University of Applied Sciences Darmstadt

Faculty of Media

Bachelor of Arts Animation and Game

Semester 3

We Sports: Last Resort

Logline: “We Sports: Last Resort” is an sport and action parody game, where you step into a clumsy agent, which causes a zombie apocalypse in a resort and fights through the zombies with a tennis racket and tries to cure the zombies in a facility.

Mike Li, 1117838

Date of Submission: 07.02.2023

1. Introduction

‘We Sports: Last Resort’ is a PC parody game of an already existing Wii game called “Wii Sports Resort”, which uses a device called a “Wii Mote” with “Motion Plus Extension” to play several sport related games. In this game the player slashes through zombies invading the town, created from poison spread out through the entire town due to a clumsy agent protagonist breaking the poisons container in mid-air, and reaches the end of the level with the belief of saving this outbreak. This game not only uses the Wii Motion Plus Remote to swing the sword, but also integrates the “Wii Balance Board”, a device tracking the standing user’s balance by pressure points on the board, to make the player step to move forward. Programming this in Unity, thankfully with already existing plug-ins, for the PC with external hardware, which is not dedicated to a Windows PC is a lot more than a challenge and will be explained further in this document along with some learning points and challenges from the implementations in the game detailed in the analysis part.

2. Individual Contributions

In summary the writer’s individual contributions are as follows:
15% A+G Design, 75% Game Development/Technical Art, 10% Methodology

2.1 A+G Design

In the Design Department generally as Game Designer, the game’s core loop and mechanics needed to be adapted constantly througout the whole process due to scope being decreased by too much stress from sparse time. Originally the game had 3 levels with different gameplay, from sword fighting on foot to shooting tennis balls on a jetski, and an additional intro level, dedicated to introduce player to the balance board controls with skydiving., with switchable weapons during the level, e.g. bowling a bomb at zombies in the distance. Now the game core loop consists of moving forward in the level, then encounter and defeat zombie waves and continue moving forward. Because being the programmer means that working with the Unitor Editor itself is self-explanatory, tweaking stats from enemies (e.g. health, move speed) and the player (e.g. step distance, attack damage), balancing zombies to the player (e.g. by adding shields to prevent player just rushing through the level) and adding UI animations for the “Player Dead” screen.

2.2 Game Development and Technical Art

The Technical side was the main part of this project. For one programming this project, which means creating structures and mechanics for the game, which the design parameters are editable in the Unity Editor, dealing with the Wii Hardware for the PC and also managing version control for the entire team using ‘Git’ and the universities’ ‘GitLab’, and for the other doing tech art work by implementing an animator for the main gameplay part and multi scene management, which was greatly beneficial in combination with version control. One of the reasons why the scope was so high and had to be decreased by a lot was, not only the already high expectations the team as a whole had for this project, but also the Wii Hardware connecting to the PC, but that will be described in the analysis part with more detail.

2.3 A+G Methodology

The Methodology percentage consists of writing the “Producing and Production Management” paper and this project documentation. The Production Manager of the team also got help by lending a helping hand with the “Miro” webpage and little insight of the previous projects.

3 Analysis

3.1 Connecting and Integration of Wii Hardware to the PC

This section completely handles the topic of connecting Wii Hardware, with that it specifically means a Wii Remote with Motion Plus (either built in or as extension) and a Wii Balance Board, to the PC and further integrate it with the Unity Editor. In the Unity Editor the first asset, which makes the implementation possible was an, now outdated, asset from 2015 (Flafla2. 2015), which made it possible to integrate one of the devices needed in the project: The Wii Remote with Motion Plus. This asset is used for the first prototype and works well enough for an asset. Though there is one restriction: There is no Wii Balance Board support and also not all Wii Remotes are useable, like the other asset, which was used in the final build. Data send out from the devices is made easier due to a plug-in provided from the ‘Unity Asset Store’, in particular one called “WiiBuddy”, released in the year 2019, which provides C# functions to connect both devices and use the data for gameplay programming in the process and is the only asset for Unity, which can make the Wii Balance Board work (Unity Asset Store, 2019). Unfortunately this asset also has its downsides. For one connection with Wii devices work in general, but sometimes with unwanted errors (e.g. not being able to find the Wii Motion Plus accessory on the active Wii Remote, despite the Motion Plus being plugged in the hardware itself or even built in, not being able to calibrate the Motion Plus at all) and it also does not include “RVL-CNT-01-TR” and further serial number Wii Remotes, which only limits the variety of Wii Remotes, which are possible to use. This problem can not be fixed with using both assets. The reason being, they block each other out on which device is belonging to which API of the individual assets making it impossible to use it together in one Unity project. This problem of compatibility is also extremly visible in connecting the Wii Remotes with a PC in the first place. Wii devices connects with the help of bluetooth, which also applies on normal Wii consoles (Nintendo. 2006). The only computer, which can handle connection with the Wii devices is the one of the gameplay programmer, which is incredibly inconvenient for task distribution among the other teammates, e.g. user testing, getting footage. Noticible is also the fact that versions above Windows 10 have made it even harder to connect Wii hardware due to necessary work arounds, because a pin is normally needed for this device, whose access is not knowable. Connecting is one thing, making them active is the other, which in comparison is thankfully much easier if you know what you need to do. This project uses “Dolphin Emulator”, which has a neat option menu for bluetooth connected Wii Remotes and other accessories, to make the Wii hardware even useable. From there on, the plug-in should do the rest sometimes, but not consistently.

In Conclusion: Although there are good enough plug-ins to use in Unity, the Wii hardware is definitely not reliable for PC connection and if there are other options (like trying out a software development kit or better: using hardware already compatible with the PC, like VR-Hand controllers) definitively try those out first, before using this technology from roughly 2006 (Nintendo). The experience gathered in this project does not approve using Wii devices without having troubles with trust issues in connectivity and also making it playable, when it is published.

3.2 Feature Creep

In combination with having hardware troubles, everyone in the team wanted to implement different kind of games inside of the project what comes naturally from a parody game, which in retrospect was too ambitious and leads to a lot of scope changes. The team already anticipated that the hardware is going to be a very problematic bottleneck, but did not know how severe it was and therefore did a lot of planning ahead. In the end, the learning was to plan one major mechanic for the game instead of multiple smaller ones, it is not planable to what extent of work is needed for more mechanics. One gameplay-loop makes it more manageable as an experience of a game and can be expanded upon more.

3.3 Multi Scene Management

In this project the multiscene approach by adding additive scenes (Unity. 2022) in the Unity Editor is used to separate different components necessary for a complete gameplay experience into categories and adaptable amount of scenes for each teammembers, making it far more easier to avoid merge-conflicts in version control systems. Without this system there are going to be more mandatory discussions and communications with teammembers to hold on their actions on one scene, so that another teammember can change something in the same scene for a level. This is because Unity does save the scene information in text form, but merging changes together, unlike in assets to a certain extent, is more fatal in a way that the scene is getting corrupt and unusable, which means you have to get an old version again and replace by merging the versions together with the already corrupt scene, which is quite annoying. There is an editor-only asset called “MultiSceneSetup”, which makes open up combinations of scenes far easier (svermeulen. 2016). With that asset additional scenes can be saved up as a ‘scriptable object’ asset and be open up with the exact same order of scenes in the hierarchy. The only downside is that the asset is an ‘editor-script’, meaning that the scriptable objects can not be used in loading these scenes bundled together with that if the game needs to be build. What this project now has is a complete copy of the scriptable object called “SceneGroups” saving up the scenes like the “MultiSceneSetup” scriptable object, but only for saving up the scene and its properties and being able to be called in scripts without trouble. The benefit of multi scene management in general is also that each scene can be viewed separately or put together with other scenes, so duplicates of game objects in different scenes can be avoided. Also blending scenes together can made much more easily with additional scenes by loading and unloading scenes asynchronously (Unity. 2022). In this project there were still times, where the scenes needed to be fixed due to merge conflicts. In order to reduce that in the future, the scenes can be split in more parts so everyone can work in their own assigned scenes.

3.4 Integrating Design Principles

The third semester finally makes it possible to program a project with more organized structuring, which includes design principles from software engineering for games, which were teached in the previous semester (Leissner. 2023). Here are some implementations of three design patterns: Observer Pattern, Finite Statemachine & Object-Pooling.

3.4.1 Observer Pattern

Instead of doing everything in the Unity Monobehavior’s Update-Loop, the observer pattern can trigger methods, when a certain condition is met in oberver-classes. In this project the “Action-Delegate” system from C# (Microsoft) is used to transmit data around to make communication between decoupled scripts more eloquent and independent. The advantage of actions is also to limit its usage to specific classes or objects, for example the enemies have their own event script, which uses actions to communicate inbetween scripts individually, without having a static “Event-Manager”, which needs to check for which the event is called upon and which observer is affected.

3.4.2 Finite Statemachine

One of the most tedious pattern to build and set-up, but very useful and fast to create states for the game. In combination of inheritance, which can be used for example to trigger event calls from any states and override it if necessary, it is a very helpful tool to use for any kind of game. This state machine is implemented with having a Enter, Update/Reason and Exit method inherited from a parent class. This project also uses a struct container for transmitting data inbetween states (e.g. in order to carry player position if necessary for the zombies to calculate rotation and movement)

3.4.3 Object Pooling

At one state of the project Unity-Editor has been crashing constantly without any indication of the cause. It was suspected that it may have to do with performance spikes. In order to prevent major performance issues in the future, there exists an “EnemyManagerScript” holding two queues, for active and inactive zombies, which get enabled and reset again or should be disabled. The queue gets working, when a triggerbox gets crossed by the player and transmits spawning information, e.g.: health, speed, guard direction with shields, to the ‘EnemyManagerScript’. Another thing getting pooled are particle effects with the similar implementation of the zombie queue. At the end the actual cause of the constant Unity crashes was about spawning all zombies after entering a triggerbox at once, which causes a great performance spike, and not actually the overall performance, but now with these changes the overall performance has been improved, which is still a good benefit.

4 Conclusion and Future Work

This experience of this project has shown that working with outdated motion sensoring hardware really is not a cake walk and a challenge, which is simply not worth, at least when you work alone as a programmer. The set goal was originally to just program a game with motion controls, but it ended up with more learning experience that overall it can be said, that even though motivation has been decreased more and more, the overall project has been a very helpful experience. Trying to maintain a solid project structure, implementing design principles, more version control management and overall better understanding of programming, at least with C# and the Unity Editor. Next time the focus should be to be able to develop a prototype faster, working together with another programmer to see how to communicate and be efficient together, learn to understand and use third-party assets faster and differentiate if it works better for the game or not.

List of References

- Leissner, Martin. 2023. “GAME DEVELOPMENT 2 | SS23”. Dieburg
- Flafla2. 2015. “C# / Unity Wii Remote API”. https://github.com/Flafla2/Unity-Wiimote
- Nintendo. May 9, 2006. https://web.archive.org/web/20080212080618/http://wii.nintendo.com/controller.jsp
- Nintendo. “Nintendo History”. https://www.nintendo.co.uk/Hardware/Nintendo-History/Nintendo-History-625945.html
- Nintendo. “Wii”. https://www.nintendo.co.uk/Hardware/Nintendo-History/Wii/Wii-636022.html
- svermeulen. 2016. “MultiSceneSetup.cs”. https://gist.github.com/svermeulen/8927b29b2bfab4e84c950b6788b0c677
- Microsoft. “Action Delegat”. https://learn.microsoft.com/de-de/dotnet/api/system.action?view=net-8.0

Unity:

- Unity Asset Store. 2019. “WiiBuddy”. https://assetstore.unity.com/packages/tools/input-management/wiibuddy-4929
- Unity. 2022. “LoadSceneMode.Additive” https://docs.unity3d.com/ScriptReference/SceneManagement.LoadSceneMode.Additive.html
- Unity. 2022. “SceneManager.LoadSceneAsync”. https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadSceneAsync.html

Appendices

Important source codes in the game folder location: ‘Oui Sports - Last Resort\Assets\Scripts’ will be listed here:

- ScriptableObjects\LoadSceneSetups.cs
- EnemyScripts\Zombies\Scripts\ZombieScript.cs
- EnemyScripts\Zombies\ZombieStateMachine.cs
- EnemyScripts\Zombies\ZombieStateVariableContainer.cs
- EnemyScripts\EnemyManagerScript.cs
- Effects\ParticleEffectManager.cs
- FigureCharacterController.cs
- PlayerParticleSystem.cs

No third party assets are relevant for this document.