Wilminktheater 2D Point & Click Game

Table of Contents

Name:	**Amber Kortier**
Student Number:	**497337**

Introduction

My name is Amber, I'm an engineer in the project. For this project, we are making a 2D point-and-click game for the Wilminktheater. The goal of the game is to help children aged 5 – 10 learn the rules of the theatre after Bo; the client, came to us with this paint point. A big issue the theatre faces is that children on school trips to the theatre behave poorly and so they sought out a team to make a 2D point-and-click game that teaches kids these rules in a fun and interactive manner.

As an engineer, my responsibilities lie in making tools that the designers can use and features that will make the game function. I tackle this using the double diamond method, where I apply this to (very near) every possible feature and implementation.

Concept phase

We decided to use the Double Diamond Method in our team. This system allows us to explore and iterate more. We plan to use it to find what art style to use and other core gameplay mechanics. During the empathize phase we plan on collecting the needs and goals of our target audience (children aged 5 – 10 going to the Wilminktheater and the theatre itself). I will also use this phase to research about many point-and-click approaches and engines to see what fits with our project.

Empathizing

The empathizing phase was a teamwide process for us, there were also individual parts. That was figuring out what style of game (engine) would fit the best for the target audience, but also what would be viable for us to make within this time period. We want to make a 2D point-and-click game to teach the children the rules of the theatre.

This was predominantly achieved by researching different types of point-and-click engines and games. A request of the team was to not use a custom engine due to the time investments/requirements that come with that. Also, the client requested specifically for the game to be 2D, but I will come back to that.

I started off by researching a lot of different point-and-click engines. This of:

- Construct	(https://www.construct.net/en)
- GDevelop	(https://gdevelop.io/)
- Game Maker	(https://gamemaker.io/en)
- ClickTeamFusion	(https://www.clickteam.com/clickteam-fusion-2-5)
- Visionaire	(https://www.visionaire-studio.net/)
- Adventure Game Studio	(https://www.adventuregamestudio.co.uk/)
- Godot	(https://godotengine.org/)
- RPG Maker	(https://www.rpgmakerweb.com/)
- Renpy	(https://www.renpy.org/)
- Wintermute Engine	(Became unavailable during the project)
- Unity	(https://unity.com/)

All of these engines have their own strengths and weaknesses. Some are really good for specifically point-and-click games and would for sure fit our needs; however, as a team, we had another big obstacle to face. The game needs to run on a website, not as a downloadable executable. This makes almost all these engines obsolete and only Unity stays behind as a viable option.

Because of that, I started looking into different point-and-click game solutions for Unity, there are a couple of very nice ones. For example:

- Adventure Creator	(https://adventurecreator.org/)
- Naninovel	(https://naninovel.com/)
- Game Creator 2	(https://assetstore.unity.com/packages/tools/game-toolkits/game-creator-2-203069?srsltid=AfmBOopOroTct8rpRDzjoiaXnmBArxSsMIxDvDdvgcNhoBLTZnGyYVnc)

These solutions work very well and technically suit our needs, they cost money however and we have a budget of zero euros. We also wanted the ability to playtest and change things up. If we commit to an engine/solution before we do any target audience research it could be a waste of time. The waste of time is the biggest contributor. By getting a custom solution we would waste a lot of time teaching the team about the new solution. Since we don't have the luxury of extended development time this might not be suitable for our needs.

Because we chose a self/custom-made Unity implementation, it was up to me to figure out how we would approach that. The team had concluded that they wanted a character to be able to move around the environment and scale properly depending on the depth. That is easy to do in 3D but difficult to do in 2D due to the ever-changing camera angle. It was also difficult for me to find a proper 2D path-finding solution in Unity that worked well for our game. So ultimately, I decided we build the game in 3D and use the build in Unity 3D Pathfinder because it would fit our needs perfectly.

The problem is, that the client requested a 2D game, so why are we making a 3D game. Well, in reality, a lot of 2D games are secretly built in 3D. I figured it would be the best for us to do this as well as it would make the development of the game easier but we can still make it look 2D by making everything unlit and appear 2D by putting the camera super far away and on a very low FOV.

Before all this research came to fruition the designers made a set of empathy maps that we as a team rely on. They provide us the knowledge of what the end user wants, needs and how they feel. But also what our client wants, needs and feels. Because of this I made very deliberate steps in what engines to research, why they were and were not viable and how we ultimately end(ed) up making a 3D game that would look 2D. (The product owner approved of this in a meeting).

All of these mechanics I researched and build (prototypes, in order to see if something works/is viable you need a prototype of it) are there to ensure that the children get an easy way to learn the rules of the theatre. By making the mechanics intuitive and easy to understand we aim to satisfy all the client's needs.

Problem statement:
In order for this to stay clear in my head I came up with a problem statement: "How do we create a fun and educative game that can teach children from age 5 to 10 the rules of the theatre whilst we keep this project viable to build in the timeframe we have?"

Design Phase

Our plan for this phase is to get a vertical slice of the game so we as a team can get an idea of what we want to make, but also so the client knows and can expect what will be delivered. The idea is that most systems work on a basic level. For this I had to develop multiple solutions based on what the designers have tested and… well, designed. These solutions aim to fix the needs of the product owner, which is a point and click game that helps kids learn the rules of the theatre.

However, in order to create a game; and a final end product; you have to build global systems as well. The global systems are my responsibility. The catch with them however is they are less for the client and more for the team. The reason for this is because the tools I develop are used by the team to ultimately create a game based on the clients wishes. The clients wishes gets relayed to the team; by extensions to me as well; and we think out what the game should be capable of doing. Based on that I set out to create solutions and tools that can aid that development. Then afterwards, based on feedback during the testing phase tools get changed, documented, removed and/or added.

On of the biggest challenges was accounting for the teams wishes. The design phase for me started relatively slow because of this. I wasn't per-se designing for the client so much as I was designing for the team so they could build what the client desired. The team however didn't fully know what they expected yet either because they were still designing it so I mostly spend my time building global systems. This was of course not a waste of time, as I will explain in later systems this is simply something that *has* to be done for a feature to work, no matter how. E.g. it doesn't matter what kind of button you design if inputs aren't properly handled yet.

Production and Testing Phase

This was the hardest phase for me because it involved a lot of work. The designers came up with a lot of different concepts that fit the research question to make sure the game that teaches kids aged 5 to 10 in a fun and engaging way the rules of het Wilminktheater. This process involved a lot of proper planning on my part and pointing out stuff that would be difficult to program beforehand so we could make a proper planning and meet expectations. This phase was also very team intensive where we had lots of meetings and planning sessions.

I want to go over my working process for some of the features. There was a lot that I had already made/implemented during the design phase as well. This was because as an engineer you need to know how viable the most difficult parts of a design is to bring to fruition and relay that information to the team. Based on the tests we did during the design phase we could now implement fully functioning code.

This phase is where my involvement in the steam skyrocketed compared to before as well. Most the systems I build in this phase were in conjunction with what the designers or artists have made, and so it was very collaborative. This phase is also where most bugs came to light and had to be fixed.

Interaction system

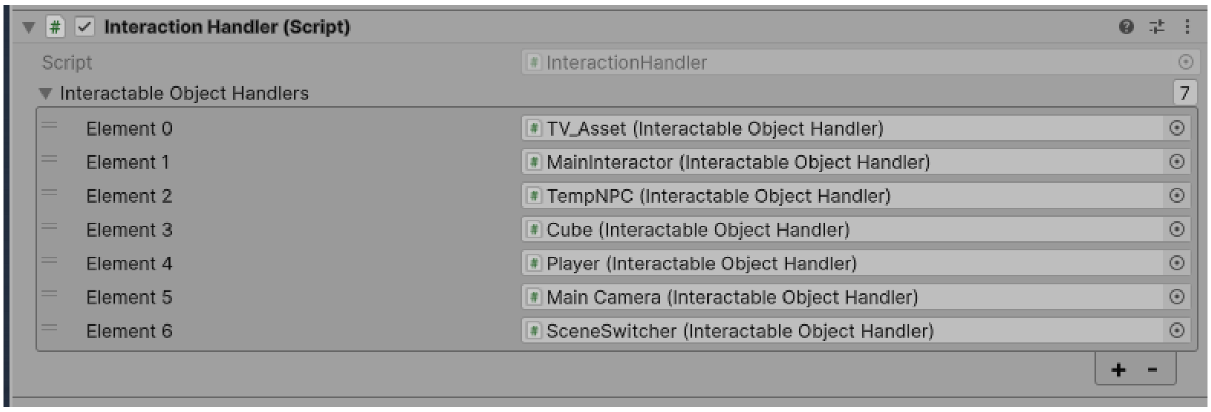
Our interaction system is a core component to the game and is just like the queue system not something planned out by the designers but a system I knew we needed for the game to work. The interaction system looks for interactables in the scene. They have a priority number, the higher the priority the more it favours this interaction. This is because some interactions should be handled with a higher priority than others. Think for example about a button to click in the game. That should have a higher priority than walking because otherwise nothing would be clickable and you would walk everywhere.

Over the whole project I have added many different interaction types that the designers can use. These are usually buttons and walk to interactables that can be used for all kinds of systems. Walk to interactables are interactables that queue a walking action to their location and then execute their code. They all work using Unity Events and are very simple to use. The designers would often ask, can we do this (a certain action) and I could then simply tell them, no, but I will add it or, yes, use this element and due to their design the designers understood how to use them.

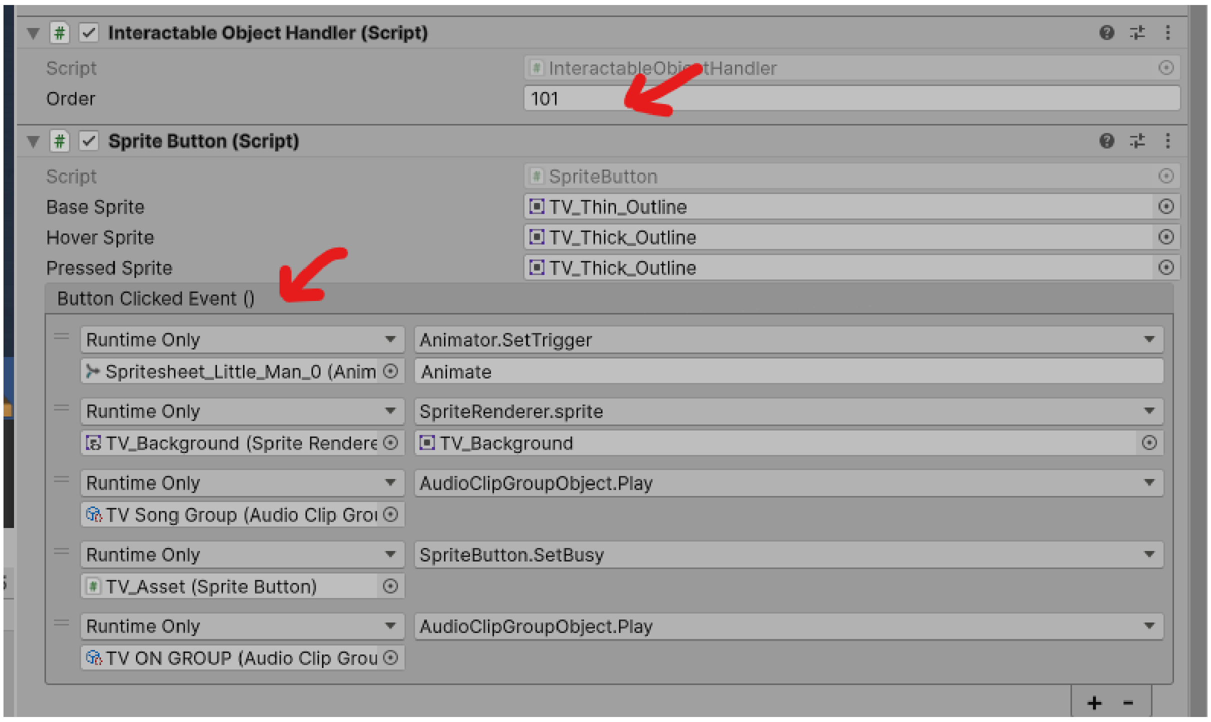
Just like the queue system this was also not made using the diamond model. This is because it isn’t a designed system that can be iterated upon. It is an ever evolving system that just exists within the game to make it work. I always describe it as a law of existence. If you want a system to function you must have some sort of system that handles for those systems to be allowed to function. Buttons are a great example of this, for buttons to exist you need to handle mouse inputs, how else would you know you clicked a button. This is why it’s not an iterative process but an ever evolving process. You cannot change it, it *has* to exist, you just iterate and build upon how you use that system.

The priority system and the different elements that I build around the interaction system however are part of the diamond model. This is because based on feedback from designers and (bug) testing we came across functionality that we would like to see improved.

Interactable objects are made by request of the designers. If we wanted a button to show an outline, or fade in a certain way, I made that and we implemented it together. Based on their feedback, we would often change how the interactions worked or make new interactable entirely.


Interaction Handler (Script)

Script	InteractionHandler
Interactable Object Handlers	7
Element 0	TV_Asset (Interactable Object Handler)
Element 1	MainInteractor (Interactable Object Handler)
Element 2	TempNPC (Interactable Object Handler)
Element 3	Cube (Interactable Object Handler)
Element 4	Player (Interactable Object Handler)
Element 5	Main Camera (Interactable Object Handler)
Element 6	SceneSwitcher (Interactable Object Handler)


Interactable Object Handler (Script)

Script	InteractableObjectHandler
Order	101

Sprite Button (Script)

Script	SpriteButton
Base Sprite	TV_Thin_Outline
Hover Sprite	TV_Thick_Outline
Pressed Sprite	TV_Thick_Outline

Button Clicked Event ()

Runtime Only	Animator.SetTrigger
Spritesheet_Little_Man_0 (Anim	Animate
Runtime Only	SpriteRenderer.sprite
TV_Background (Sprite Rendere	TV_Background
Runtime Only	AudioClipGroupObject.Play
TV Song Group (Audio Clip Gro	
Runtime Only	SpriteButton.SetBusy
TV_Asset (Sprite Button)	
Runtime Only	AudioClipGroupObject.Play
TV ON GROUP (Audio Clip Grou	

Queue system

The queue system was not a system that the designers had planned for but I knew that we would need for our project to work the way we wanted it to work. There had already been a lot of talk about walking to a destination, which would in turn play a sound or animation and afterwards an action gets executed. There were also other interactions like clicking a TV which then plays an animation. All of this boils down to, a simple list of elements that should execute in order. Because of this I decided we would have to be able to queue certain interactions and execute them.

My initial design is also the final design. There will be a queue handler that simply tells each object in the queue to execute after each other. Each object will then execute code by itself and each different system will automatically create a queue element of itself when you tell it to play its given action.

The queue system is quite simple in execution. Every implementation will become a queue object. A queue object has a start, end, and error callback; there is also overhead to cancel a queue element early. This system is very easily implemented to stuff we already have in our game. Pathfinding can simply become a queue object where it says it ended when the character reached the end of the path. Same goes for dialogue or audio. Every element can decide itself how it wants to become a queue object, due to this it is very easy to build a queue of tasks, even for designers.

They can drag and drop an element into the queue, and depending on what order they put it in, that's the order the queue will handle the elements. The queue is a system no one has to deal with directly. A queue element will insert itself into the queue and then the queue will tell that object to start doing stuff. This way the designers don't even need to think about timing or working around edge cases, the objects do that themselves. This system is used for every interaction in the game whilst being completely invisible to the designers.

Because of this I didn't build this system around the diamond model. The queue system would simple add every new system to its catalogue by forced design and I would fix bugs where they showed up. It didn't need to iterate because it was an ever evolving system. The queue system is so simple that there also isn't really anything to iterate on regardless. You provide it a list of actions to perform in order, it performs the actions.

Pathfinding

Very early on our designers had decided that we want our character to be able to walk and interact with stuff. They wanted a forced perspective where the character scales with distance and walks slower, my solution to this was to build the scene in a 2.5 perspective. We can ensure that the game looks and eyes 2D but we build it in 3D which is a lot easier to work with. This allowed me to also use simpler path finding solutions like A*.

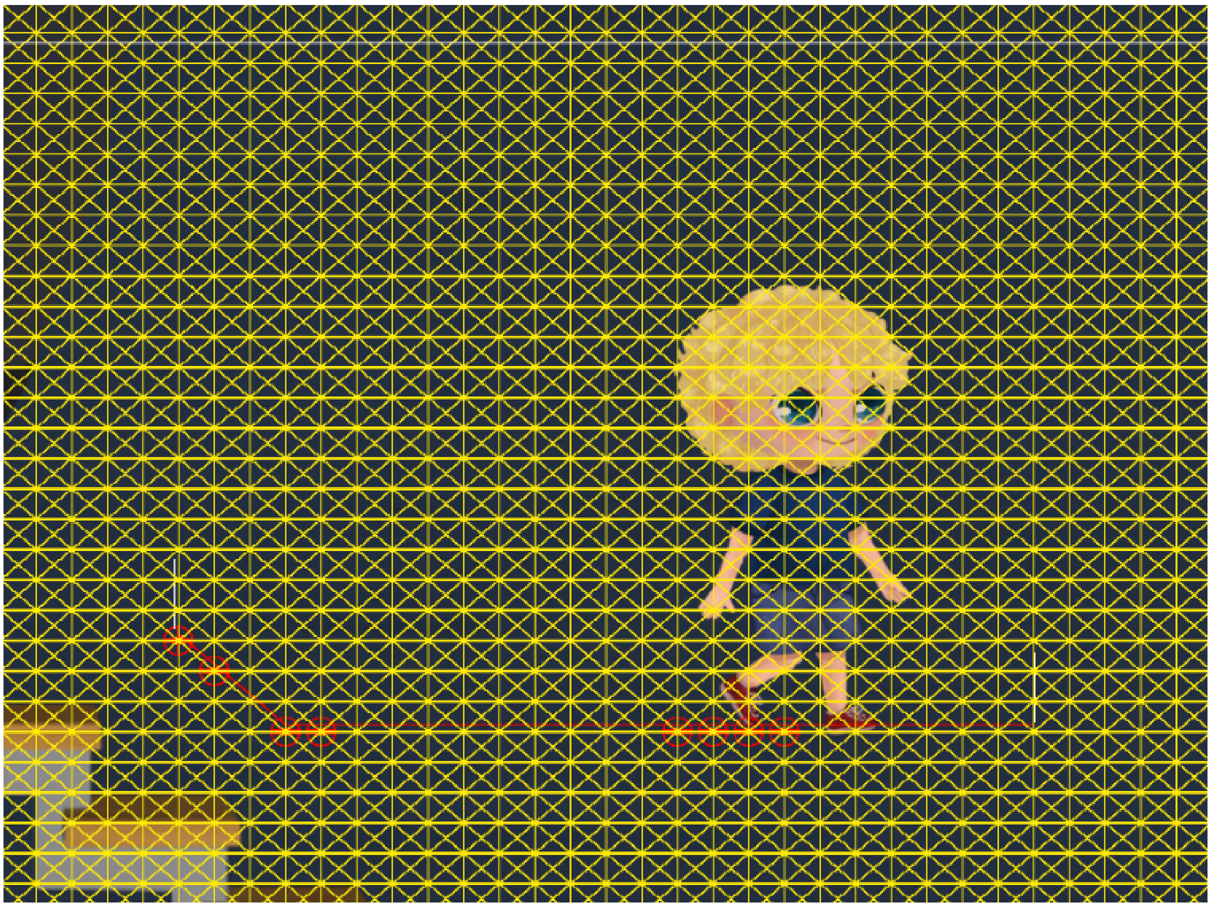
Initially I imagined to use Unity's mesh finding system, but it came with a lot of quirks that would ultimately make it not fit our needs. I chose the Mesh path finding system initially because it is known to be quite easy to work with and it is used very often for simple path finding solutions. Our path finding didn't need to do anything complex, it really involved us to just walk from point A to point B. The mesh finding system really prefers normal terrain and not 2.5D terrain however. It caused clipping and unwanted bumps in locations that I seemingly couldn't fix and it wasn't very customizable like I initially thought it was. It was also hard to talk to due to the lack of events I could hook in to (think of start, end, error, success). It would also require us to build the scene in actual 3D which is not what we were looking for. More on that later, because eventually it turned out the ideal solution was to actually build the scenes in 3D.

Because of this I decided to use a simple A* solution. The reason for this wasn't perse based on research I did during the project but due to experience and prior research. It's a system I have worked with a lot, which is simple/easy to implement but it would also fit our needs. The implementation I went for has a couple more customization options that allows you to specify how far from the edge you prefer a character to walk. This is so we prevent a lot of clipping. I also added smoothing to the path so it walks around corners in a curve rather than straight. This was done due to feedback that the character looked very choppy when walking. I also fixed this by making the system prefer less corners and more straight lines to walk in.

The system itself works by building a grid on top of the floor. This grid then builds connections from one node to the other. The path finder will pre-bake every hard to calculate value like distance and cost to move to this node. This speeds up the pathfinding process significantly and allows the generated path to be made rather quickly. Pathfinding also got implemented in the queue system and will throw errors at the right times to ensure that moving to a point never feels weird and gets interrupted properly.

We also decided to make the system 2D-based initially, but after we switched to 2.5D and I implemented A* our designer came up with a good solution to use ProBuilder; a Unity package; to build our floors. This would also allow us to properly place layers and layering at the right depth. It is very easy to create a 3D mesh with it so our floor can easily follow all kinds of shapes. Because our floor is also created slanted for this reason I made the path-finding system work slanted as well, basically it will group vertical nodes closer together compared to horizontal nodes whilst keeping the cost the same (this ensures that it thinks its on a flat plane when in reality it is squished).

The system initially was very buggy. But after sitting down with the designers who used this system constantly, we were able to quickly solve certain issues and fix bugs. Not only that, this is also why I made walking smoother and made it prefer to stay away from the edges. I'm talking about pathfinding as if it was a one-time and done implementation but in reality this has been constantly evolving from day one right up until the final day of the project. We are currently very happy with how it works, and I'm personally very pleased by the implementation and how well it is usable. It is not only linked to the player either, any entity that has a path-finding agent attached can use it.



The variable system

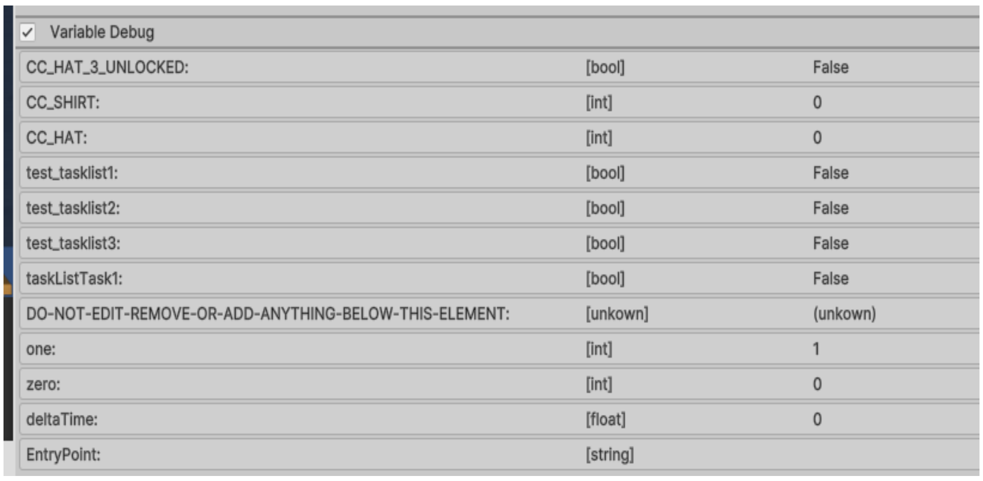
Writing code is not a real solution for our designers and artists. They are not trained, and I am afraid they will not following coding standards and break the game in ways that I was unable to foresee. Because of this I looked for quick and simple solutions. I heavily considered Unity's visual scripting pipeline, but that had multiple downsides:
It was not available for our version.
It was, quite frankly, overkill for what we would use it for.
I would still not alleviate all issues because designers would still have to program manually.

Instead I set out to find a package that could simply store different variable types for me between scenes and ideally use Unity Events to call and set those values. I found a package that supports that, which is Unity's translation system. This variable system is meant to store values so you can directly use them within translations but you can very easily repurpose it. It did not however satisfy my requirement of using them via Unity Events but I could easily remedy that by building that myself. The real challenge was storing variables of different types anyways.

This package solves different problems I want to solve. The designers made a couple ideas and requests of what they wanted to be able to do. Since they don't know how to program or implement stuff it was up to me to decide what the ideal solution would be that would take the least amount of time to implement and is still robust enough to be useful. I ultimately decided on the Unity Event system because of this, since this is a universal standard in Unity itself I figured it would make it the easiest/most intuitive to work with.

The code that makes the system work eyes complicated due to the massive amount of inheritance that I used. This is because I made the whole system work on a small amount of base scripts that I could mix and match to make my solution quickly work. The downside of this is that it is hard to understand for another engineer, even me if I didn't document it properly. The upside is that I didn't spend multiple weeks writing the same code over and over. The system deep down all runs on the same lines of code making bugs less likely to happen and changes a lot easier to implement as well.

In the production and testing phase the designers have now used the system to implement multiple variables of their own and using the quick if-statement system to determine whether certain events should be active or not. What this has shown is that the system is still hard to use, which is good feedback for projects I work on in the future. I deemed it not necessary to fix this for our project due to the time investment, not to mention that after some coaching they do understand it now.

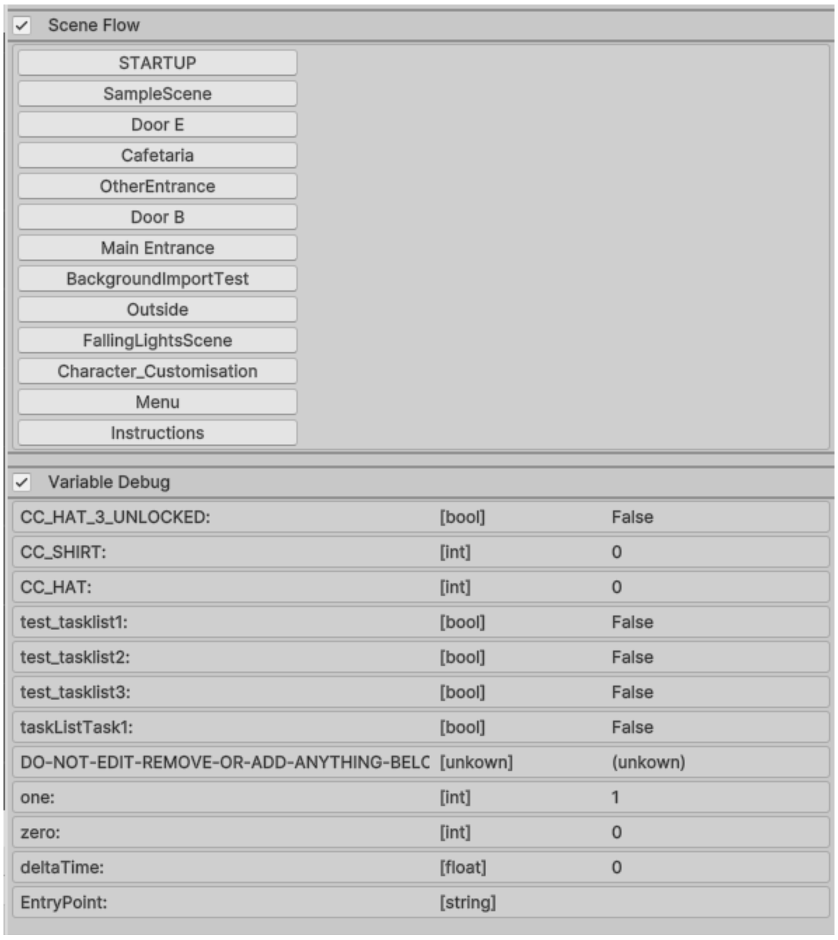


☑ Variable Debug

CC_HAT_3_UNLOCKED:	[bool]	False
CC_SHIRT:	[int]	0
CC_HAT:	[int]	0
test_tasklist1:	[bool]	False
test_tasklist2:	[bool]	False
test_tasklist3:	[bool]	False
taskListTask1:	[bool]	False
DO-NOT-EDIT-REMOVE-OR-ADD-ANYTHING-BELOW-THIS-ELEMENT:	[unkown]	(unkown)
one:	[int]	1
zero:	[int]	0
deltaTime:	[float]	0
EntryPoint:	[string]	

Editor Tools

The first thing I made during this phase was basic tools for the designers and artists to make the game quicker. This was done in the way of Unity editors and scriptable objects. This editor allows everyone to quickly navigate the game and see the values of variables in the list. This is a tool I, and everyone in the team uses a lot and we actually went through a phase of trying to upgrade it in the form of grouping. Sadly, due to the way Unity handles scenes, this was not feasible. But the editor was an evolving process that got iterated on and is well used. It used to also be able to either delete or create scenes. Later on in the process we upgraded to Unity 6 and deemed this not a high enough priority to fix.



✓ Scene Flow

- STARTUP
- SampleScene
- Door E
- Cafetaria
- OtherEntrance
- Door B
- Main Entrance
- BackgroundImportTest
- Outside
- FallingLightsScene
- Character_Customisation
- Menu
- Instructions

✓ Variable Debug

CC_HAT_3_UNLOCKED:	[bool]	False
CC_SHIRT:	[int]	0
CC_HAT:	[int]	0
test_tasklist1:	[bool]	False
test_tasklist2:	[bool]	False
test_tasklist3:	[bool]	False
taskListTask1:	[bool]	False
DO-NOT-EDIT-REMOVE-OR-ADD-ANYTHING-BELC	[unkown]	(unkown)
one:	[int]	1
zero:	[int]	0
deltaTime:	[float]	0
EntryPoint:	[string]	

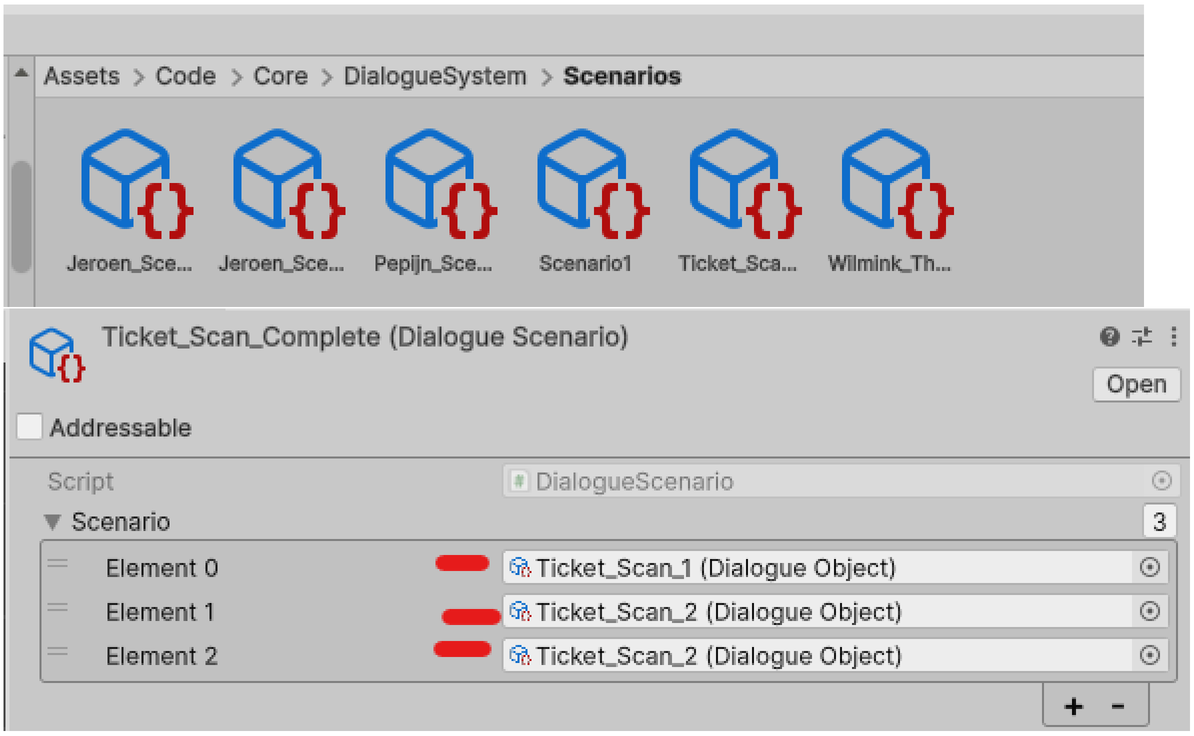
I like this editor and the other editor tools I made because it really helped our workflow as a team, but also made the project feel more welcoming to work on. Joining a project late or getting taught how something works, well... works, but often brings with it a slow working process. By having designed a game the designers know exactly what they want and so I was quickly set out to make those tools.

There are however restrictions, if designers don't like using scriptable objects there's sadly not a lot I can do about it, as the alternatives are either, write code, use the inspector (which as an engineer I know is not at all a feasible solution due to the way objects are handled) or still use scriptable objects. Of course, the fourth option, creating our own system was also considered but quickly shut down after some talking. The downsides way overshadow the upsides of the provided freedom. Mainly due to the enormous time investment it would take.

Scriptable objects

The whole game also runs on scriptable objects. The beauty of this is that designers never have to look at code in order to make something happen. I made sure to use them as much as possible for almost every situation. Think of dialogue, sounds, transitions, the Task List, movement and a lot more. I also tried to keep the design between scriptable objects very similar. This is something I had free control over. I could make every system as small or expansive as I wanted, but by keeping a uniform look and workflow designers were much quicker to pick up any new systems.

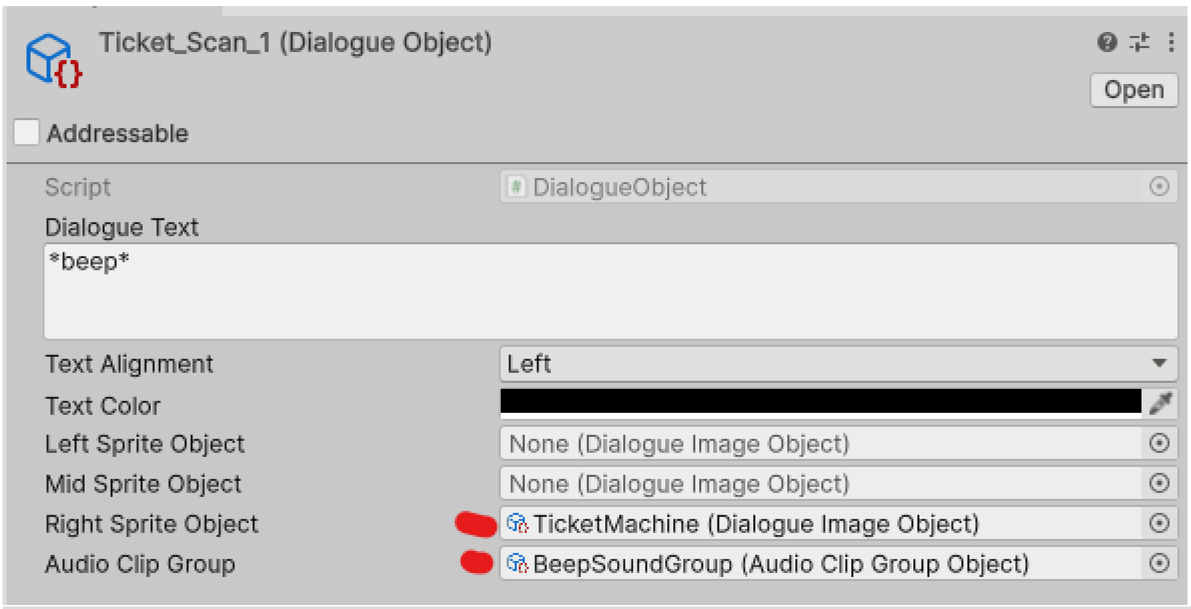
The base system consists of groups/holders/scenarios that hold a lot of smaller objects, and a group of smaller objects that you can create manually. The dialogue system consists of 3 systems and is a bit of an outlier in that sense. It has a scenario; this holds a group of text and each separate text elements holds a line of dialogue and also another group comprised of the speakers image and audio. I wanted to point this out because it still follows my core principle of Main Element and Groups of Smaller Elements. In this case the smaller elements also hold other Small Elements but also potential Main Elements when it comes to audio.



Ticket_Scan_Complete (Dialogue Scenario)

Addressable

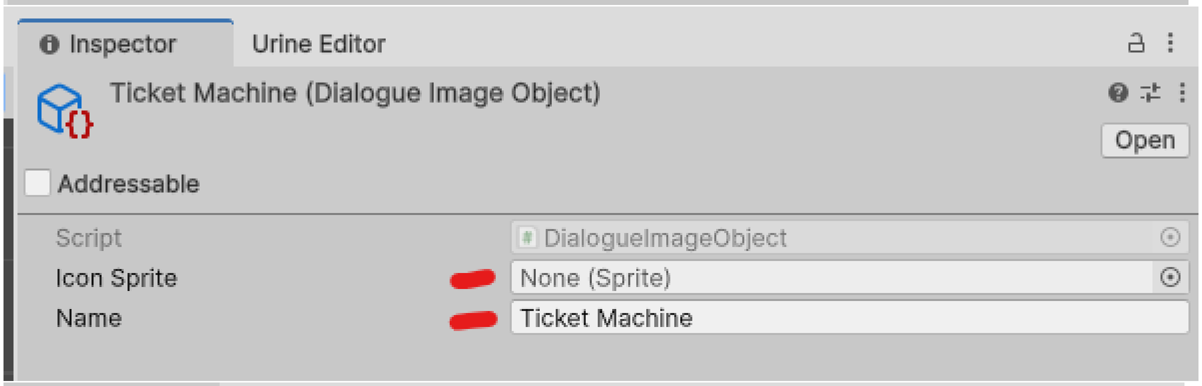
Script	DialogueScenario
Scenario	3
Element 0	Ticket_Scan_1 (Dialogue Object)
Element 1	Ticket_Scan_2 (Dialogue Object)
Element 2	Ticket_Scan_2 (Dialogue Object)



Ticket_Scan_1 (Dialogue Object)

Addressable

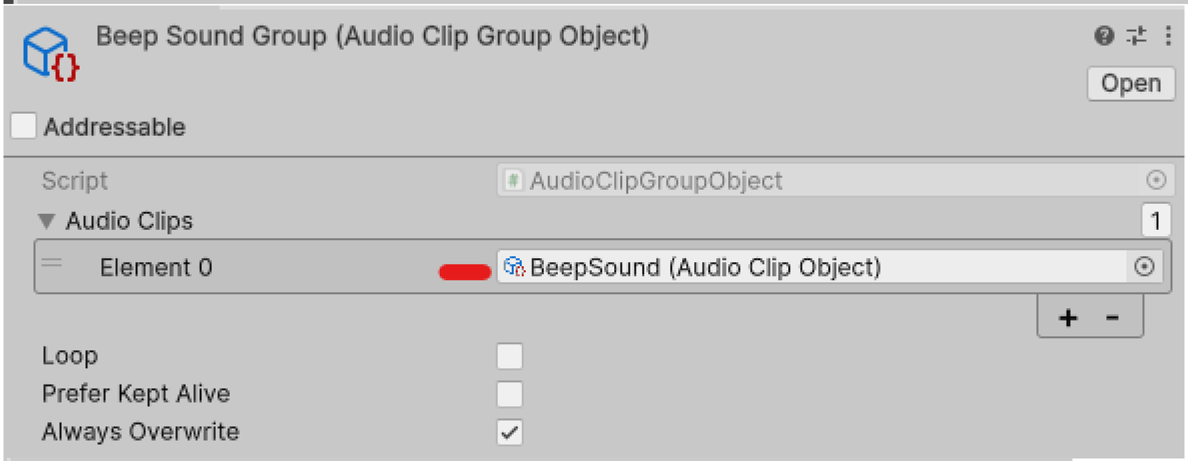
Script	DialogueObject
Dialogue Text	*beep*
Text Alignment	Left
Text Color	■
Left Sprite Object	None (Dialogue Image Object)
Mid Sprite Object	None (Dialogue Image Object)
Right Sprite Object	TicketMachine (Dialogue Image Object)
Audio Clip Group	BeepSoundGroup (Audio Clip Group Object)



Ticket Machine (Dialogue Image Object)

Addressable

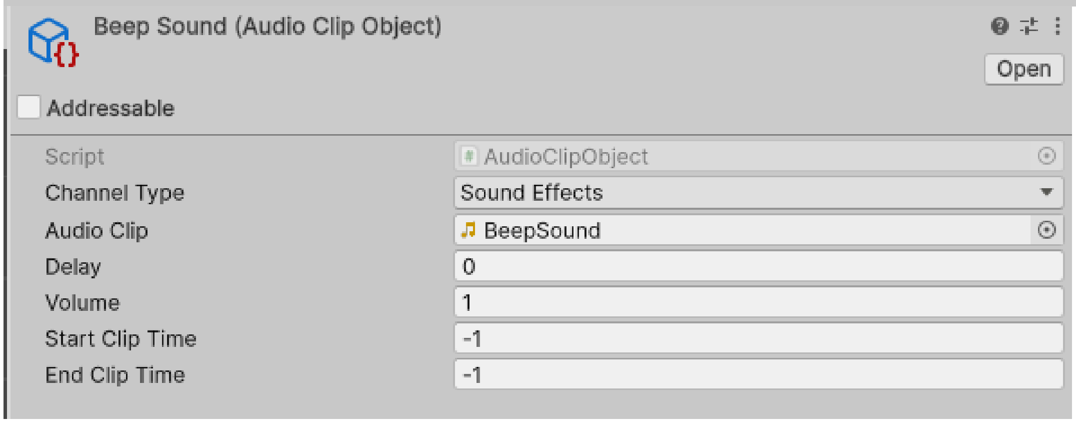
Script	DialogueImageObject
Icon Sprite	None (Sprite)
Name	Ticket Machine



Beep Sound Group (Audio Clip Group Object)

Addressable

Script	AudioClipGroupObject
Audio Clips	1
Element 0	BeepSound (Audio Clip Object)
Loop	☐
Prefer Kept Alive	☐
Always Overwrite	✓



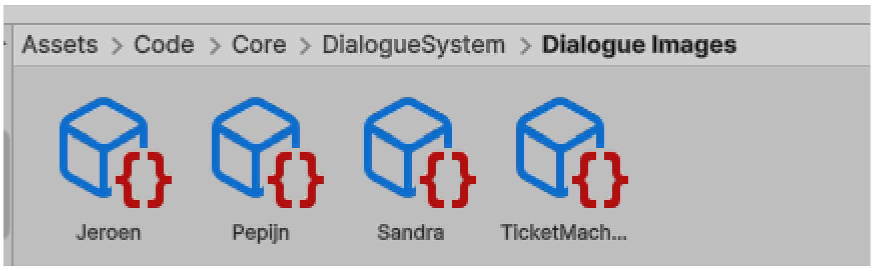
Beep Sound (Audio Clip Object)

Addressable

Script	AudioClipObject
Channel Type	Sound Effects
Audio Clip	BeepSound
Delay	0
Volume	1
Start Clip Time	-1
End Clip Time	-1

This is a process I personally decided on. Designers know what they want and in what ways they want customization to work. I would implement that and later we iterate over that process. It also eyes quite confusing, if you look at the images there are sub systems in sub systems in sub systems. This is why I personally decided to keep systems uniform, but not only uniform, they also all function on the same core principle. Because I iterated on this process over time it became a lot easier for designers to work with it as well.

The beauty of making objects like this is more than just ease of customization, it also creates a workflow where we do not have to make the same resources over and over again. Instead of manually making a scenario that holds dialogue that might be the same as elsewhere with the same image as elsewhere that might hold the same audio as well we make 1 instance of everything and then mix and max. This not only saves on storage space and allows for more creative freedom, it is also a massive increase in performance. Not that Unity would noticeably lag ever if I used the other system. It still is good practise to not waste resources in the first place, especially if later on the game starts to perform worse after growing larger. Having this pre-optimized allows you to focus your time elsewhere.



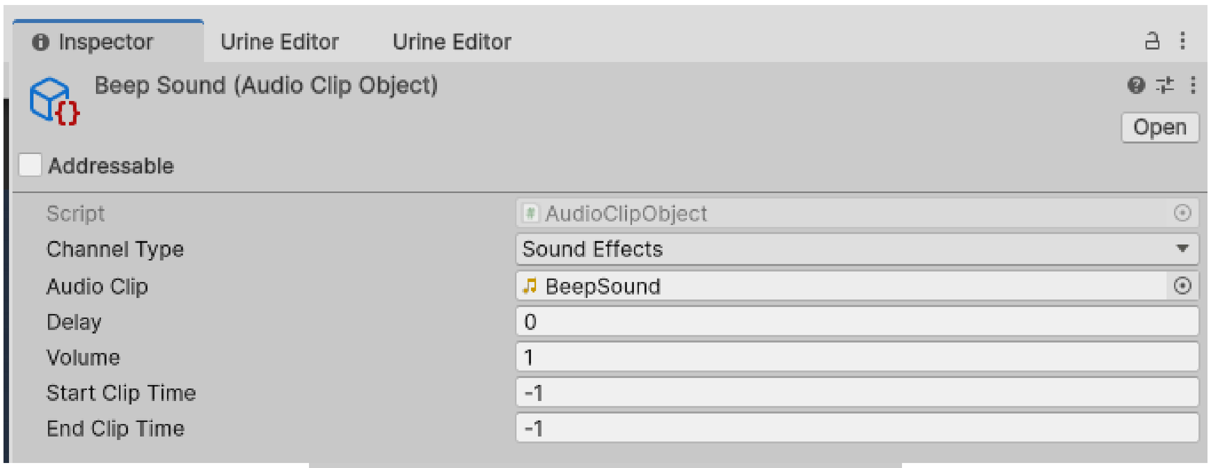
(We only have 1 asset per character that stores data once, instead of once PER dialogue object.)

This process also greatly improved over the weeks as designers used it more. Initially we didn't have audio with the dialogue for example. Now we have full support for it due to the system being so expandable. By investing more time initially when developing the system we saved time later by trying to broaden the concept.

Audio System:

Because I wanted audio to be made using Scriptable Objects as well I made a system that allows for that to be possible. Placing audio in the scene is nice, but I'd rather the designers use the previously build systems instead of working around it. This also allows for assets to be reused. By making a set amount of audio channels and clip players attached to the camera I can ensure that audio always plays in the correct spot, can be called from anywhere and doesn't hard rely on a reference in a scene.

This is useful for cases where you want to for example group audio with dialogue. This way you don't have to make sure the audio for a dialogue clip is located arbitrarily in some random scene but you can keep it all located in the game data itself, where it belongs. The use of Scriptable Objects also made it so the audio system could follow the same workflow as the rest of the game. Even if it didn't have the other benefits I would've still converted it to the Scriptable Object system for this reason alone.

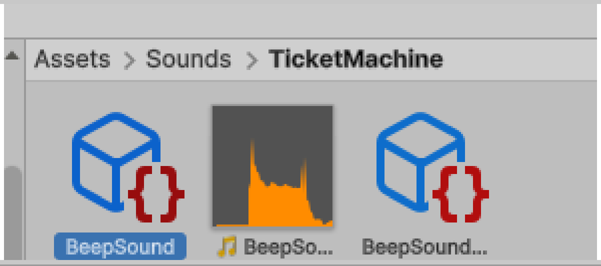

Inspector | Urine Editor | Urine Editor

Beep Sound (Audio Clip Object)

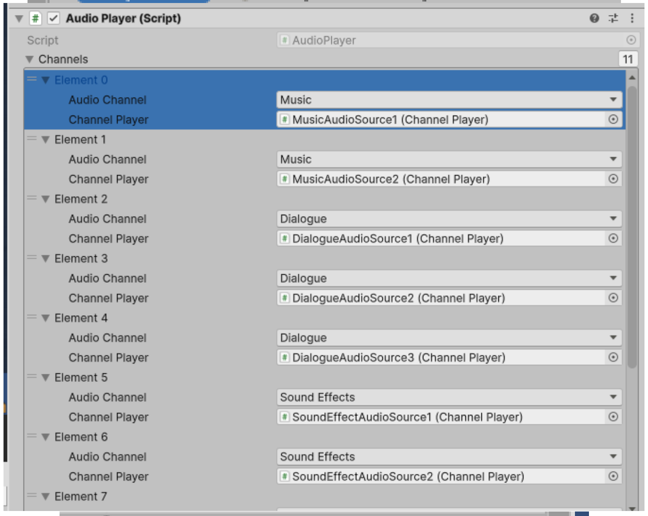
Open

Addressable

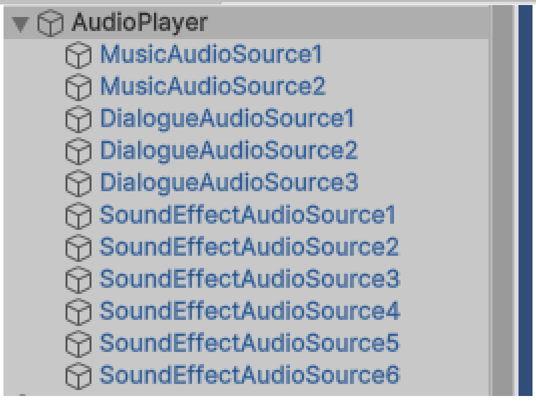
Script	AudioClipObject
Channel Type	Sound Effects
Audio Clip	BeepSound
Delay	0
Volume	1
Start Clip Time	-1
End Clip Time	-1


Assets > Sounds > TicketMachine

BeepSound | BeepSo... | BeepSound...

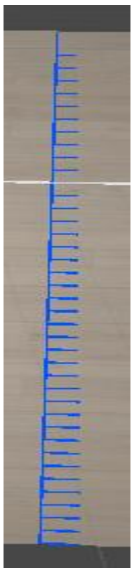

Audio Player (Script)

Script	AudioPlayer
Channels	11
Element 0	
Audio Channel	Music
Channel Player	MusicAudioSource1 (Channel Player)
Element 1	
Audio Channel	Music
Channel Player	MusicAudioSource2 (Channel Player)
Element 2	
Audio Channel	Dialogue
Channel Player	DialogueAudioSource1 (Channel Player)
Element 3	
Audio Channel	Dialogue
Channel Player	DialogueAudioSource2 (Channel Player)
Element 4	
Audio Channel	Dialogue
Channel Player	DialogueAudioSource3 (Channel Player)
Element 5	
Audio Channel	Sound Effects
Channel Player	SoundEffectAudioSource1 (Channel Player)
Element 6	
Audio Channel	Sound Effects
Channel Player	SoundEffectAudioSource2 (Channel Player)
Element 7	


AudioPlayer
- MusicAudioSource1
- MusicAudioSource2
- DialogueAudioSource1
- DialogueAudioSource2
- DialogueAudioSource3
- SoundEffectAudioSource1
- SoundEffectAudioSource2
- SoundEffectAudioSource3
- SoundEffectAudioSource4
- SoundEffectAudioSource5
- SoundEffectAudioSource6

Scaling Systems

Eventually we changed how we wanted to have players move slower and scale smaller the further away they were. This is where the scaling system I made comes in. This system allows designers to draw a line from point a to point b, and depending on how far along the line they are they can set their own scaling values. This works for Players and NPC's (had we decided they could move later on). This was a process I was mostly involving Alexis in as this was a direct request from him. This system is built around his wishes and thus works super intuitively.



You can set two points in the editor and it'll draw a scaling indicator along it. This scaler can be applied to multiple systems by choice which will then handle themselves. The systems you can apply them to are also very easily implementable onto other systems. Because from experience I knew that the designers would ask if we could apply this onto other systems as well. I made sure the answer was yes rather than no.

It is still important as an engineer to weigh the costs though. Sometimes making a system more expandable really is a waste of time, especially if the system is not that big to begin with. Think of a onetime visual effect in the game. It's not that important to build a whole system around it, because even if more of them should be made; unless you suddenly decide to want many; the freedom is more important for visual effects. But a scaling system is a prime example of a system you should invest more time into, because that way you can reliably make it based on any entity in your game without the entity being aware. This is the ideal way of programming as it allows for linear scalability rather than exponential. (Yandere.cs from Yandere Simulator proves my point)

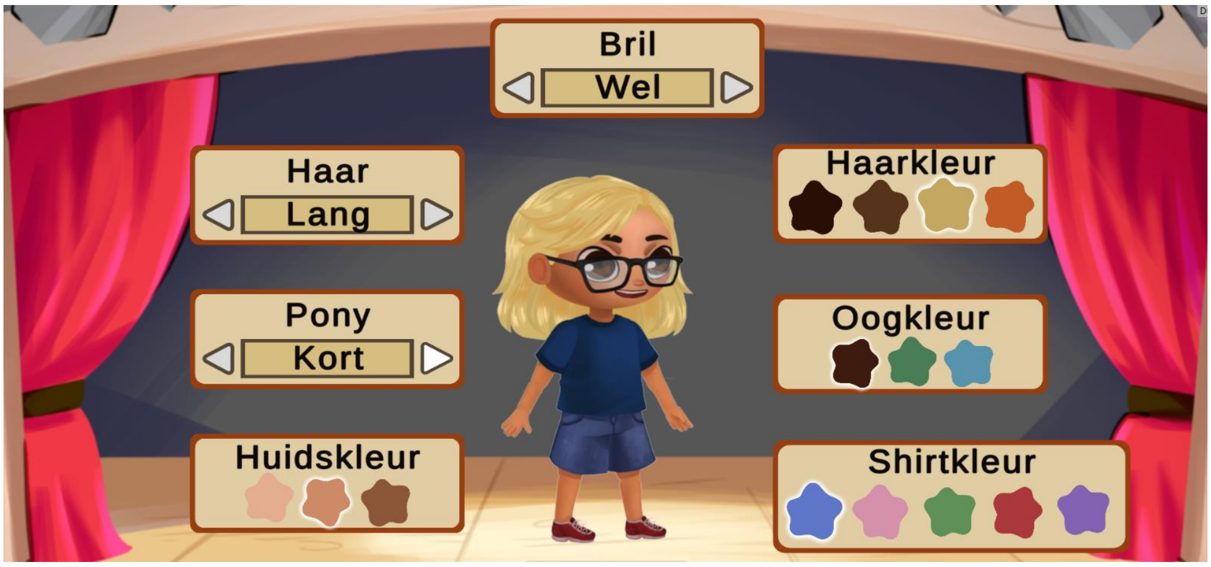
Spine 2D

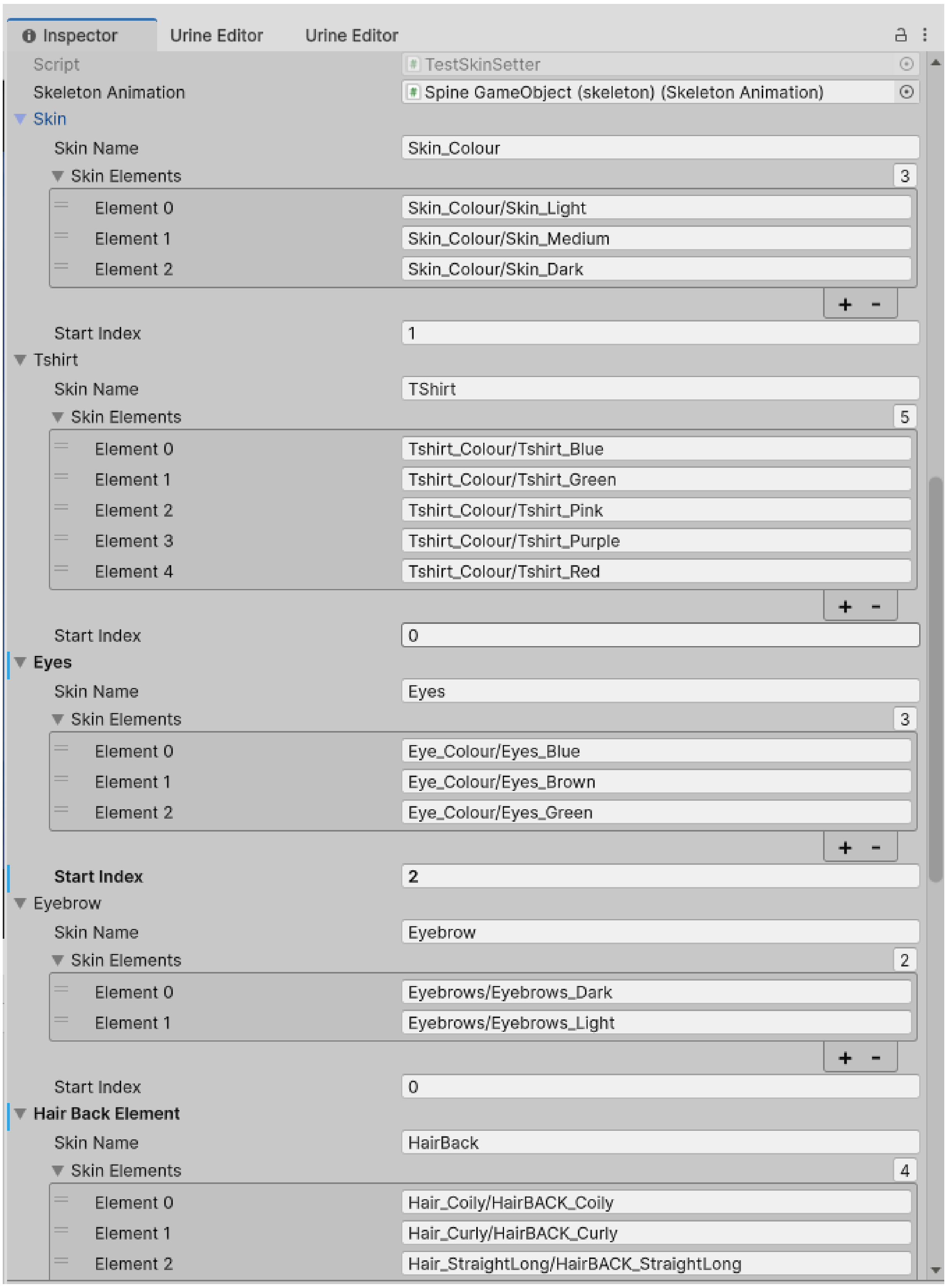
Spine 2D was a massive process involving one of our artists where we put a character with all its animations and customizations in the game. I was lacking a significant amount of knowledge when it came to Spine 2D, especially since they break every Unity standard as well, so I had talks with the artists about how to go about this. Spine 2D is used for our main character and ideally also the NPC's so our system had to be robust. I have tinkered a lot with different Spine 2D implementations and how to achieve this best and thought it was impossible for it to meet our standards. The standards being: Customization, Animation and Colouring.

Turns out, it was actually rather easy, there is a document outlining how to do all of it which I accidentally didn't stumble across during multiple research sessions. If I were to reflect on my research sessions, I wouldn't even say I approached it poorly, the reason I overlooked it so often is because they used internal names to explain things. It is hard to find something you don't know. Now that I know Spine 2D likes using this approach I would definitely tackle future searches differently, this was unknown to me at the time.

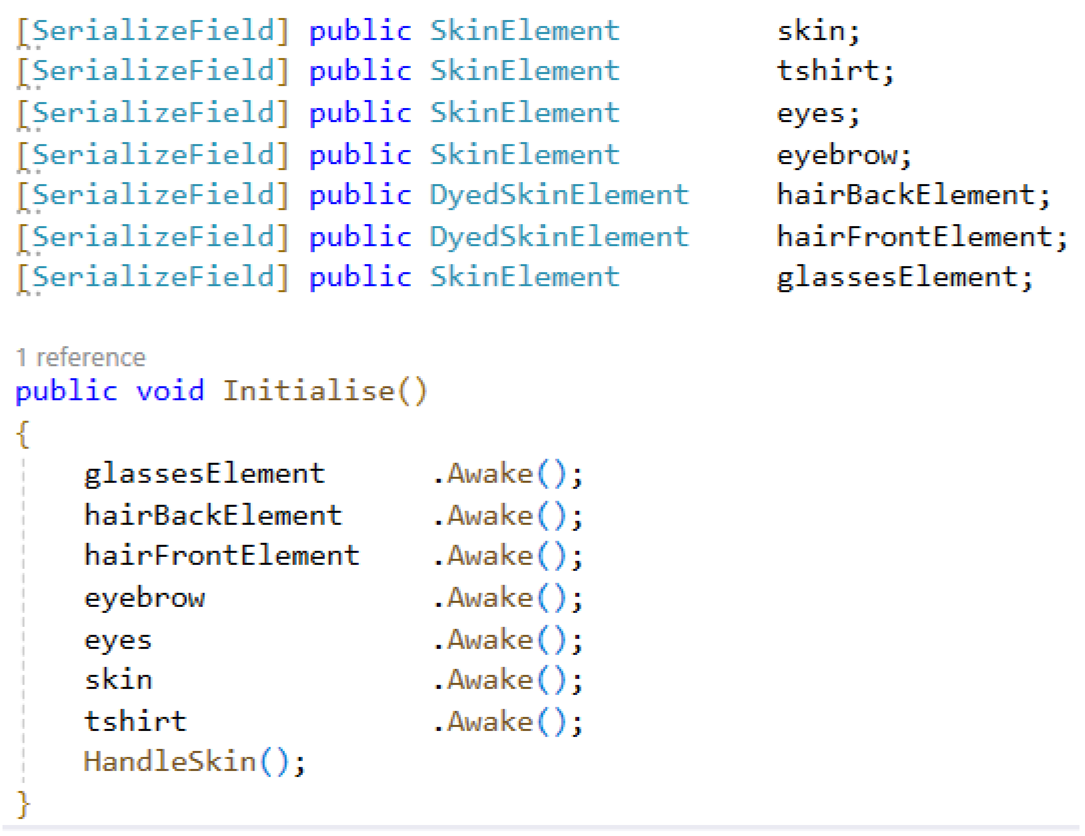
Despite me having this document I still didn't quite understand how our artists would have to implement the system within Spine 2D itself. So, I linked them the article that I found, and they understood immediately. This helped my case because I was send multiple test and prototype skeletons that I could use within Unity to build an early prototype of the system. This is also the reason I think the Character Customization is one of the best build systems in our game currently. Since it dealt with less time crunch than the other systems I worked on, due to the accidental confusion on my part and the sheer amount of time it takes the artist to make the art for this system.

The character customization system ties in very neatly with the variable system and is extremely easy to work with once you understand it. It is rather easy to show classes of serialized data within the Unity Inspector, so I made great use of this. I wrote one system that can easily handle every customization option on its own so if the artists make a change to the character (they do this a lot) it is very easy to adapt. I think the character customization part turned out great and as expected by the team.




Inspector | Urine Editor | Urine Editor

Script	TestSkinSetter
Skeleton Animation	Spine GameObject (skeleton) (Skeleton Animation)
Skin	
Skin Name	Skin_Colour
Skin Elements	3
Element 0	Skin_Colour/Skin_Light
Element 1	Skin_Colour/Skin_Medium
Element 2	Skin_Colour/Skin_Dark
Start Index	1
Tshirt	
Skin Name	TShirt
Skin Elements	5
Element 0	Tshirt_Colour/Tshirt_Blue
Element 1	Tshirt_Colour/Tshirt_Green
Element 2	Tshirt_Colour/Tshirt_Pink
Element 3	Tshirt_Colour/Tshirt_Purple
Element 4	Tshirt_Colour/Tshirt_Red
Start Index	0
Eyes	
Skin Name	Eyes
Skin Elements	3
Element 0	Eye_Colour/Eyes_Blue
Element 1	Eye_Colour/Eyes_Brown
Element 2	Eye_Colour/Eyes_Green
Start Index	2
Eyebrow	
Skin Name	Eyebrow
Skin Elements	2
Element 0	Eyebrows/Eyebrows_Dark
Element 1	Eyebrows/Eyebrows_Light
Start Index	0
Hair Back Element	
Skin Name	HairBack
Skin Elements	4
Element 0	Hair_Coily/HairBACK_Coily
Element 1	Hair_Curly/HairBACK_Curly
Element 2	Hair_StraightLong/HairBACK_StraightLong



```
[SerializeField] public SkinElement         skin;
[SerializeField] public SkinElement         tshirt;
[SerializeField] public SkinElement         eyes;
[SerializeField] public SkinElement         eyebrow;
[SerializeField] public DyedSkinElement     hairBackElement;
[SerializeField] public DyedSkinElement     hairFrontElement;
[SerializeField] public SkinElement         glassesElement;

1 reference
public void Initialise()
{
    glassesElement      .Awake();
    hairBackElement     .Awake();
    hairFrontElement    .Awake();
    eyebrow             .Awake();
    eyes                .Awake();
    skin                .Awake();
    tshirt              .Awake();
    HandleSkin();
}
```

Dialogue system

The dialogue system has been thought out by the designers and contains many mock-ups on Figma. I build it together with our designer. The basic implementation of the dialogue system was very simple and crude. It would hold lines of text in the Unity inspector and display them one after another on a text element with a grey background.

Over time when my systems evolved, I had to make the dialogue system evolve with it as well. I tackled this a bit earlier with the scriptable objects, but instead of using in scene inspector references to dialogue I wanted to make it scriptable objects. This switch was rather easy due to the freedom scriptable objects provide over direct scene references. Not only that, but it is also a lot more stable and easier to use.

The dialogue system now uses scriptable objects that very easily allow for dialogue to be created by anyone and have it run in game. I build dialogue scenes with our designers, and we implemented them into the game.

Systems I didn’t mention

The game has many more systems than the ones I mentioned in this DevLog. The reason I didn’t bring them up is because I would sound like a broken record. All systems I build are always build in the same way. Where I figure out the problem I am trying to solve, compare potential solutions against each other, start performing tests to see what is viable; where possible with my target audience as well, implement the feature(s), test it with the audience and start back from any previous steps that become relevant.

Reflection

This reflection will be a short reflection on how I think I performed this project and in the group. The project had two engineers which was quite frankly overkill for what we had to produce, and not only that, our skills in coding are also varying immensely. Because I wanted to achieve my learning goals and actually improve my coding skills it sometimes became hard to involve the other engineer in my processes. I regret not spending more time on this because it could have been a good learning process for the both of us.

Regardless of this overall work in the team went very smoothly. We adapted very quickly to each other and quickly made processes and rules of how to do things. We also held weekly planning meetings and were aware of what people were doing. I'm very happy with how easily we worked together. Not everything went well of course, but I have a reflection on that specifically.

Publishing

We have published a website where our game is playable as the theatre wanted.

References:

https://esotericsoftware.com/
https://www.geeksforgeeks.org/a-search-algorithm/
https://learn.unity.com/project/beginner-ai-pathfinding
https://github.com/LordEnma/YandereSimulatorDecompiled/blob/main/Assembly-CSharp/YandereScript.cs
https://esotericsoftware.com/spine-unity-mix-and-match
https://docs.unity3d.com/Manual/class-ScriptableObject.html
https://www.wilminktheater.nl/?srsltid=AfmBOoqx9egJ_iZiUGM6YrWpzKkptJECRRx_Oa4gBxs8hoPdynYDctpQ