Portfolio Item 1:

Journal about my learning process of C++ for Unreal Engine 5

Introduction:

When choosing how to approach this item I had 2 ways to do it. First, dedicate a few days and watch 8 hours of video each day and one day for putting all the info together. Second, spend 2 hours a day on videos while doing notes for about two weeks and then based on notes create the journal. After some thinking I decided to stick to the second option to not overwhelm myself with information, screentime and have some time to process the material in between sessions.

As the title says this part is dedicated to my journal about C++ for Unreal. The journal will consist of 22 parts, one for each watched video. I decided to go this way about it so that I and other possible future users can easily find the needed information and the original video. Each part is going to have same structures for further ease of use. Each part is going to consist of following elements: *video title*, *video link*, *short overview of the info covered in video*, *explanation of key information and instructions, and some personal notes*.

Some terms and elements maybe not be familiar to the reader with no Unreal Engine knowledge, so be ready to look up some information or watch an introduction video about the engine to be prepared **link: video.**

Playlist my playlist of the watched videos in order: Playlist

Video 1:

Title: Creating A Project
Link: Watch on YouTube

The video covered:

- The concept of actors as objects that can be placed or spawned in the world.
- How to create a new C++ class derived from *AActor*.
- How to add components to the actor and set its properties.

Explanation

An actor is any object that can be placed into the world. This includes characters, props, cameras, and more. Actors can have components attached to them to define their behavior and appearance.

Creating a Custom Actor

To create a new actor:

1. In the Unreal Editor, go to **File** > **New C++ Class**.
2. Choose **Actor** as the parent class.
3. Name your class (e.g., MyCustomActor) and click **Create Class**.

This will generate two files:

- *MyCustomActor.h*: The header file where you'll declare your actor.
- *MyCustomActor.cpp*: The source file where you'll define the behavior of your actor.

Adding Components to the Actor

To give your actor functionality or appearance, you can add components such as meshes or lights.

Notes

- *AActor* is the base class for all actors in Unreal Engine.
- *PrimaryActorTick.bCanEverTick = true;* enables the actor to execute code every frame.
- *BeginPlay()* is called when the game starts or when the actor is spawned.
- *Tick()* is called every frame.
- *ProjectName.Target.cs is a file in which you can add external libraries*.

Video 2:

Title: Actors & Components & Custom Move Component
Link: Watch on YouTube

The video covered:

- The concept of components as modular pieces of functionality.
- How to create a new C++ class got from *UActorComponent*.
- How to add the custom component to an actor.

Explanation

Adding Components to the Actor

Components add functionality or appearance to actors. For example, a *UStaticMeshComponent* can be used to render a 3D model. In the actor's constructor, you can create and attach components.

Implementing a Custom Movement Component

A **Movement Component** defines how an actor moves. To create a custom movement component:

1. Create a new C++ class derived from *UMovementComponent*.
2. Override the *TickComponent* function to define custom movement behavior.
3. Attach the movement component to your actor.

Notes

- Launch the project using code editor to avoid issues on start.
- Typing "*this*" shows all available functions for the current object in script.

Video 3:

Title: Unreal Engine C++ Tutorial #3 - Creating a Custom Scene Component
Link: Watch on YouTube

The video covered:

- How to create a custom editor module in Unreal Engine 5 using C++.
- Visualization of custom components within the editor viewport
- Enhancing the editor experience by adding visual cues for custom components.

Explanation

Creating a Custom Editor Module

Unreal Engine allows developers to extend the editor's functionality by creating custom modules. These modules can add new tools, visualizations, and editor behaviors.

Visualizing Custom Components

By implementing custom visualization logic, we can provide visual representations of our components directly within the editor viewport. This helps in understanding and debugging component behavior during development.

Notes

- Don’t run the project while editing.
- Keep the live coding off to avoid compiling errors.

Video 4:

Title: Blueprints & C++
Link: Watch on YouTube

The video covered:

- Creation of a custom movement component in C++.
- Integrating the custom movement component with Blueprints.
- Balancing the use of Blueprints and C++.

Explanation

Creating a Custom Movement Component

A **Movement Component** in Unreal Engine defines how an actor moves. By creating a custom movement component in C++, I can define specific movement behaviors fitting my game's needs.

Integrating with Blueprints

After creating the custom movement component in C++, I can expose its properties and functions to Blueprints using macros like *UCLASS(Blueprintable)* and *UFUNCTION(BlueprintCallable)*. This allows to tweak movement behaviors without modifying the C++ code.

Notes

- *UMyCustomMovementComponent* is a custom movement component derived from *UMovementComponent*.
- *TickComponent* is overridden to define how the actor moves every frame.
- *SetMovementSpeed* is a Blueprint-callable function that allows designers to change the movement speed at runtime.

Video 5:

Title: Custom C++ Actors
Link: Watch on YouTube

The video covered:

- How to create a Blueprint Function Library in C++.
- Exposing static utility functions to Blueprints.
- Utilization of these functions within the Blueprint editor for enhanced functionality.

Explanation

A Blueprint Function Library is a collection of static functions that can be called from any Blueprint in your project. These functions are not tied to any specific object and are ideal for utility or helper functions that perform general tasks.

Why Use C++ for Blueprint Function Libraries?

Implementing these libraries in C++ allows for:

- **Performance:** C++ functions can execute faster than their Blueprint equivalents.
- **Reusability:** Functions can be used across multiple Blueprints without duplication.
- **Maintainability:** Centralizing utility functions makes the codebase easier to manage.

Notes

- *UBlueprintFunctionLibrary* is the base class for creating function libraries accessible in Blueprints.
- *UFUNCTION(BlueprintCallable)* exposes the function to the Blueprint editor, allowing it to be called within Blueprint graphs.

Video 6:

Title: C++ Interface
Link: Watch on YouTube

The video covered:

- Create and utilize Blueprint Function Libraries in Unreal Engine using C++.
- Expose static utility functions to Blueprints for reuse across various Blueprints.
- Implement functions that can be called within the editor using the CallInEditor specifier.

Explanation

Blueprint Function Libraries

Blueprint Function Libraries are collections of static functions that provide utility functionality not tied to a particular gameplay object. They are ideal for functions that need to be reused across multiple Blueprints.

Creating a Blueprint Function Library in C++

To create a Blueprint Function Library in C++, you define a class that inherits from *UBlueprintFunctionLibrary*. Within this class, you can define static functions and expose them to Blueprints using the *UFUNCTION* macro with appropriate specifiers.

Notes

- All functions in the Blueprint Function Library must be static, as required by Unreal Engine.
- Use *CallInEditor* to call the functions in editor.

Video 7:

Title: Custom Player Pawn
Link: Watch on YouTube

The video covered:

- How to create a **custom Pawn** class in C++, from *APawn*, to give the player a controllable character.
- Setting up a *root component* (such as a *USceneComponent* or capsule) and attaching visual components (e.g., a mesh or camera) to it.
- Overriding S*etupPlayerInputComponent* to bind input axes and actions so the Pawn responds to keyboard/gamepad controls.
- Using a *UPawnMovementComponent* (or a custom subclass) to handle movement logic, letting us call *AddMovementInput* for smooth locomotion.

Explanation

A Pawn is a special *AActor* that can be possessed by a *APlayerController*, giving that controller direct input and movement control (think of it as the player's "body" in the game world).

1. Create a C++ Pawn Class
2. Create And Add Key Components
3. Set Up Movement And Bind It

Notes

- Don't forget to have an include for each used component.

- Keep in mind SetupPlayerInputComponent runs **before** *BeginPlay()* and only after the Pawn is possessed.

Video 8:

Title: Random Actor Spawning
Link: Watch on YouTube

The video covered:

How to spawn actors at runtime using *UWorld::SpawnActor*.

Exposing a *TSubclassOf<AActor>* property so designers can pick which actor to spawn in the editor.

Generating random spawn locations (within a radius) and rotations (yaw) using *FMath::RandRange*.

Invoking spawning on BeginPlay, via timer, or in response to player input.

Explanation

By default, actors you place in the level are static. To create dynamic gameplay—like enemy waves or collectibles—you use C++ to tell the engine, "Please make me a new actor now."

1. Expose a Spawnable Class, so you can choose any actor type in the editor dropdown.
2. Pick a random transform, SpawnRadius and SpawnHeight are editable parameters.
3. Call SpawnActor
   Run this in BeginPlay(), a timer callback, or an input-bound function.

Notes

- Call the spawn only after *GetWorld()*.
- Spawn actors on a static height above the ground to prevent clipping.

Video 9:

Title: Random Actor Spawning
Link: Watch on YouTube

The video covered:

- How to transfer a player class from deriving **APawn** to **ACharacter** to leverage Unreal's built-in movement, collision, and animation features.
- Replacing a manually created root component and floating movement with the Capsule, Skeletal Mesh, and Character Movement components.
- Binding input to call Movement Input and Jump on the character.
- Configuring capsule size and mesh orientation.

Explanation

An *APawn* gives you full control but requires you to wire up physics, collision, and input from scratch. *ACharacter* is a specialized pawn that already includes:

- A *Capsule* for collision.
- A *Skeletal Mesh* (named "Mesh") for rendering and animating your character.
- A *Character Movement Component* that handles walking, jumping, gravity, and networking.

By switching your class to inherit from *ACharacter,* you instantly gain smooth, replicated movement without writing low-level physics code.

Notes

- Ensure the mesh origin alignment point is correct to avoid visual bugs.
- Check the capsule sizing to have proper collision.

Video 10:

Title: The ItemData struct and data tables
Link: Watch on YouTube

The video covered:

- Creating and defining data row structure.
- Creating and preparing a spreadsheet.
- Importing CSV as a data table.
- Look up of the data in runtime.

Explanation

1. **Define Your Row Structure**
   This struct mirrors each column in your CSV.
2. **Create & Import the CSV**
   - First column: row names (unique identifiers).
   - Next columns: *ItemID, DisplayName*, *Icon*, *Weight*, *Value*.

- Right-click **CSV** in **Content Browse**r → **Create DataTable**... → select **FItemRow**.

3. **Expose the DataTable**
   Assign the imported DataTable asset in your manager or GameMode.
4. **Lookup Rows at Runtime**

Notes

- Column headers must exactly match the *UPROPERTY* names.
- Set the *ItemDataTable* in the editor leads to not get a null-pointer error.

Video 11:

Title: The ItemBase class
Link: Watch on YouTube

The video covered:

- Creation of *AItemBase* class to represent in-game items.
- Easy way to select a desired row from *DataTabel*.
- Runtime item data lookup.
- Visual & collision setup for itmes.
- Pickup behavior for items in game.

Explanation

1. Create the Item Class
   - Make a new class called *AItemBase* that extends *AActor* (which means it will be a physical object in the game).
   - Add *MeshComponent* (visualization), *SphereComponent* (collision detection).
2. Connect the Item to a Data Table
   - Add a FName ItemRowName property to store the item's reference in the DataTable.
   - When the game starts (BeginPlay()), find the item's data using its ItemRowName.
   - Extract item details (like icon and mesh) from the row.
3. Apply the Item's Properties
   - If the row has an icon, use it in the UI.
   - If the row has a mesh, apply it to MeshComp so the item appears properly.
4. Detect Player Interaction & Handle Pickup

- Make SphereComp react when the player enters its area.
- If the overlapping object is the player, add the item to their inventory and update the UI.

Notes

- *FName* used in the CSV and the *ItemRowName* must match to look up data.

Video 12:

Title: The Interaction Interface
Link: Watch on YouTube

The video covered:

- Creating an interaction system of items.
- Defining interaction methods.
- Applying methods to objects.
- Detecting and triggering interaction with line trace (similar to Unity's raycast).

Explanation

1. Create an Interaction Interface
   - Make a *UInteractable* interface so objects can be interacted with.
   - Add *IInteractable*, the C++ version, to define interaction behavior.
   - Create an *Interact()* function that lets the player trigger actions.
2. Apply to an Actor (e.g., Door)
   - Make a DoorActor that inherits *IInteractable*.
   - Override *Interact_Implementation()* to define what happens (e.g., opening/closing).
3. Detect Interactable Objects in Player
   - Each frame, check if the player is looking at an interactable object using a line trace.
   - If an object is found, store it and display "Press E to interact".
4. Trigger Interaction on Input
   - When the player presses E, call *Interact()* on the stored object, activating its behavior.

Notes

- Check if the current interactable is valid to avoid null exceptions.

Video 13:

Title: User Interface Work Part 1
Link: Watch on YouTube

The video covered:

- Building the inventory UI.
- Connecting data in C++ with UI elements.
- Filling the inventory list with entries.
- Making the UI show and hide on key input.

Explanation

1. Create the Inventory UI
   - In the Content Browser, add a Widget Blueprint named *WBP_Inventory*.
   - In the Designer tab, add a ScrollBox called *ItemList* to hold item entries.
   - Make a separate *WBP_InventoryEntry* with an Image (for icons) and *TextBlock* (for name/count). Check "Variable" to expose them.
2. Connect to C++
   - Create a *UInventoryWidget* class.
   - Use *BindWidget* to link *ItemList* to the UI.
   - Define a *RefreshInventory()* function to update the inventory UI when needed.
3. Fill the Inventory UI
   - *RefreshInventory()* clears *ItemList* and adds entries dynamically.
   - Each item gets an *InventoryEntry* widget with its icon and name set.
4. Show & Hide Inventory
   - In *AMyPlayerController*, create *ToggleInventory()*.
   - If the inventory UI doesn't exist, create it and add it to the screen.
   - Toggle visibility when the "I" key is pressed.
   - If opening the inventory, refresh the list with the player's items.

Notes

- C++ *BindWidget* properties must exactly match the widget's "Name" in UMG
- Refresh the list only when the widget is first shown.

Video 14:

Title: The Pickup Class
Link: Watch on YouTube

The video covered:

- Building an actor for pick up mechanic.
- Setting up item data for pick up.
- Defining the spawn areas of items.
- Implementation of timed and random spawning.
- Editor visualization to preview spawn boundaries.

Explanation

1. Setting Up *APickup*
   - Add a *UStaticMeshComponent* for the item's visuals.
   - Add a *USphereComponent* for detecting player interaction.
   - In *BeginPlay()*, check *ItemRowName*, find the item in the *DataTable*, and apply its mesh using *MeshComp->SetStaticMesh(...)*.
2. Creating *ASpawnVolume*
   - Define *MinExtent* and *MaxExtent* to set spawn area size.
   - In *OnConstruction()*, draw a green wirebox to visualize the spawn zone in the editor.
   - In *BeginPlay()*, start a looping timer to call *SpawnPickup()* repeatedly.
3. Spawning Items
   - In *SpawnPickup()*, generate a random spawn location inside the volume using *FMath::RandRange()*.
   - Place the pickup at that location using *SpawnActor<APickup>(PickupClass, SpawnLoc, FRotator::ZeroRotator)*.

Notes

- Timer for timed spawn shouldn't be set before the existence of the world.

Video 15:

Title: InventoryComponent Part 1
Link: Watch on YouTube

The video covered:

- Making an inventory component for the actor.
- Tracking stored items and their visualization in the editor.
- Creating inventory functions, like adding and removing items.
- Handling refreshing of the inventory on update.
- Setting up inventory for cleaning and loading saved inventory.

Explanation

1. Keep the Player Class Organized: Instead of cluttering the character with inventory logic, use a *UInventoryComponent* to handle everything.
2. Make Inventory Easy to Use: Functions like *AddItem()* and *RemoveItem()* are *BlueprintCallable,* allowing designers to manage inventory without coding in C++.
3. Automatically Update Other Systems: The *OnInventoryUpdated* event ensures that UI widgets and other Blueprints react whenever the inventory changes.

Notes

- *FItemRow* has a valid *operator*== or row name is used so that delete function removes the correct element.

Video 16:

Title: InventoryComponent Part 2
Link: Watch on YouTube

The video covered:

- Attaching the inventory component to the player.
- Saving and loading inventory between levels.

Explanation

1. Setting Up the Inventory Component
   - In the player's constructor (or Blueprint parent), add the *InventoryComponent*;
   - Mark it *BlueprintReadOnly* so UI and Blueprints can access it.
2. Saving & Loading Inventory
   - Before level transition or player death, store **InventoryComp**->**Items** in a *USaveGame* subclass.
   - On *BeginPlay*, load saved inventory, retrieve Items, and loop through *AddItem()* to restore them.

Notes

- Keep the saved game object in memory by storing a reference to it until *LoadGameFromSlot* finishes.

Video 17:

Title: User Interface Work Part 2
Link: Watch on YouTube

The video covered:

- Setting up UI classes for future elements.
- Filling the Inventory UI widgets.
- Connecting UI to the HUD.
- Binding Player Input for UI toggle functionality.
- Organizing code logic.

Explanation

1. Define UI Classes
   - *UInventoryWidget*: Stores *ItemList* (a *UScrollBox*) for inventory items and *EntryClass* for item slots.
   - *UInventoryEntryWidget*: Holds Icon (*UImage*) and *NameText* (*UTextBlock*) for displaying item details.
2. Fill the Inventory UI in C++
   - *RefreshInventory()* clears *ItemList* and adds item entries dynamically.
   - Loops through the player's inventory, creates *UInventoryEntryWidget*, sets its icon and name, and adds it to the ScrollBox.
3. HUD Setup
   - In *BeginPlay()*, create *UInventoryWidget* and connect it to the player's *InventoryComponent*.
   - *ToggleInventoryMenu()* hides or shows the inventory UI when needed.
4. Bind Input for Inventory Toggle
   - In *SetupPlayerInputComponent()*, bind *ToggleInventory* to a key press.
   - When pressed, call *ToggleInventoryMenu()* on the HUD to open or close the inventory.

Notes

- Create the UI widget after the player and component already exist to prevent null errors.
- Don’t forget to turn the cursor on and restrain player movement when the Inventory is displayed.

Video 18:

Title: User Interface Work Part 3
Link: Watch on YouTube

The video covered:

- Finalizing UI components for the Inventory.
- Drag detection system.
- Creation of Drag-Drop operations.
- Handling drops on empty space.
- Spawn of items from drops.

Explanation

1. Drag Start
   When the user clicks and holds on an inventory slot, the slot widget detects the drag and creates a drag-drop payload containing the item's data (its row struct).
2. Drag Operation
   Unreal's built-in drag-and-drop system carries that payload as the cursor moves; you can customize the drag "visual" (e.g., show the item icon).
3. Drop Handling
4. If the user releases outside any valid UI slot (i.e., onto the game viewport), the inventory widget's *OnDrop* handler fires. We then translate the drop screen position into a world location and spawn the item actor back into the scene, removing it from the inventory array.

Notes

- Call the refresh inventory after the item is dropped.
- Use *UWidgetBlueprintLibrary::DetectDragIfPressed* to avoid working with UI geometry.

Video 19:

Title: The HandeStackableItems function
Link: Watch on YouTube

The video covered:

- System for merging of similar items.
- Checker for item limits.
- Updating *AddItem()* with stacking mechanism.
- Testing the systems and debugging.

Explanation

1. Merging Item Stacks: Added *HandleStackableItems()* to *UInventoryComponent*, making new items merge into existing stacks instead of creating extra slots.

2. Checking Stack Limits: Each *FItemRow* has a *MaxStackSize* property. If a stack isn't full, the item is added to it; otherwise, a new stack is created.
3. Updating *AddItem()*: Instead of just adding items, *AddItem()* now calls *HandleStackableItems()*, increasing the count of an existing stack or creating a new one if needed.
4. Testing the System: Tried adding the same item repeatedly—watched the UI group items into stacks up to *MaxStackSize*, then create new stacks when full.

Notes

- *OnInventoryUpdated.Broadcast()* should trigger once per merge to avoid excessive widget rebuilds.

Video 20:

Title: Adding aim functionality with a timeline
Link: Watch on YouTube

The video covered:

- Adding a timeline component.
- Creation and use of a float curve.
- Zoom control with input.

Explanation

- A *TimelineComponent* is like a built-in animator: you give it a float curve (e.g. 90→60 over 0.5 seconds), and it calls your update function each frame with the current curve value. By driving the camera's FOV, you get a smooth zoom transition when the player holds and releases the "Aim" key.

Notes

- Actor needs *bCanEverTick = true* so the timeline would advance each frame.

Video 21:

Title: Project updates
Link: Watch on YouTube

The video covered:

- Upgrading to Unreal Engine 5.4.
- Switching to Enhanced Input.

- Optimizing Item Spawning.

Explanation

- **Engine Upgrade:** Let Unreal's project converter handle the bulk of API changes, then resolve any compile errors (often changes in function signatures or removed functions).
- **Enhanced Input:** Rather than raw key names, you create reusable Input Action assets, assign them in C++, and then bind to them through the enhanced input API—centralizing controls and making rebindings easy.
- **Lookup Caching:** Instead of doing a *DataTable* search every time (which hashes and compares keys), do it once per pickup instance and keep a pointer to the row—so mesh/icon setup is a simple pointer dereference.

Notes

- Ensuring *CachedRowPtr* remains valid (the *DataTable* asset must not be unloaded) or checking for null before dereferencing.

Video 22:

Title: Raycasting in Unreal Engine with C++
Link: Watch on YouTube

The video covered:

- Line trace basics.
- Trace channels and parameters.
- Handling hits and output.

Explanation

- A raycast (or line trace) sends an invisible line through the world and reports what it hits first. By choosing a start and end point and specifying collision rules, you can detect walls, items, NPCs, or any actor in between making raycasts perfect for shooting, interaction targeting, or vision checks.

Notes

- Don't forget to add ignore parameter to the actor from which it is casted.

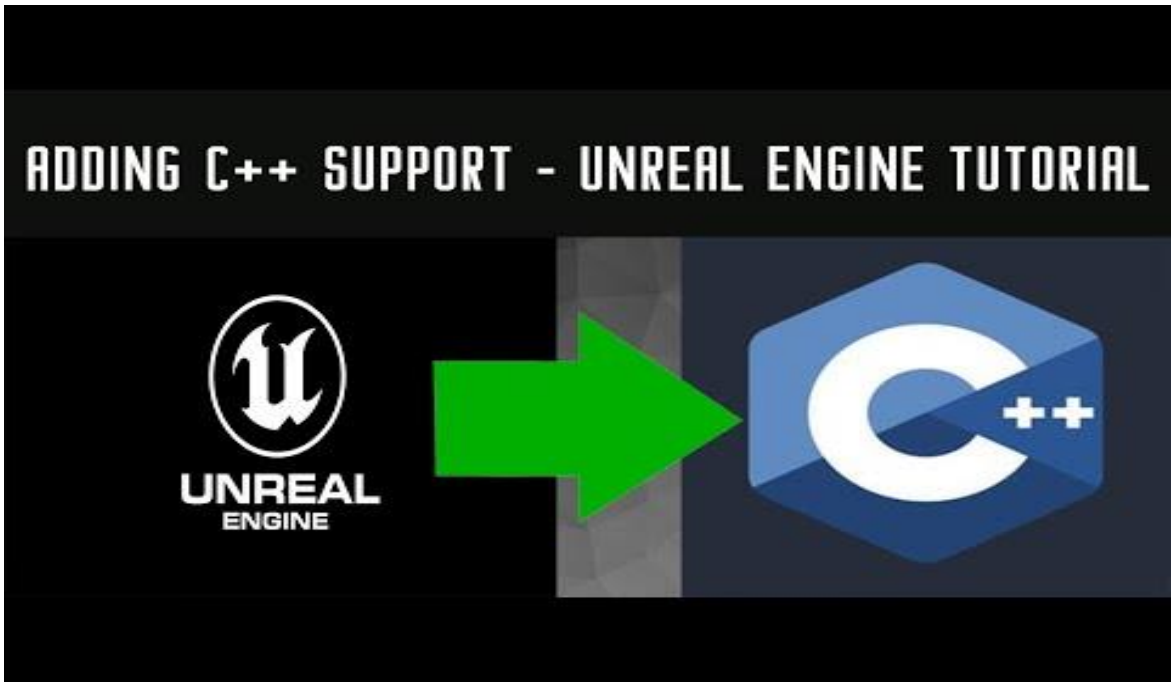
Portfolio Item 2:

My Unreal Engine game modified with C++

Return to the project:

Upon returning back to my Unreal Engine project with game I faced my first problem. When I initially created the project, I chose the preset type based on Blueprints. It was problematic for me, because these types of presets don't display the functions for creation of C++ classes, so I had to put in quite some time into solving this matter. This work mostly consisted of searching the Internet for solutions and analyzing the content of websites and videos. I found the solution to my problem within the following video, so if anybody faces the same problem, they may visit the link and get help here.

Video:
https://www.youtube.com/watch?v=tW1_rD0Geiw&ab_channel=KomodoBitGames

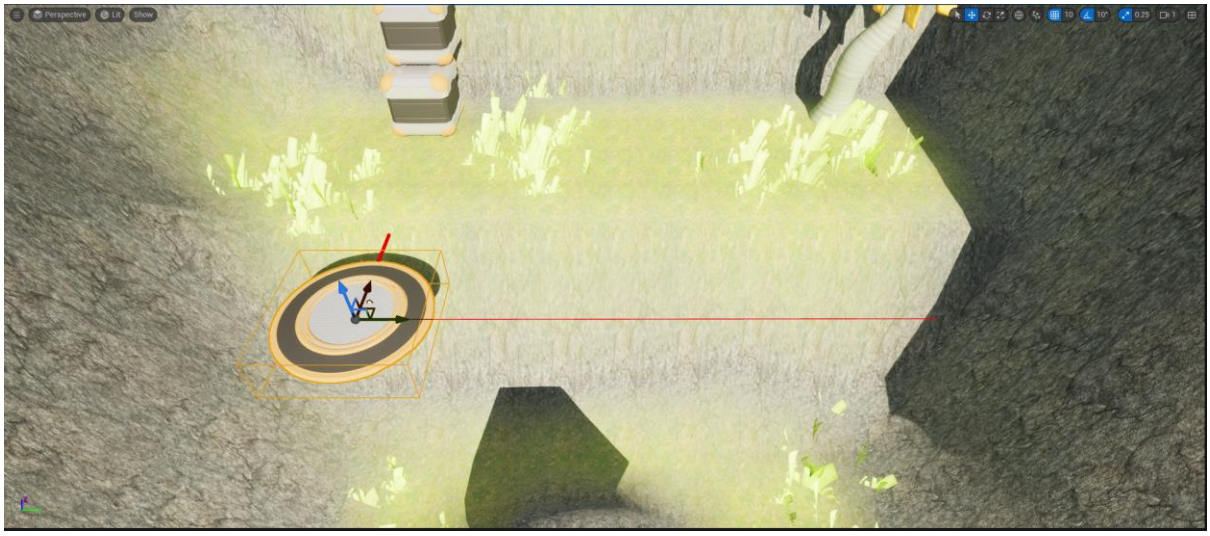


Movin platforms:

After resolving the matter of C++ integration in the project, I started to work on the new mechanics. The first new thing I made for my game was moving platforms, I initially didn't expect to create them, but I got inspired by one of the videos I watched for my journal.

The video link:
https://youtu.be/mWJwAtwkPKQ?si=GFFVz9p8GwfZhBpT

After watching the video, I didn't feel like the work shown in the video would be hard to recreate, but after starting to work on it I faced an unexpected challenge in the least obvious place. At some moment author show how to add a line in the editor to have easier time when planning the level design.

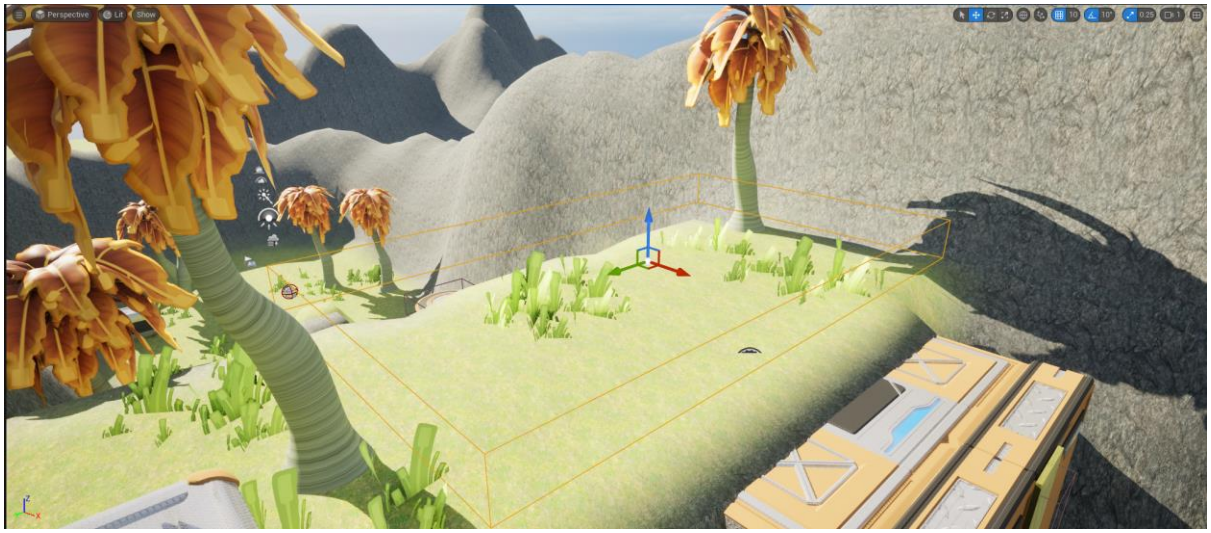


My result in Unreal

Turns out, in order to do something as simple as that, that takes no effort in Unity, in Unreal Engine a person will need to go outside the project in the engine and go edit the project's execution files. Getting the code right, fixing it and setting it up for my project brought me hours of suffering and wasted time for a basic function. This small step became the most difficult one in all of my work. In compression, the rest of the code for the platform was fairly simple, and I made the moving platform between points A and B without any problems.

Random spawn of orbs + building project:

After the moving platforms, I moved on to one of the promised mechanics. I turned to working on the random spawner of actors. Operating principle of the mechanic is simple, we place a cube without mesh and collider and do not apply physics to it, then we calculate the borders of the cube based of its transform, next we take a random location within the borders of the cube and spawn there the selected class.



As an additional functionality the actors can be spawned in different ways, like once at the start of the game as well as using a timer with randomly offset intervals, which allows for a wide range of use with little time consumption.

Sadly, I didn't get away without a challenge on the way. In the middle of working on this mechanic I when I left the project and came back to it the other day, it stopped launching and just threw an unknown to me error, so I started looking for the solution. First when I opened the main script I was greeted by hundreds of errors, at that moment I felt like I wouldn't be able to fix it in the time I had on me and still finish the portfolio, I even started to think whether I should continue in this project or should I create a new one from the ground. But my dedication perfectionism didn't allow me to give up and so I resumed my search for the solution.

After a whole day of searching, I wasn't able to find a working solution, so I decided to try something so obvious that I thought it couldn't have been what I needed all along. I noticed that on launch of Visual Studio I got a notification about an update, so I decided to finally do it, and to my surprise after the update I was able to launch my project without problem, which both brought me happiness and disappointment.

UI + Animation update:

After such strain I decided to do something less complicated. I focused on improving the UI, precisely the sphere counter as at that moment it was just to show how many spheres the player collected from the set by me number. I decided to make two key improvements to it. First, I made it calculate the total amount of the spheres so that that the count would always display the correct maximum number without any need for manual input. Second. I finally made it connected to the finish door, so the player has to collect all the spheres in order to finish the level.

After finishing with the UI, I moved on to the other part that I wasn't able to finish the first time due to the lack of knowledge on the topic. Now with the ability to work with the

C++ I managed to fix the animation logic, but continuing the tradition I faced another problem close to the end on the stage of script implementation. The function of the script was simple, check if the player actor is on the ground and based on that information decide to play a walking animation or falling, but no matter what it kept playing walking animation in the air. So, started to look for the cause of the problem, spent hours looking everywhere and as it always happens to me the problem was right in front of me, I forgot to exclude player actor from collision layer, which resulted in walking in the air.

Inventory:

After so much time spent fixing problems had little time left for the inventory system. So, I had to make a choice: try to recreate the inventory system from the videos and likely not finish it in time, or use the videos as the base and create personal simpler version which can pe problematic as it will challenge all my knowledge and skill with C++ I obtained so far. After much consideration I chose to make my own simpler version of inventory and challenge myself.

While working on this system, it was the only time I didn't have so unexpected problems. The system consists of two component classes that work together: interactable item component which we can put on any actor to pick it up and put in the inventory, and the inventory component which stores the picked actor and on button press releases it back in the world.

And so, I finally finished the items for my personal portfolio.

Conclusion:

1. Coding, learning new scripting language;
2. I think those things went well because I had extensive and vast experience with Unity and C# during my studies, group projects, and personal work. They prepared and taught me to not give up on difficult and new tasks, to search for alternative solutions, and to challenge myself.
3. To maintain this, I have to always search for ways to improve myself and keep up the dedication to my goals.
4. Journal
5. This part of my personal portfolio didn't go so well because I underestimated the amount of information put in those videos and how overwhelming and tiring it could be for a person.
6. To avoid this in future I will concentrate more applying the knowledge and creating something new with it instead of getting as much information as possible.

Writing this documentation, I keep feeling happy from understanding that no matter how hard it was on the way of making these pieces of personal portfolio I have achieved all goals that I have set before myself. I honestly was afraid of working with C++ after watching the tutorials as it looked much more difficult than C# for Unity, but now I feel like understand it and I can use it, not to the fullest of its potential as I have still a long way to go with Unreal Engine, but I can see the way I need to follow to succeed with it.

I am not certain what my next work is going to be about, but to keep the balance in my designer skills I will certainly do something more artistic, as I have a strong understanding of the technical design, so I need to concentrate more on the receivers (player) perception of my design, so maybe something about levels design.