons in your game at once, regardless of how many of them you have.

Last but not least, you have the **Text** tab. As the name suggests, it allows you to add text to the UI element and also to adjust the Font Source, Color, Scale, Alignment, Anchoring and all other details that you may expect for rendering Text. Notice that if there is no Font Source, no Text gets drawn. And if you want the UI element to only have a Text and not a Background, you can simply change in the **Behavior** tab to have an Alpha Value of 0.

One more thing that it's worth mentioning about the buttons (*button = UI Element that is clickable*) is that when it comes to logic, they have slightly different behavior than a Python component, for example. Because UI Elements that are clickable are also clickable while the scene is paused. Allowing you to quickly make a pause menu for example.

Adding Logic to the UI Elements:

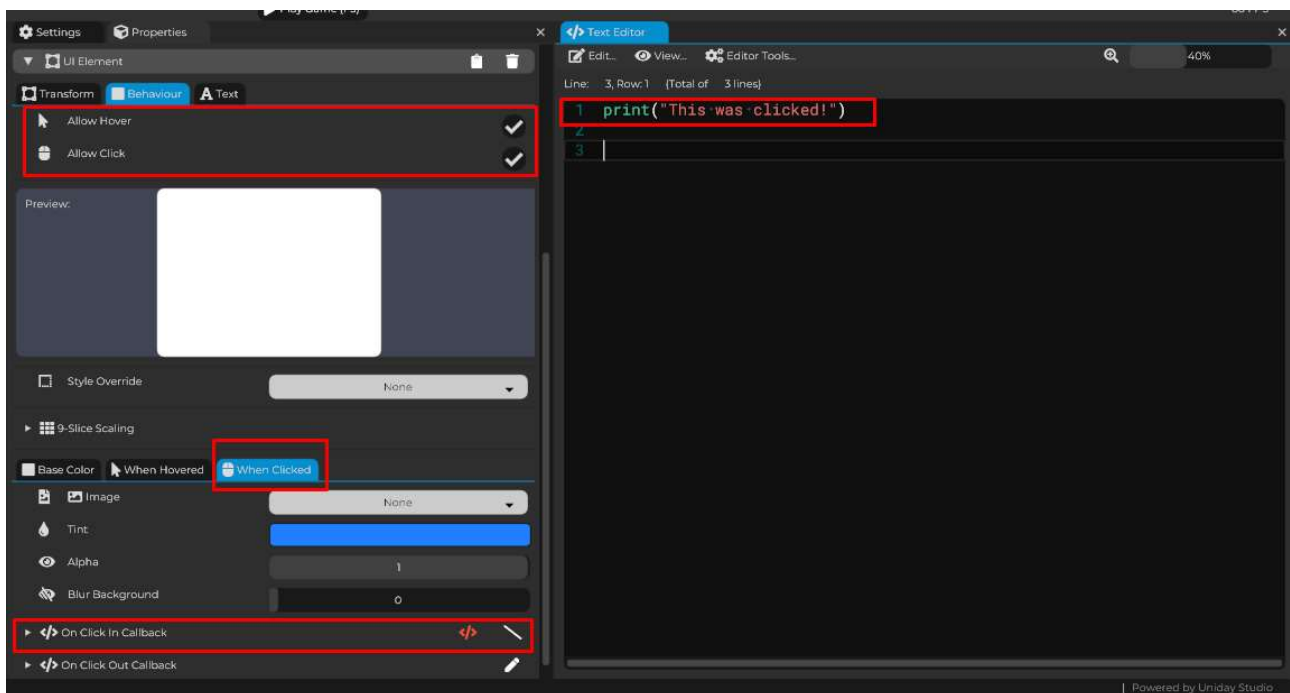
All the logic related to a button or any UI Element that involves hovering and/or clicking **should also be within the UI Element itself**. You will see that hoverable

and/or clickable UI Elements, such as buttons, do have fields for you to add Python code as a callback.

For example, if you want to create a button for your game that does something when the user clicks it, you can right-click the scene graph and add a new entity of type **UI Element**, which is simply a blank entity with the UI element component attached to it, then you do the following process:

- ➔ Go to the “Behavior” tab of the UI element and enable Allow Hover and then Allow Click to make the element Hoverable and Clickable.
- ➔ Then go to the “When Clicked” sub tab and click the Pen icon to edit the “On Click In” callback.
- ➔ Finally, add the code that you want to execute. For example, a print to the console.

This should look like this for you:



You will notice that for both **“when hovered”** and **“when clicked”**, you will have two callbacks, distinguished by the **in/out** name.

When it comes to **“when hovered”**, the first callback **“in”** is called as soon as the mouse enters the UI Element’s area and the second callback **“out”** is called as soon as the mouse leaves its area. For the **“when clicked”**, the **“in”** callback is called when

the user just pressed the mouse button (while hovering the Element) and the “**out**” callback is called when the user released the mouse button.

How to Extend the System?

The game UI in Cave Engine is not meant to create everything pre-made for you, such as drop-downs, sliders, or text input. They are meant to be very simple, yet scalable, to allow you to create any user interface you want and need, and customize it to your own needs. **The UI Element works similarly to a HTML div.** It has a position and a scale, and this position and scale have anchor points, and also allow to be either **relative** or absolute in pixels.

This **relativity** is recursive, meaning that a root UI element is **relative to the screen**, but a child UI element is **relative to its parent**. With this, you can create simply any UI you probably want. If you saw a website doing it, you probably can do it in Cave, because it's very similar to the HTML div, as said, except that it is not code-based as the HTML is, of course. The UI element can have a raw color, or be entirely transparent, or it can have a texture that has settings to allow 9 slicing and other configurations.

You can also add a text to the UI element. And the text have every formatting options that you expect from, well, a text.

Here are some examples to feed your creativity and improve your workflow with Cave's UI Systems:

A **Button** is just an UI element with a text in it. A **Slider** is just an UI element to delimit the slider area with another UI element to represent the slider gizmo inside of it and some Python logic attached to it. So if you click the element it moves horizontally or diagonally according to the mouse position. Please notice that every single function you need to do this is already present in Cave.

A **drop-down** is just a button that, when you click, it activates another UI element, right below it, that have child UI elements for you to select which option you want, and each of those children are just buttons.

[Cmp] Terrain Component



Cave is becoming a very good and powerful option to create open world games.

A crucial part of a game like that is the ability to have terrains with slopes, cliffs, hills and so on. So this is where the terrain components come into play.

In the image above, you see a terrain being created using a combination of many cave systems: The **Terrain Component** as the base and foundation for the entire landscape, custom mega scan **Meshes** as Entities (with mesh components) to represent the rock cliff sides and custom **Shader Program** to create a custom **Material** for the terrain itself. Allowing you to paint roads, different texture variations and so on.

Knowing this is important to understand how the terrain can be broken into multiple different and separate parts that are not necessarily related to each other, but they all contribute to the final look and feel of the environment that you are creating.

In this section **we will be talking about the terrain component in particular**, which is one responsible for the base where the player will stand on and it will also give the rough shape of the terrain. As I said, you can additionally add more entities, meshes and custom materials to make it work the way you want.

Creating a Terrain:

The first thing you need to do in order to create a terrain is to create an empty (an entity with a transform) in the world. Then all you have to do is add a **terrain component** to it, adjust the settings as you want and need them (you will see more about the settings in a moment), create the terrain height map and attach a material to it to be rendered.

If you create a new height map from scratch, it will create a texture with the same resolution as your terrain needs in order to fit and it will pre fill this texture with random perlin noise to create an initial shape for you to start working. Alternatively, **you can also load an existing height map** to the engine and use it as the basis for your terrain by simply selecting it instead. When using an existing texture instead of creating a new one, you will not be able to scoop or modify the terrain unless you click the button to make it local to the terrain. This will effectively create a copy of the existing texture and store it inside the terrain component allowing you to modify it as you want.

***Note:** Cave Engine's Terrain System uses a Height Map based, chunked approach. This approach have some implications, for example: ironically, it does not support creating caves, holes or any concave shapes in it. So, if you need to create a cave in your terrain, you have to create a custom mesh and place it **on top** of the terrain, **not through** the terrain. This approach is used by the engine because besides this implication, it does have a lot of advantages and ends up being overall a best system to fit most cases that the developers will need.*

In order to appear and be drawn, **the terrain must have a material attached to it**. A terrain material is exactly the same as any other regular materials. And as you probably saw at the beginning of the explanation of the terrain component, you can create custom materials specifically for the terrain, writing a custom Shader Program for it to suit your needs.

The terrain component also handles physics for you by default. If you don't want it to have physics, all you need to do is to go to the physics tab and disable physics. If

you don't do this, it should be able to calculate the physics for every dynamic, reached the body on top of it automatically.

Note: Due to the large size of the terrain, if you enable the **“draw debug physics”** in the 3D view by pressing the key 3, cave will not render the terrain physics shape. So if you try to debug the physics and don't see the terrain being drawn, don't worry this is expected.

Now let's understand **how the terrain settings work** for you to be able to adjust it as you need:

- ➔ The terrain **height**, as the name suggests, specifies the height of the terrain going from the minimum point possible to the maximum point. Please notice that the terrain component will also take the transform component into consideration. So if the scale of the entity owning the terrain is not one, this may affect the height and also the size of the terrain itself.
- ➔ The terrain is composed into **components** and each component have a mesh that is simply a grid of quads. The **grid size** in the terrain system specifies how many quads both for the X and Z axis are per grid. A single quad in the grid should represent 1x1 meter (if the empty scale is 1), so a grid size of 80x80 means that each component of the terrain is 80x80 meters.
- ➔ You also need to specify the **number of components** that the terrain have. And this number corresponds to the components in both X and Z axis. So if you have the number of components set to 16, the total components of the terrain will be 256, because it is 16 on the X axis, times 16 on the Z axis.

You may ask yourself: instead of having multiple components of a smaller grid size, let's say 10 components of grid size 100, why not have a single component with grid size 1000? Because in theory the resolution and deterring size will be the same.

And indeed, the visual representation of the terrain will be the same as that for one very important thing: the **terrain system is optimized on a per component basis**.

Meaning that as the camera of your game gets further away from each component, it will be getting more optimized and a lower resolution version of

that particular grid will be used instead to render the terrain. Effectively optimizing the game, improving performance and allowing your graphics card to render the same terrain more efficiently.

So it is important to have the terrain split into different components, which allows the system to function better and optimize the rendering. If you only have a single giant component, it will always be visible and rendered and it will always be at its highest possible resolution, which is not ideal for performance. In other hands, if you have a very large number of components with a very small grid size, it will also not be perfect because the engine will be handling a lot of different smaller parts making the entire system inefficient.

The default values of the terrain component are a good start for you to start fine-tuning the settings to fit your needs. In most cases, they are more than enough and if you need a larger terrain, you can experiment first increasing the number of components, to only then adjust the grid size. If you need more information on how your terrain is being handled, you can use the **debug** tab of the terrain component to enable the bug visualization of the individual components.

***Important:** It is important to know that if you change the terrain size by changing either the number of component or the grid size, you will probably have to use a different height map resolution. So if you change this after creating the height map, you won't be able to resize the map to the new resolution without any loss. So it's good to first figure out the terrain size to only then start painting the height map.*

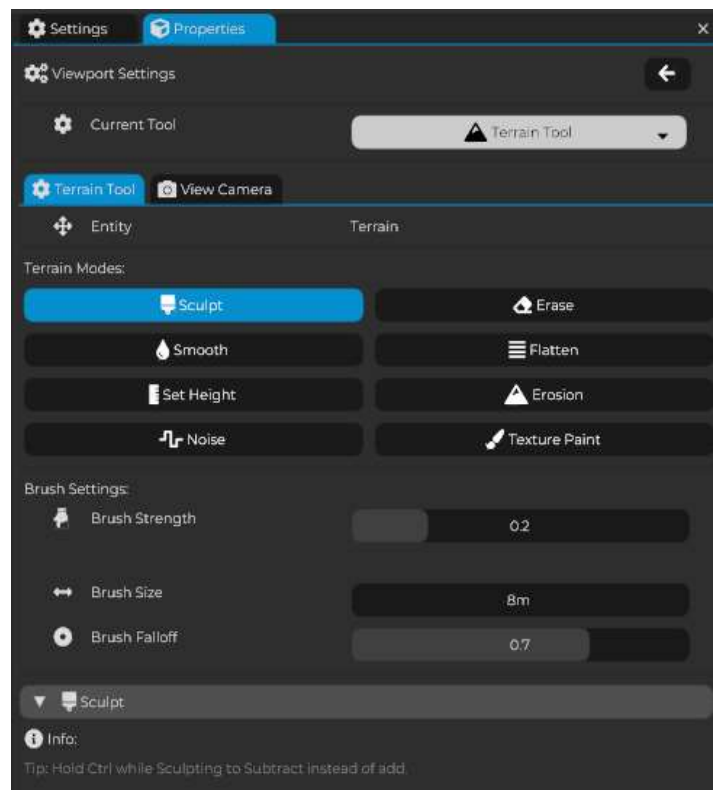
1.8. Terrain Creation Techniques

Now that you're aware of the main engine components and especially the terrain component, let's expand this and understand how we can create terrains. As we already explained, the first thing that you need to do is to create an entity in that the terrain component with the initial configurations and height map.

Note: *It's worth remembering that the height map needs to be local to the terrain component. So if you are using an external image as the height map, you won't be able to paint the terrain unless you click the button to make it local in the terrain component interface.*

Once everything is set up and created, you will find an **"Edit Terrain"** button in the terrain component interface. If you click it, it will switch the 3D view to the terrain tool. Alternatively, you can go in the 3D view top right corner and change the tool that it is currently active to the terrain tool. If you do this by hand, it's worth mentioning that you must have the terrain entity selected first.

As soon as you launch the terrain tool, the **Properties Tab** will be populated with the **3D View Settings** that shows the current tool (which is the terrain tool) and for this tool, all its available settings. You can see them in the screenshot. If you click somewhere else or select another asset and lose this tab, you can always make it appear again by clicking in the gear icon in the top right corner of the 3D view.



Right away you can see that you have a variety of different **Terrain Modes** such as **sculpt, erase, smooth, flatten, set height, erosion, noise and texture paint**. For all those terrain modes you have a universal brush settings that controls the **brush**

strength, brush size and brush fall off. You also have a section below those options for the specific settings to each mode, if apply.

The **Brush** is referring to the 3D Brush that now gets displayed in the 3D View at the location that the mouse is hovering. If you're not seeing it, it's because you may have disabled the line drawing for the engine. Press the key ONE to enable it back.

If you mouse over the terrain and left click it, it will start applying the corresponding Brush to that region of the Terrain. Each brush have its own purposes:

- **Sculpt**: Sculpts the terrain, raising or lowering it.
- **Erase**: Makes the terrain's height be 50% of the total height supported.
- **Smooth**: Smooths out the rough terrain shape.
- **Flatten**: Given the initial position that the Brush is at, flattens the region around it to level with that initial height.
- **Set Height**: Sets the terrain height to a specific, manually typed, height.
- **Erosion**: Applies a simple algorithm to simulate terrain Erosion.
- **Noise**: Applies a random Perlin Noise height variation to the terrain.
- **Texture Paint**: Allows you to paint a custom texture based on the Terrain's UV Map. This is useful to crease texture masks for advanced materials. We will see more about this one in a moment.

It's also very important to know how the Brush settings work. The Brush will always only affect the terrain within its area, determined by the **Brush Size**. If the terrain is out of this area, it should not be affected. In order to smooth the edges between the affected and not affected area, you have the **Brush Falloff**. It is a ratio that, starting from that percentage away from the Brush center, it will start reducing the Brush influence until it reaches zero in the outer edges of the Brush.

Finally, you have the **Brush Strength**, that controls the main influence that the Brush have to affect the Terrain within its area. A strength of zero means no influence and a strength or one means total influence. Notice that most of the times, if you press and hold the mouse down, the strength will actually influence

how “fast” the terrain will be deformed. This is particularly obvious in the Sculpt Mode. In other modes, such as Smooth, the Strength will influence in how smooth the final result is.

By simply using and playing around with this modes you will really be able to create any sorts of terrain you want.

Terrain Texture Paint:

Let’s understand a little bit better how the terrain texture paint mode works. If you select this mode you will be prompted in its settings to select a texture to paint. What this will allow you to do is to, based on the terrain UV map coordinates, paint the texture accordingly. So, if you have a completely black texture, then use the **set color mode** of the terrain paint, select a white color with alpha equals to 1 and use the brush to click in the middle of the terrain, you will see a white dot appearing in the middle of the texture you selected. Pretty straightforward.

But what can cause you some confusion is **how to use this mode and the texture in question**. Because if you plan to use this straight to the Terrain Material, as the Albedo color for example, you will quickly realize that the resolution of the texture will never be enough to cover the terrain. For example: if you have a 2K texture in a 2 km wide terrain, each pixel of this texture will only cover **1 meter** of the terrain, which is very low resolution for today’s standards. So why is this tool here and how to properly use it?

The texture paint is meant for you to paint **stencil masks** for your material. In other words, you ideally want to create a new Shader to use in your material and then, let’s say for example that you have two different textures that you want to blend: a grass texture and a sand texture. So how do you decide which texture gets used into which part of the terrain? You actually need to create a stencil texture and paint them as a filter. The black part of the texture will be used to specify sand regions and the white part of it will be used to specify grass regions, for example.

With this in mind, all of the sudden it makes sense to paint a texture across the entire terrain. Because in this case there’s not much of a problem if each textile of

the texture is one meter long. Because this is not the final texture that the player will see. Instead it's a mask for other textures that can be blended in the shader.

Custom Terrain Shader:

Let's understand practice how to do this using the custom shader program. Before any of that, you need to make sure that you have the terrain properly created and adjusted for your game. Then, that you create a **new material specifically for this terrain** and apply it to the terrain component. This material will be our base, so you can create add a **sand** albedo texture to it and also a normal map for the sand. You can set the roughness to 1 and the metallic to 0 to make this example simpler. Finally, you need to tweak the UV scale of the material to make sure that your sand texture appears in the right size considering the scale of the terrain.

With all that done and your terrain looking good, with sand all over the place, we can finally create a new **Shader Program** to use a stencil mask to paint grass on top of it. Right-click the Asset Browser and create a new shader program. Name it something like terrain shader. Then select the material you created for the terrain, in the shader overwrite field, select the newly created terrain shader.

Go back to the terrain shader and click to edit the **fragment shader**. The first thing that we need to do is to **create three new uniforms** to add our grass albedo and normal map textures and also the mask texture. You can do this by adding the following code in the beginning of the shader code before the main function:

```
uniform sampler2D m_grass;           // Grass Albedo
uniform sampler2D m_grassNormal;    // Grass Normal Map
uniform sampler2D m_terrainMask;    // Stencil Mask
```

Click to recompile the shader in the Shader Program properties and then go back to the material properties, selecting it again. If you scroll down to the end of the properties, you will see a button to **"sync uniforms."** Click it and you will see three new fields appear for you: the **"Grass"** and **"Grass Normal"**, for the grass as we said, and the **"Terrain Mask"** for our stencil mask texture.

Go ahead and add a grass and grass normal to the corresponding fields. Just like with the sand, you can use plenty of online websites to get those images for free.

For the terrain mask, in other hands, you're not going to set an existing texture. Instead, we will create a new one from scratch. You can do this by right-clicking the asset browser again and creating a new texture. The engine will create a default texture for you with a perlin noise pattern by default. It is a good idea now to select this new texture and change its resolution to a proper number that will be enough for you to paint your terrain. I recommend using 1k or 2k textures for this.

With this texture created, go back to the material and assign it to the terrain mask. And then we can finally go back to the shader program to do the last necessary step for our code, which is simply blend the textures together using the Mask we provided.

In the fragment shader, locate the main function (and make sure it's the one for the main pass, not the shadow pass). Right at its beginning, find the part where we are defining what the albedo color looks like (`vec4 albedo = ...`) and also the `gNormal`. Then simply extend this with the code below:

```
// Existing code (remains unchanged):
gNormal = vec4(m_GetNormal(uv), GetDistanceToCamera());
vec4 albedo = texture(m_albedo, uv);

// New Code, to sample the Mask and grass Textures:
vec4 terrainMask = texture(m_terrainMask, fsIn.texCoord);
vec4 grass = texture(m_grass, uv);
vec3 grassNor = m_GetNormalExt(uv, m_grassNormal, m_normalMapFactor);

// Finally blending the colors together, using the terrainMask:
albedo.xyz = mix(albedo.xyz, ground.xyz, terrainMask.r).xyz;
gNormal.xyz = mix(gNormal.xyz, groundNor.xyz, terrainMask.r).xyz;
```

And that's pretty much it for the shader. Recompile it in you will be ready to start painting. Actually, since the default texture that cave creates for you have the perlin

noise pattern, you will probably already start to see the blending going on in a random form on top of your terrain.

Notice that we are only using the red channel of the terrain mask to blend the grass texture. This means that you can reuse the same terrain mask with the green, blue and alpha channels to blend more textures if you want.

Painting the Terrain Masks:

Once the custom shader is created and applied to the terrain, you can select the terrain also again and edit it, going to the **texture paint mode** and selecting your newly created terrain mask to start painting the colors and mask it out. In our example, I recommend that you first set the brush strength to 1.0 and the brush size to a very large number to be able to pre fill the entire texture with a black base color. That way it's easier to reduce the brush size and strength after, select a white color and start painting pathways and everything else that you want in your terrain.

2. How to Architect Your Game in Cave

When creating games using a game engine, it's very important to understand that you have to follow some specific architecture guidelines in order to create and organize the game in a way that the engine can properly interpret it and you can benefit from its architecture to create better games faster and in a more optimized way. Cave allows you to have a lot of freedom when it comes to designing your game. But you still need to understand its core concepts and how to organize your ideas in a way that the engine can understand such as the concepts of scenes, entities, templates, assets, and others.

In this section we will understand how we can structure your game in a way that can work well with the engine's architecture.

2.1. Understanding Levels and Objects

The very first thing that you need to know is to understand how Cave Engine structures its **Objects, Levels**, etc. Before anything, what is even a Level or an Object? **Those are high level concepts** that not necessarily have a matching concept in the engine's architecture, but since they are probably easy to grasp, especially if you're new to game development, we tend to refer to game parts a lot using the concept of levels and objects.

Think about a specific game and everything that you can see, touch, interact, etc. We can start splitting this into pieces by the Level and Object concept:

- ➔ **Level:** A self-contained game environment where gameplay occurs, including terrain, obstacles, enemies, and interactive elements. In an open world game such as GTA, we can initially say that there is only "one" level, which is the world map. In a game with different "parts", we can split it into different levels, different places.
- ➔ **Object:** Any interactive or static entity within a level, such as a player, enemy, item, or trigger.

If you want to create a Game in it, you might want to have different levels, scenes, menus and so on and inside them, having objects with different looks or behaviors.

Imagine you're playing a platformer game where a character jumps across floating islands, avoids spikes, and collects coins. The level is the entire map: floating islands, background, enemies, and coins, essentially the "stage" where the game happens. Within this level, each object is something you can interact with or that affects gameplay, like the character itself, the coins that can be collected, the spikes that cause damage, and even the moving platforms. Levels provide the setting, while objects bring the gameplay to life.

As you can see, these two concepts are very easy to grasp, **but cave does not have levels or objects.**

Why?

While levels and objects are easy to understand, they are very specific to certain use cases and may not represent everything that you want to create in a game. For example, when you launch a game, is the main menu a level? Or the credits? Or even the pause menu? Probably not.

What about the object? Is the terrain that you are walking on an object? Or needs to be something different from the concept of an object? Is it okay calling the terrain the same thing as a box? Can sound weird.

Because of that, Cave uses a **low level concept of "Scene" and "Entity"**. It also have an **"Entity Template"** concept that is essential for you to understand and we will discuss it in a moment as well, but let's start with those first two.

2.2. Understanding Scenes and Entities

You can think of a **Scene** as a playable area. Of course, in a video game, everything is a playable area. Even if the only interactivity you have is clicking a button or pressing a key to go to the next playable area. In this matter, a scene can be a level,

as we described before, but it can also be the main menu, the start menu, the map selection menu or anything else.

In Cave, the scenes are where all your game will happen. There will always be a scene loaded, one at the time, and you can change scenes or reload them anytime you want during gameplay. Think about each scene as each individual level or area of your game.

If your game is a sandbox with only one map like GTA, for example, you should really only have a single scene for the entire map. This is how the scene is supposed to work. If you game have different levels, you should have one scene per level.

Understanding Entities

Entities are everything that composes a given scene. These could represent objects, such as, again, the player, enemy, items, a trigger, etc. But also more abstract things, such as the ground you're standing on, the terrain, the cliffs, the trees, the buildings. Even non-tangible and non-visible things, such as an entity that manages the gameplay, but really does not have any shape or form in the world, but it's there. A folder that you use to organize the other entities, this can also be an entity in Cave.

As you probably noticed at this point, **Entities** can be composed by having different **Components** in it, such as it transforms the mesh, the light, and the python script. And they can also have child Entities.

Entities **are unique to the scene they belong** and cannot be shared across different scenes. If you have to separate scenes, even if you try to create the same entity in both ones, they will be considered duplicates. You will understand how to better handle the situation soon.

But now that you have this basic concept in mind, let's go back to the scenes to finish understanding them in general:

Scenes are meant to be UNIQUE (not copies)

The engine is not meant to have two different scenes that have the same content. If the content is the same, then it should be the same scene. If you want to create a cutscene, the cutscene should be placed in the same scene that you wanted to happen.

This is a very common mistake that beginners to game development in general tend to make. They duplicate the same content over different scenes for different situations, such as cutscenes or different moments of the game. But this is problematic and can lead to a lot of confusion and problems, not to mention that it will inevitably make your project hard to maintain. Since now you will have to keep this duplicated data the same in a manual labor.

So if the map is the same, the Scene should probably be the same.

What about shared objects?

Let's say that you are creating a linear level based game like **Uncharted**, where you progress through different maps in the game. Even though the map is different, meaning that the scene should be different, many things are shared across the different maps that are essentially the same, such as the player, the NPCs, the enemies, weapons, or even the props you find across the map, such as the cliffs, climbable walls, or collectable items. These are all the same objects, but copies instantiated across the game.

If you have different elements in your game that will be shared across different scenes, such as enemies, props, platforms or even a player, this is what the **Entity Template** is meant to do for you. Those objects need to be an entity template that can be easily instantiated over different scenes.

2.3. Entity Templates

Every time you want to create a reusable entity or object, in other words, an object that you want to duplicate over your game more than one time, such as the player, the enemy, the items, doors, or even static things such as the cliffs that you will probably have a bunch in your map, trees, etc, you probably want to make them first an Entity Template. And this is actually a good way to start thinking about them. Every time you're about to create something new for your game, first think to yourself:

- ➔ Do I need **specific Editing Controls** for this (or those) Entity?
- ➔ **Will I be editing it a lot?** (like the Player)
- ➔ Will I need to have **multiple copies of it across the Game**? Either in the same scene or different ones.

If the answer for at least one of those questions is “Yes”, then you should probably create an Entity Template for it.

What's that?

You can think of an entity template as a recipe. Just like a cake recipe that you provide the step-by-step instructions on how to build the cake, the entity template is a way for you to create an **Entity Recipe** with a step-by-step process on how to create such entity. Then, Cave can use it to instantiate a new entity based on it.

The advantage of having a recipe to create the entity is that every time you change the recipe, **cave will automatically update every single entity in your game that uses that recipe**, even if they are spread across different scenes or situations. Meaning every single entity that was built based on that specific **entity template**.

Imagine that you have a **cliff** in your game and since you have a huge game, you have 10,000 copies of this same cliff object spread across all your levels. Then imagine that you found a bug in the cliff physics. Now you have to update the physics for every single instance of the cliff in your project Remember you have 10,000 copies of them!

This will be a nightmare... unless the cliff is an entity template!

Because then, everything you will need to do is to edit the template, fix the physics bug and that's it! Cave will update all the 10,000 instance of it for you.

Another good example is the player. If you have 100 levels, all of them will have the player. Now imagine how hard would he be to update the logic or the behavior of your player if each level had a different copy of it. But if you create an Entity Template for the player, it will be as easy as double clicking the template in the editor, doing the necessary changes and that's it. It will be reflected in all 100 levels you have.

2.4. Creating a Scene and Entities

If you create a default project, it will automatically have a default scene at the root folder in the asset browser. But if you want more scenes, you can right-click an empty area in the asset browser and create a new scene. Then double-click it and you will be able to edit it.

As I said, the scene is almost entirely composed by entities inside of them. The only thing that are scene level is the ambient and sunlight settings, etc. This can be edited by left-clicking the background in the 3D view or clicking edit scene. A scene also has a Boolean to control if it's paused or not. You understand more about what a paused scene means later in this book.

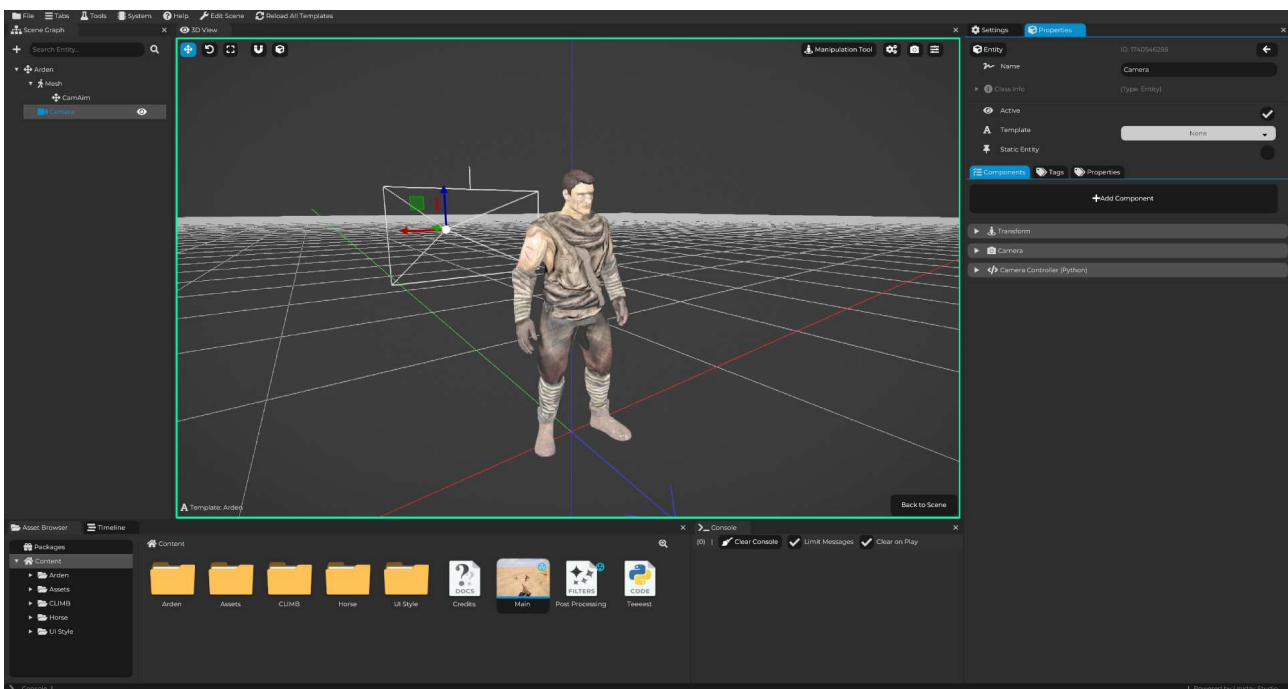
Once the scene is created, you can create entities inside of them **by following the editor basics section** of this book, where we explain further the controls and how to create and manipulate different entities, including handling parenting.

2.5. Creating and Editing an Entity Template

There's a lot of different ways you can create a template. The first and easiest one is to right click an empty space in the asset browser and just like you did with the scene, you can create a new entity template in it. Then just double-click it and you are ready to start editing the template you created.

Alternatively, if you have a local entity in your scene and you want to promote it to a template, you can right click it in the scene graph and click "Promote to template". This will create a new entity template in the current directory you are in the asset browser and you will be able to edit it normally. The only difference is that now the template will be created using the entity you right clicked as the basics and this entity will also be linked to the template. Meaning that it will be an instance of it.

Editing a Template looks almost the same as editing any regular Scene, including with the same controls:



The main difference is that you cannot PLAY the game while editing a Template, since templates are not valid playable scenes. It will be very easy for you to recognize that you are currently editing a template because the 3D view will have

a green outline around it to indicate that and a “back to scene” button in the bottom right corner.

An entity template can only have a single root entity. So if you try to create a new entity in it, you will see that it will always be a child of the root one. And this is necessary by default because when you instantiate the template in your game, it will be of course represented by a single entity, which is the root entity. Remember, an entity in cave can have child entities and this is how you can build your templates.

The root entity of the template cannot have its transform component modified. You will see a warning in the root transform component regarding that. This is because this transforming particular is meant for you to overwrite it when you instantiate the template in the world. Child entities of the root one can have normal Transform operations.

2.6. Instantiating Entity Templates

Now that you created your templates, how to instantiate them in your scenes. This is actually absolutely simple. All you need to do is to go back to editing your scene. Then simply drag and drop the entity template from the asset browser to the 3D view in the position you want it to be instantiated.

In game (during gameplay), in terms of coding, there is no actual difference between a regular entity and an entity that uses a template. Cave instantiates them in the exact same way, as regular entities. The only difference you will find in code is that templated entities are marked accordingly. So you can see if it is templated or not, get the root template entity of the instance, which template it is using, etc.

While the editor will prevent you from accessing or modifying specific values of a templated entity instance, such as its components or child entities, in game you can modify them all via code.

2.7. Modifiable Entity Template Properties

As I just said, you cannot modify any data of an instance from a given entity template locally in the instance itself (other than its root transform). But what if you want to modify something?

For example, let's say that you have an enemy template for your **enemies** and you want to **expose the health and level** of them for you to manually specify which health and level each instance of this enemy you instantly use. So you can add two enemies guarding the entrance of a castle, but they have a much higher health and level to make it harder for the player.

How to handle this case?

Cave allows you to expose as a modifiable value every single public **property** that you add to the **root entity of your template**. So, instead of storing the health of the enemy somewhere in the code, all you need to do is to create a public variable as a **property** (*in the Properties tab of the entity, sub tab "properties"*) in the **root entity** of your template to represent the health and also do the same thing for the enemy level.

A public property is every property that doesn't start with an underline (_). If it starts with it, Cave will consider them private and not show it as modifiable.

Those public properties are exposed for you to locally modify in the Properties Tab, when you select an Entity Template instance in your scene.

3. Understanding the Engine, Editor and Player

Cave Engine is Split into three different parts:

- ➔ The **Game Engine**
- ➔ The **Editor** (Cave Editor)
- ➔ The **Player** (Cave Player, or just “*Game*”)

If you want to properly use the engine, it's very important that you understand why is that, how they work and connect with each other and what is happening behind the scenes. In simple words, the **Game Engine** is a library containing all the Cave code, functions, structures and behaviors and the other two (**Editor and Player**) are standalone applications (*executable*) that uses the Engine (statically linking it) in order to work.

If you open Cave Engine's folder, you will probably find two executables (probably named something like “**Cave Editor.exe**” and “**Game.exe**”) and they reflect exactly that. The Engine itself is inside those two files (a copy for each one) and you won't see it separately.

Why is it like that?

When making a game, there is two steps that are essentials for you to know:

- **CREATING** the Game (*so you can Edit everything*)
- **SHIPPING** the Game (*so others can Play*)

At both steps, the engine itself is very much required, so you can access its drawing, logic, physics, interface routines and so on. But there are some key differences that obviously differs both parts. While when **CREATING** your game, you want a fully fledged editor with access to everything (to see it, modify it and make it work the way you want), this is not what you want in the exported game. Instead, you want to delivery to the players the exact experience you created during the first step. This is exactly why Cave Engine is split like that.

When you're editing the game, you are solely using the **Cave Editor** for that. It is the only one that can open your Source Project and allow you to edit everything. But when you export the game made with Cave, it is only shipped with the Cave Player, that is a straight forward application with the only purpose of playing it and delivering the experience you made to the players.

And before you ask: While the Editor can open the Source Projects, it CANNOT open the exported project files. Meaning that if you export your Game in Cave, it is automatically secured against players trying to open it back and access their sources. Only the Player can read and execute the exported game files.

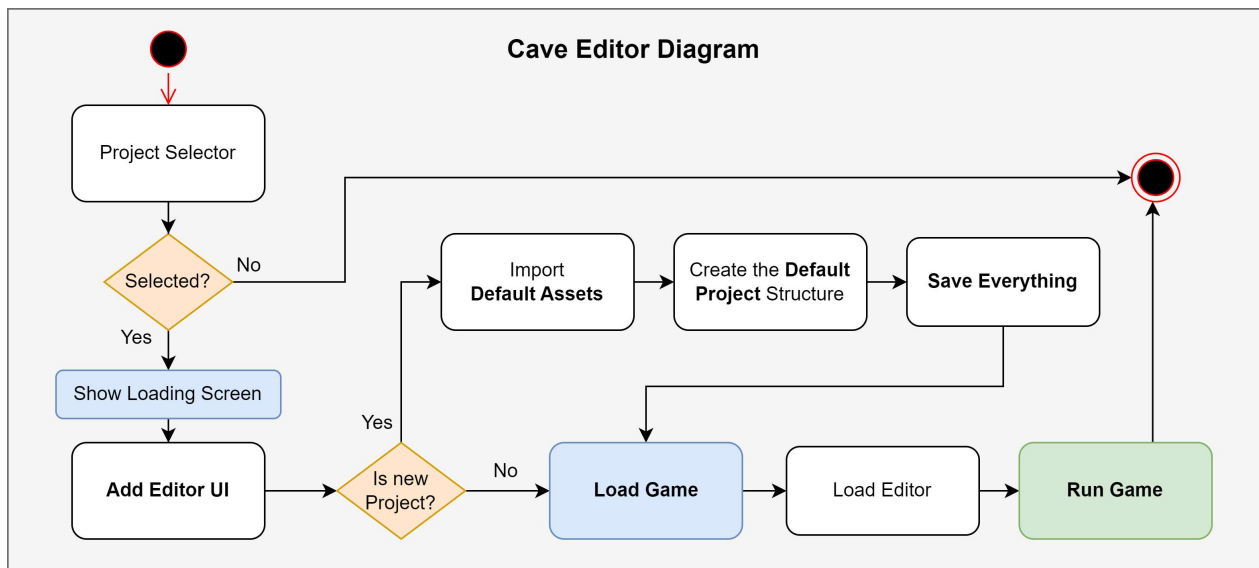
Alright, and how does all that work behind the scenes?

If you're new to Cave Engine and this is your first time using it, you may not be very interested in this part, specially if you're also new to game development in general. But it's my role to explain everything to you, so as you advance and get more experience with it and some questions regarding the engine's internal behaviors arises, you know where to find the answers right away.

I'll briefly explain how the Editor, the Player and finally the Cave's Main Loop works behind the scenes, using **UML Activity Diagrams** (*you can do a quick search on it if you're new to Software Engineer or never heard of it*). Remember that the UML Activity Diagrams always starts at the closed black circle and ends in the open one (with the red circle around it). You can follow the arrows to understand the entire activity flow.

Understanding those activities will hopefully give you a better sense of what is going on and how the Engine is handling your game, as well as some hints on how it works, how you can make things faster and better use the full potential of the Cave Engine.

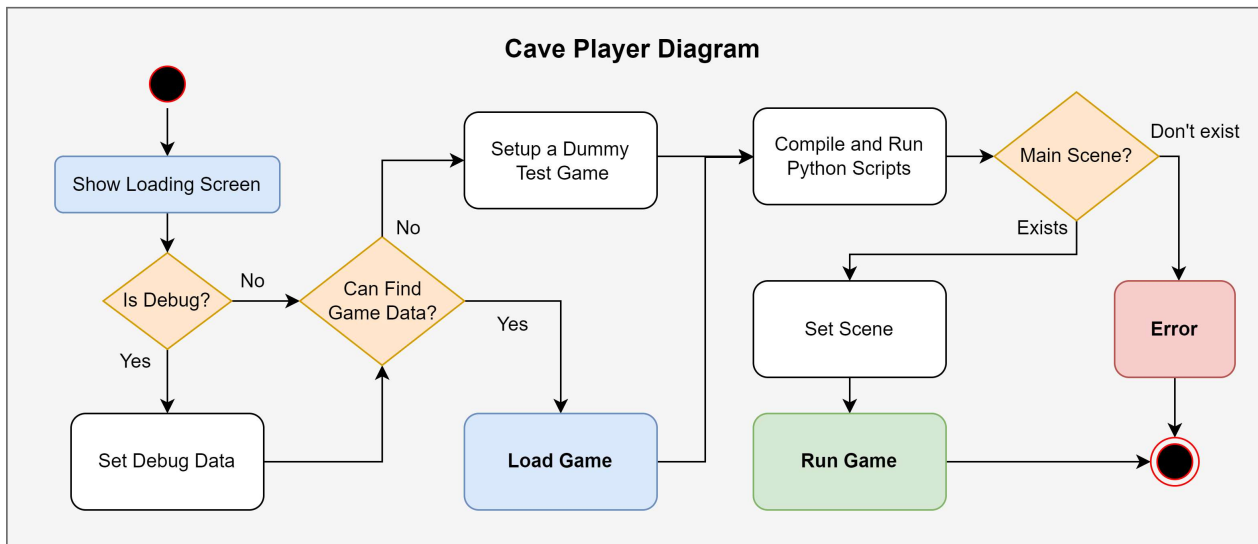
So let's start with the Editor! Here is how the **Cave Editor** Activity Diagram looks like:



Notice that the first thing it does is opening the Project Selector Window, that will show you all your projects and options to create or locate a new one. At this point, you can either close the window (and it does nothing) or open a project (either an existing one or a new one). If you open a project, you can follow along with the diagram to see exactly what are the steps performed by the Editor to properly display it for you to edit.

Also notice that in the end, it reaches the **“Run Game”** symbol. This is where the engine's main loop is located and you will find later on a diagram on how does it works.

The Activity diagram for the Cave Player is very different from the editor. You will notice that it worries about completely different things and it's much more straight forward, meaning that it does not wait for any user input in order to run the game. Here is how the **Cave Player** Activity Diagram looks like:



The ***“Is Debug?”*** decision symbol is checking if the Cave Player was launched by the Editor, when you test the game as a Standalone Project. If so, it needs to load some extra debug data, such as what scene you want the game to start at (and this will override the default project’s scene).

Another interesting thing to notice in this diagram is that the Player is meant to successfully run even if there is no Project at all. In this case (where it can’t find any Game Data), it will setup a Dummy scene (that looks exactly like the initial scene you see when you create a new Cave Project) and run it. This is meant for testing, so if you want to know if a computer can run the game (when it comes to OpenGL requirements and so on), the Player alone is enough of a test for it.

The only failure condition for the Player is when there is a Project, but no Scene could be loaded for it. If you reach here, it is likely a signal that the Game Files are corrupted or missing something.

Last, but not least, notice that the same **“Run Game”** symbol you saw in the Editor diagram is also present here. It is not a similar one, it’s exactly the same and they both refers to the Cave Engine’s main loop routine.

That’s possible because the engine does not care if there is an Editor attached to it or not, it know how to properly behave in both cases. This is a design choice to ensure that everything that works in the editor, also works in the standalone game, with no surprises or incompatibility issues, providing a smoother workflow for you.

3.1. Cave's Main Loop

While the other diagrams were simple, **the Main Loop Diagram** is a bit more complicated, since it is literally the BRAIN of the Engine. You will find all its behaviors, all the Game Logic, Rendering, Physics, etc, updates in it.

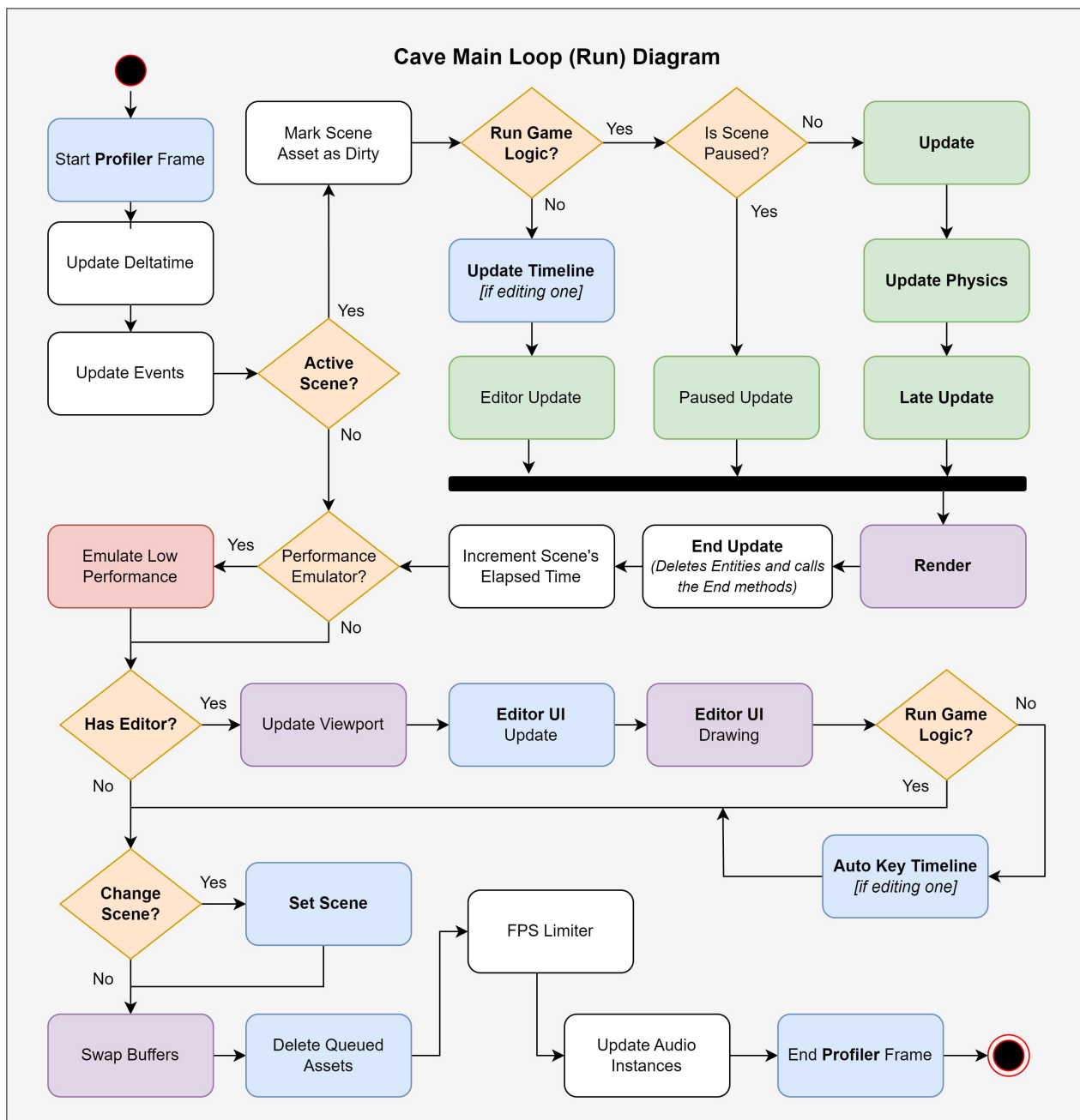
Keep in mind that the **Main Loop**, as the name suggests, **Runs EVERY FRAME**, in a while loop. So as soon as the diagram reaches the end, it goes back to the beginning, to start another engine frame.

This diagram also does not consider other threads, but they do exist in cave, mainly for audio playback and some local multi threaded logical steps, such as preparing the scene for render inside the Render step. I'm omitting them because they won't affect the main engine activity.

In the next page, you'll find the diagram... but beforehand, I would like to advice you about a good thing to keep in mind while reading the diagram:

Pay close attention at **where each step is called** and the order of it, such as the Updates (*Update, Physics, Late Update, Editor Update, etc*) and so on. Also pay attention at the **Branchings** and how they run different logics according to the results. This will help you understand how the engine deals with your project and hopefully you'll be able to make smarter code decisions based on that while creating your game.

This is how the **Cave Engine's Main Loop** Activity Diagram Looks like:



Use this diagram as a reference for your game development process whenever you feel like it's necessary to know the order of which something gets called or if it gets called. It does some important branching that it may be necessary for you to know about.

3.2. Understanding Cave's Internal Signals

You may have noticed that the engine does have some important **methods** that relates directly with **how the Entities and Components are updated every single frame**, to me more specific, I'm talking about the **Update, Late Update, Paused Update, Editor Update**. We also have the **Start and End methods and some others**, but they work a bit different and we will discuss all that, including the update ones in more details, in a moment.

But you need to understand something important first. If you're not familiar with this already (it's fine, now you will be), it's important to know that most of those methods I mentioned are not only a Scene behavior, but also an Entity and a Component behavior.

With that in mind, if you think about this diagram, you may raise the following question:

"Are they (updates) called every frame for everything?"

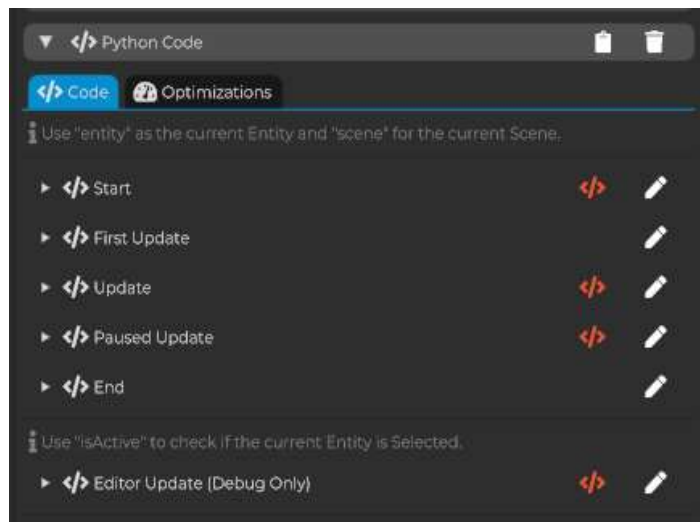
Imagine having a scene with thousands of entities and each of them with dozens of components... Calling the Update, Update Physics and Late Update for them all, every single frame, is probably not the best solution performance wise. That's exactly why cave engine does **NOT** work that way.

Instead, Cave relies on what we call "Signal Systems". Each Scene stores multiple signals for those different methods (*Update, Late Update, Paused Update, Editor Update*) and each signal stores **a list of Components** that needs to be called when the signal is "emitted" (called). That way, each Component can register itself, as needed, to the corresponding signals.

For example, the **Mesh Component** does NOT register itself into any signal, since there is no need to run any code every frame to modify its state.

The **Python Code Component**,

otherwise, will check if you wrote code on each of those methods and if you did, will register itself to their corresponding signals. The example image in the right have a Python Code with scripts written in the Update, Paused Update and Editor Update, meaning that



those three methods will require the Component to register it into their corresponding signals. Also notice that while we have code in the Start method, I didn't mentioned it as part of the signals, since the Start and End methods works a bit different and we will see this in a moment.

Note: *The Signal System is handled INTERNALLY by the engine and you don't have direct access nor need to worry about it. But it's important to know that it exists.*

The signal System is a great Optimization that Cave does in order to dramatically reduce the performance costs of such methods. In our game made with Cave, "Pixel Nightfall", we benchmarked the system and noticed gains all the way up to 30% in frames per seconds without any impact in the engine's usability.

3.3. The Start and End methods

As I mentioned before, those two methods works a bit differently from the other. If you go back to the Main Loop Diagram, you will notice that none of those two are present. Then where are they called?

They are guaranteed to be called a single time when the Entity owning the component is activated (calls the **"Start"**) or deactivated (calls the **"End"**). The Entity is activated or deactivated either by adding or removing (killing) it from the Scene, activating or deactivating it respectively, or by literally just toggling the entity active (using methods such as `entity.activate(scene)`, `entity.deactivate(scene)`, etc).

Important Tip: We already discussed about Entity Activity in the section “**Part 5: Entity Names and Activity**” (33), but as a reminder, if you deactivate an Entity it will call the End method, but if you activate it again, it will repeat the process, calling the Start once again and so on...

Both methods are usually (not guaranteed) called during the **End Update** (check the Main Loop Diagram), which is a Scene only method that handles Entity deletion and/or additions. The Scene ALSO have its own Start and End methods that are called when you set the Scene to be the active one. They do call the Entities corresponding methods too when invoked. In this case, the Scene's Start/End methods are usually (not guaranteed) called during the **Set Scene** block (check the Main Loop Diagram).

3.4. Understanding all the Main Methods

Now let's dive into all the main methods that we are talking about to know a bit more about them and also their typical use case. We may repeat some of the information that we've already discussed before, but that's on an effort to keep the most useful parts in the same place to make it easier for you to look up later. So let's get it started:

[Method] Start

This method is called when the Entity is activated (or added) to the Scene. Every Component have the same method and they're called by the Entity's Start. A typical and very important use case for this one is to do initializations and related initial setups for your entities.

Properties:

- ✓ Called Once when the Entity is Activated
- ✓ Guaranteed to be called

[Method] First Update

This method is called after the Start, but before any Update. **It is guaranteed that this method will always be called AFTER every Entity's Start**, unless you are manually activating existing entities one by one, by hand, in this case, it only guarantees that it is called after every Start of THAT specific Entity.

You can use this method to initialize stuff, just like the start, but when your initializations relies on other Component's being initialized. For example if you have a Scene with an Enemy and a Player component and the enemy needs to store some player data, such as its health or inventory, but this data is initialized by the Player's Start. If you try to access this in the Enemy's start, there is a chance for this data to not have been yet initialized (meaning that the Enemy's start was

called before the Player's start). In this case, it's better to get this data to initialize the Enemy during its First Update.

Properties:

- ✓ Called Once and always after the Start Method
- ✓ Guaranteed to be called

[Method] Update

This is the Main Update method that is called every frame and you can rely on to add your game logic. If you want to write logic to move the player, enemy or anything, this is probably the ideal place to do it.

Properties:

- ✓ Called Every Frame (*if used*)
- ✓ Relies on the Signal System

[Method] Late Update

The Late Update works similarly to the Update, the difference is that it is guaranteed to be called AFTER every Update AND Physics Updates for that given frame.

It is what the First Update is to the Start: If you have some code that needs to be executed after every update, you should aim for this one instead. But it is important that you don't misunderstand the order of calls I just mentioned due to a very important information: **It gets called AFTER the physics update**. What this mean is that if you rely on this method to move Entities around, since it won't get its physics re evaluated until the next frame, you may see it being rendered to the screen with an invalid position, such as inside walls, floating or others.

A good use case for this is when your logic needs to know where and at what state your entities ended up after all their updates and physics evaluations. For example,

if you want to move a camera to the player's position every frame without relying on Cave's parenting system, if you do this during the Update, you may see some jittering occurring, caused by either the Player's update being called AFTER the camera's update, or by the physics updating the Player's position after you set the camera's position. So moving this camera code to the late update may be a good idea.

Properties:

- ✓ Called Every Frame (*if used*)
- ✓ Relies on the Signal System

[Method] Paused Update

If the Scene is paused, it won't call the Update, Late Update or Physics Update. Instead, this one will be called. So if you need to respond to the player's input while the game is paused, such as unpausing it when the user presses "Esc" or animating something in the Scene, this is where you need to put the logic in. The Paused Update never gets called if the scene is NOT paused.

Properties:

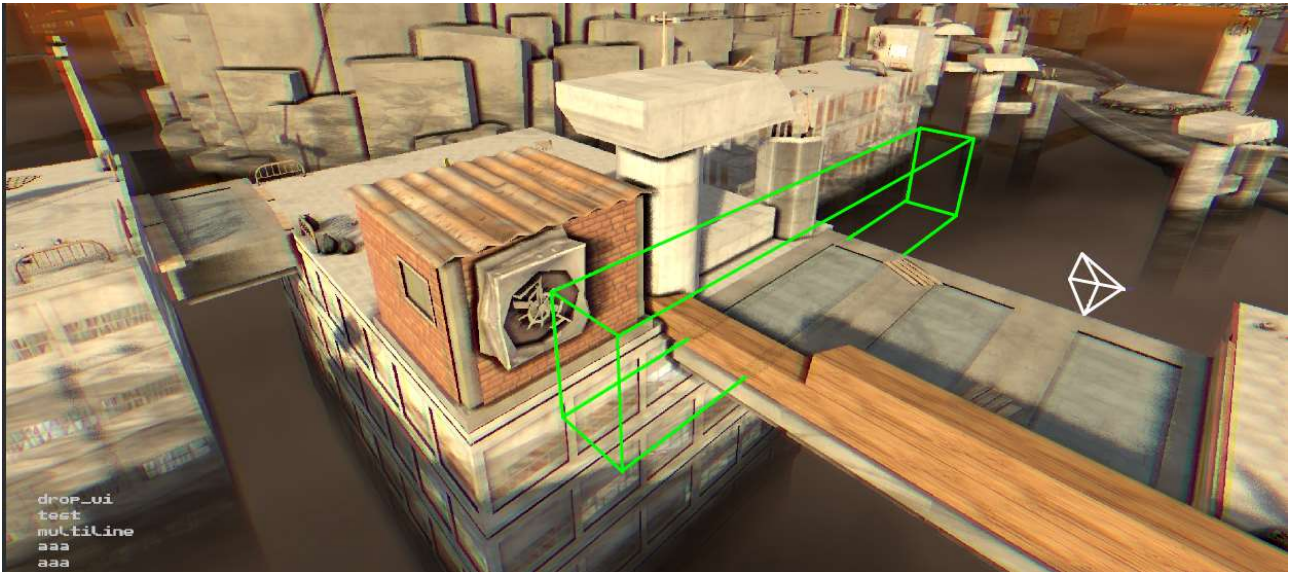
- ✓ Called Every Frame (*if used*)
- ✓ Relies on the Signal System

[Method] Editor Update

This method is meant for Debugging or helping you to develop your game. It is called when editing the Project in the Editor, for the active scene, if the game is NOT currently being played. This means that when you play the game in the editor OR export the Project as a Standalone thing (to ship the Game), this method will NEVER be called.

You can use this, for example, to take advantage of the Scene's debug drawing options to draw some Debug View for your game. For example, in our game, "Pixel

Nightfall”, we use the editor Update extensively to draw some debug shapes for triggers, detection areas for the enemies and so on. Here is an example, showing a Dialogue trigger area:



It helps editing the map and we don't need to worry about it being called during the Gameplay, so it's a good thing. Please notice that while you can rely on code initialization or destruction being performed by the Start and/or End methods, **it's not guaranteed that you will have the same thing for the Editor Update**. Meaning that the engine may (and probably will) **call the Editor Update without ever calling the Start/End methods**. This is only meant for simple debugging, not for extensive logic in the editor. This means that you should not rely on code initialized somewhere else during the Editor Update.

Properties:

- ✓ Called Every Frame (*if used*)
- ✓ Relies on the Signal System

[Method] End

The End method is called whenever an Entity is deactivated from the Scene or deleted (killed). Every Component have the same method and they're called by the Entity's End. Usually you will not use this method a lot, but it may be useful

sometimes, for example, if you have a counter that keeps track of how many enemies were killed or how many coins were collected, you can take advantage of this method to increment those counters when the Enemy or Coin gets deleted, since this action will guaranteed call this method.

Properties:

- ✓ Called Once when the Entity is Deactivated
- ✓ Guaranteed to be called

3.5. Understanding the Internal Methods

Other than the methods discussed, it's also good to understand some of the Cave's internal methods related to the Scene behavior. You won't be dealing with them directly since there is no way to write custom logic that gets executed on them, but they will impact the way your game works.

[Method] Update Physics

This method updates all the physics state of the Scene, applying the necessary gravity and forces to each Dynamic Rigid Body or Character, resolving the collisions and the Constraints and so on. Any changes in the Entity's transforms you made before this (including adding new Entities) will only guarantee the Entity's physical accuracy after the Update Physics gets resolved.

Meaning that if you place an Entity inside a wall or directly colliding with something during the Start, Update, First Update methods, this collision will only be resolved after the Update Physics.

Advanced Tips:

→ **Tip 01:** *Modifying the Entity's transform will often NOT update its Physics representation in the World. So you may experience an "issue" where you change an Entity's position but the physics (and the rendering) does not react accordingly. That's an expected optimization that cave does to save processing every*

***entity every frame.** To resolve that, you should explicitly call the entity's **`.submitTransformToWorld()`** method.*

→ **Tip 02:** While **Collision Resolutions** are only calculated during the Update Physics, **Collision Detections** can be calculated whenever you need them. Check the **Rigid Body Component's collision** methods in the API for more details. You can also check the **Scene's "check contact"** methods as well as the **ray and sphere casts**, that relies on the physics world and can be called any time.

You should plan carefully how you deal with physics to avoid incorrect behavior, taking all this information into account.

[Method] End Update

When you add or remove an entity to a scene in cave, **this action is not immediately performed**. Instead, they are queued to be executed by the end of the frame. This is necessary because you may ask for the engine to add or remove the entity WHILE iterating over the existing list of entities, either directly, via Python Code, or indirectly, since the engine does iterate over this list one way or another for certain actions that may allow you to execute code.

In order to avoid changing the list during iteration, which may cause a crash, those actions needs to be queued for later execution. But for you, this also means that the **End** method of the Entity will be called LATER (during the End Update) when you request it to be killed (deleted). The same happens when you add an Entity: their **Start and First Update** will only be called later in the end of the Frame.

The Scene's End Update method is responsible for exactly that. It first calls the End for all entities queued to be removed, then remove them all, add the ones queued to be added and calls their Start methods, followed by their First Update methods.

Tip: If you add multiple entities on a single frame, it's guaranteed that all their Start methods will be called BEFORE their First Updates.

4. Cave's Audio System

Audio is a very important part of a Game, representing 50% of the player's experience, so cave needs to have a good and versatile audio system to handle all that. In this section I'll discuss the basics of how the Audio System works, when it feeds data to the audio device and other useful information related to it. In order to continue explaining the Audio System, let's make sure that you understand the naming conventions we use:

An **Audio Track** is a Cave Engine **Asset** that stores the Audio information to a specific sound such as a music, a footstep, etc. We also commonly refer to this as "Sound" or just "Audio". Once you play an Audio Track (via `cave.playSound(...)`, for example), the engine will return to you a handler for that specific play called **Audio Track Instance**.

The **Audio Track Instance** allows you to have control over a specific Sound you played, such as pausing or unpausing it, stopping it completely, changing the volume, pitch, 3D position (if enabled), etc. It's a weak reference handler, so if you don't store it, the engine will still internally keep track of it and continue to play the sound as you initially configured.

Sample Rate, in short "SR", is how many samples per seconds a given Audio has. That said, resampling is the logic of changing a given audio sample rate to match another sample rate (such as the master SR).

The engine creates an internal **Audio Device** class to communicate with the machine's audio outputs (speakers, etc) that have a **Master Sample Rate of 44.100 Hz**. You can play any audio you want via Python API (`cave.playSound(...)`) or the **Audio Component** and Cave's Audio System will automatically handle everything for you:

- ✓ Mixing the Audio Tracks (*as many as you want to play at once*)
- ✓ Audio **Resampling** (*if the audio sample rate is different from the Master*)
- ✓ **Pitch** Adjustment (*default Pitch is 1.0*)
- ✓ **Volume** Control (*cave's volume ranges from 0.0 to 1.0*)

- ✓ **3D Audio** (*if you provide a valid Entity or Position to the instance*)
- ✓ Audio Instance **State** (paused/unpaused, etc) and **Loops**

If you go back to the Cave Engine's Main Loop Diagram, you'll find a spot near the end of it where it updates all audio instances. This is the part where it actually syncs all the 3D audio data with each corresponding instance to make them look 3D.

Internal Audio System Behavior: It's important for you to know a bit more about how the Audio System works internally. When you play an Audio Track, the system creates a new internal Handler (that can be referenced by the **Audio Track Instance**) with all the necessary settings for that specific play (like a reference to the Audio Track itself, how many loops, volume, pitch, current time, etc) and this handler gets added to a playing queue. This queue contains a list of all sounds currently being played and it is used to feed the Audio Device's buffers with data when requested. This request is usually made by the Operating System and Cave Engine listen to it in another thread, so the main thread can keep running without any problems regardless of the Audio System being busy or not.

The main thread only needs to keep in sync with the audio thread when you play or modify an existing audio instance, because it needs to be added to the queue. The engine handles that sync for you automatically.

5. Shaders and Materials

Like most engines give, provides you the option to create custom **Materials** to specify the looks of your game. And also, custom **Shader Programs** to specify how the materials will be rendered to the screen. We will discuss them in depth here, with a special focus in the Shader Writting.

5.1. Cave Shader Programs:

Cave Engine uses [GLSL](#) as the shading language for all its shaders. There are two main types of shaders in the engine:

- The **Material Shaders**, used to shade Materials.
- The **Filter Shaders**, used for post-processing.

We are not going to talk about the filter shaders here, but the **material shaders**, which is the shader created when you right click the Asset Browser and choose to create a new **Shader Program**. This is the shader used by the engine to determine both the position that each vertex of a mesh should be rendered at and also the color of each Pixel* (also known as **Fragment**) on the surface of a given mesh should have.

The material shader is where you will do all the lighting and shadow calculations and so on. Cave supports having a **Vertex Shader** and a **Fragment Shader**, also known as the pixel shader.

Shader Override Ordering

This sections covers the engine decision making of picking which shader to use to draw each element.

If you don't provide any shader, the engine will use its default shader to render all the materials, but you can customize this in **3 different places**, each having a higher priority of usage:

1. Project Shader Override

The first way to customize a shader is in a project level, meaning that you can write your own shader that will be used as the default shader to **every single material of the game**. Very useful if you want to completely get rid of the default shader of the engine and write your own.

2. Scene Shader Override

Alternatively, you can create a custom shader **to a specific scene**. This will override the project shader (if any), and have a higher priority when rendering the objects of that given scene. Useful, again, if you want to override every single shader of a particular scene, but not to your entire project.

3. Material Shader Override

Last but not least, you can specify a custom shader **to a specific material**. If a material do have a custom shader, this shader will have a higher priority than the other two mentioned, meaning that they will be used instead of them. A material level shader is one of the most common practices that you will probably encounter and use to do custom and local shaders to specific things, such as **foliage, water, flames, special effects, and so on** in your game.

Uniform Exposition:

When you create a shader uniform:

```
uniform int m_test;  
uniform float m_example = 3.0;
```

It will be exposed as a customizable (and serializable) parameter in the Material using the given Shader.

5.2. Shader Naming Conventions:

As any software, Cave also have its own naming conventions. While you're not obligated to use them, since you can create your own, it's good to state them here. Because you will find that all the code that you get by default by the engine follows those conventions. So if you want to keep consistency across your code you can follow them as well.

For naming Shader variables, use **camelCase**, meaning that the variables should start with lower case and have upper case characters to separate multiple words. Example:

```
uniform int multiWordVariable = 1;
```

To name shader functions, use **PascalCase**, meaning that the function should start with upper case and also have upper case characters to separate multiple words. Example:

```
vec3 GetColorRed(){  
    return vec3(1, 0, 0);  
}
```

Material Uniforms

Cave prefixes every Material Uniform with the prefix **m_**. This is a simple way to identify them since GLSL doesn't support namespace and it's also not possible to store every material variable in a struct (it's known to cause driver issues having **sampler2Ds** as struct members).

So if you see a variable prefixed with **m_**, you can assume it's a **Material variable**. Consequently, we recommend you to give the same treatment to your own custom material variables. Example:

```
uniform sampler2D m_albedo;  
uniform float m_alphaValue = 1.0;
```

Editor: Uniform Visibility

While editing a Material, you are mostly essentially tweaking its shader uniforms. Because of that, Cave Engine provides you some **useful "tricks"** to customize how the uniforms will appear in the editor.

If you have a **private or dynamic** uniform that you don't want it to be exposed in the editor, prefix it with a single underline (). This is useful when you want to create a variable that is only editable through code. As an example, cave have a built in internal uniform that is passed to every shader that looks like this:

```
uniform float _timer; // To use it, simply define this in your shader
```

It contains the elapsed time of the game and it's ALWAYS updated by the engine (once per frame). So it's not meant to be edited by hand. Hence why it starts with an underline.

Warning: While single underline prefixes are allowed, **do not use double consecutive underlines** (as we do for private variables in Python), since those are reserved in GLSL and may lead to shader compilation errors in some GPUs.

If your uniform name ends with the word **Factor**, Cave will display them in the editor **as a 0 to 1 slider**. It works with the variable types **float, vec2, vec3, vec4** and **sampler2D**. For the sampler2D, it will still provide the option to set a custom RGBA texture, but when not set, it will present a single slider to adjust the RGBA values all at once. Example:

```
uniform sampler2D m_roughnessFactor;  
uniform float m_alphaFactor = 1.0;
```

To make a **vec3** appear as a **RGB Color Picker** or a **vec4** as a **RGBA Color Picker**, simply include the word **Color** anywhere in its variable name:

```
uniform vec3 m_skyColor = vec(0,0,1);  
uniform vec4 m_extraColorOption;
```

You can also add comprehensive Comments to explain a uniform and it will appear in the editor when the user mouse over it.

```
uniform float m_test; // This comment will appear in the Editor!
```