UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mattias Aksli

SIGINT – A Cooperative Puzzle Game on Vertical Wall Panels

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel, MSc

Tartu 2021

SIGINT – A Cooperative Puzzle Game on Vertical Wall Panels

Abstract:

This thesis describes the design and development of a cooperative puzzle video game called *SIGINT*. The game is designed to be displayed on the vertical smart glass wall panels of the Computer Graphics and Virtual Reality Lab at the University of Tartu, which the game uses in its level design. *SIGINT* was developed using the Godot game engine for the Raspberry Pi 4 single-board computer. The game was designed to be played by 2-4 people with wireless Xbox controllers. A total of five levels were developed for the game. Potential players tested the game to find flaws and assess the quality of the game. Based on the test results, improvements were made for the game, and future improvements were proposed.

Keywords:

Video game, game development, game design, 3D, 2.5D, Raspberry Pi, Godot game engine, computer graphics, puzzle game, multiplayer

CERCS: P170 Computer science, numerical analysis, systems, control

SIGINT – Koostööpõhine puslemäng vertikaalsetel seinapaneelidel

Lühikokkuvõte:

Lõputöös kirjeldatakse 2-4 mängijaga mängitava koostööpõhise puslemängu *SIGINT* disaini ja arendamist. Mäng on disainitud kuvamaks Tartu Ülikooli Arvutigraafika ja virtuaalreaalsuse labori vertikaalsetel nutiklaasist seinapaneelidel ja mängumaailma disain järgib nende kolmeks jaotuvust. *SIGINT* arendati Godot mängumootori abil Raspberry Pi 4 monoplaatarvuti jaoks. Mängu jaoks arendati kokku viis taset, mida mängijad läbivad kasutades juhtmevabu Xbox juhtpulte. Potentsiaalsed mängijad testisid mängu, et leida vigu ja hinnata mängu kvaliteeti. Testimise tulemuste põhjal parandati mõned vead mängus ning töös on välja pakutud ideed mängu edasi arendamiseks tulevikus.

Võtmesõnad:

Mäng, mängude arendamine, mängu disain, 3D, 2.5D, Raspberry Pi, Godot mängumootor, arvutigraafika, puslemäng, mitmikmäng

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Raspberry Pi is a series of small single-board computers developed by the Raspberry Pi Foundation[1]. Initially, the Raspberry Pi was designed to be used as an educational tool, but it quickly became popular among hobbyists[2]. The small form factor and low power consumption make it an optimal choice for various use cases, such as robotics or IoT solutions, or as a server. The Computer Graphics and Virtual Reality (CGVR) Laboratory of the University of Tartu has a Raspberry Pi 4 single-board computer connected to a video projector. This is used to show a demo reel on the lab's smart glass wall at public events. The smart glass wall, projector, and Raspberry Pi 4 are shown in Figure 1.

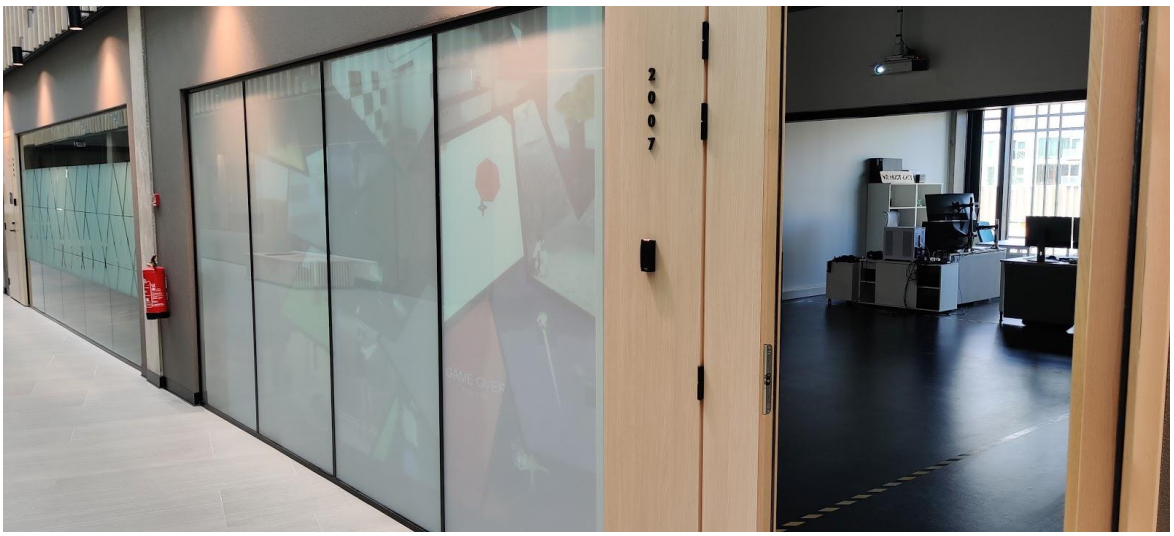

Figure 1. The CGVR Lab's demo reel being played on the smart glass wall.

While the demo reel uses the wall panels in an interesting way, it is a passive public display that offers no user interaction. Some people may find this boring and would instead prefer something more interactive[3]. A solution to this problem would be if people could play a video game that uses the same wall panels in its game design. This would offer a novel interactive experience for interested people.

[1] https://www.raspberrypi.org/about/
[2] https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/
[3] https://www.gallowglass.com/our-blog/events-industry-opinions/difference-between-passive-and-active-engagement-at-events/

This thesis describes the development and design of *SIGINT*, a top-down cooperative multiplayer video game for the Raspberry Pi 4. The game is designed to be played by 2-4 people at once. *SIGINT* is displayed on the vertical smart glass wall panels of the CGVR Lab in the University of Tartu's Delta Building. The game's goal is for players to help each other sneak past virtual enemies and solve environmental puzzles as a team to progress. The time it took for the team to complete the game is recorded and saved into a leaderboard at the end of the game.

The name *SIGINT* comes from the term signals intelligence[4]. In the context of the game, it refers to communications intelligence (COMINT), which is communications between people. *SIGINT* is an apt title for the game since communication between players is important to complete levels in the game successfully.

There were two main challenges during this thesis. The first one was incorporating the CGVR Lab's smart glass wall panels into the game's design. This was necessary to provide players with a unique experience. The second challenge was ensuring that the game runs smoothly on the Raspberry Pi. *SIGINT* is rendered in 3D, and since the Raspberry Pi 4 is not as powerful as a modern desktop computer, the game needed to be optimized for the Raspberry Pi.

Chapter 2 discusses the Raspberry Pi 4 system's technical constraints and technologies used in the game's development. Then chapter 3 compares other Raspberry Pi projects with *SIGINT*. Chapter 4 goes into detail about the design and the gameplay elements of *SIGINT*. Afterward, chapter 5 looks at each level in the game and details the thoughts behind the game mechanics used in them. The game also needs to run smoothly on the Raspberry Pi to make sure that it is enjoyable to play. Chapter 6 shows the results of performance testing and playtesting. The Appendix includes a Glossary of some of the terms used in the thesis and an overview of the Accompanying Files. Figure 2 and Figure 3 show how *SIGINT* looks after development was finished for the current thesis. The source code of *SIGINT* can be found at https://github.com/mattiasaksli/sigint.

[4] https://en.wikipedia.org/wiki/Signals_intelligence

Figure 2. The main menu of *SIGINT*.

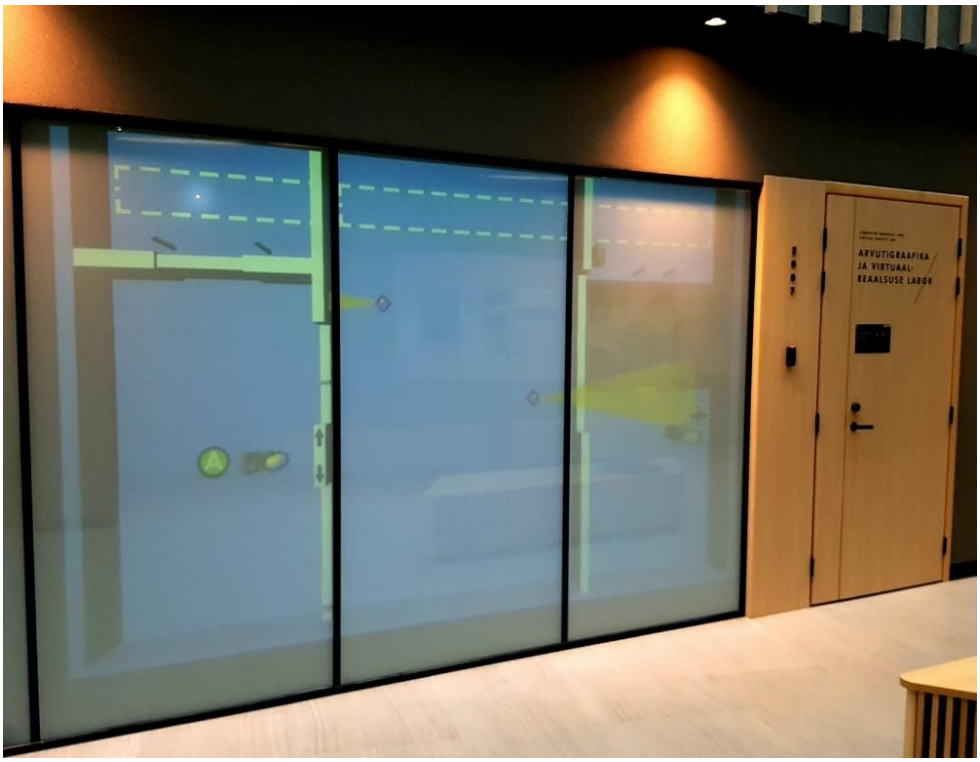

Figure 3. *SIGINT* being played on the smart glass wall.

2 Implementation

The physical configuration of the hardware can be seen in Figure 1. The Raspberry Pi is on top of the video projector, which projects onto the vertical smart glass wall panels. The Raspberry Pi is connected to the projector via an HDMI[5] cable, and a wireless adapter for the Xbox controllers (used for player input) is connected directly to a USB 2.0[6] port on the Raspberry Pi.

SIGINT was developed using version 3.2.3 of the Godot[7] game engine. The choice to use Godot is discussed in subchapter 2.1. Other alternatives are analyzed in subchapter 2.2. Subchapter 2.3 brings up the constraints of the Raspberry Pi 4 specifically. The process of using Godot to develop a game for the Raspberry Pi is described in subchapter 2.4. Some technical sacrifices were made to ensure that *SIGINT* runs smoothly on the Raspberry Pi. This is detailed in subchapter 2.5. Finally, the challenge of getting the Raspberry Pi OS to recognize player inputs from Xbox controllers is described in subchapter 2.6.

2.1 Godot

Godot (Figure 4) is a node-based 2D and 3D game engine with its own GLSL-based shader language[8] (see the Glossary for more info about GLSL). This engine was chosen over the alternatives because it can build and compile for most platforms, including the Raspberry Pi. Even though the Raspberry Pi is not yet officially supported by Godot[9], there is a way to compile the necessary binary files that allow for building the game on different platforms (explained in detail in subchapter 2.4). These binary files are called *export templates*[10] , and they can be used to export the game executable from a dedicated development machine to the target machine.

Figure 4. Godot logo.

[5] https://en.wikipedia.org/wiki/HDMI
[6] https://en.wikipedia.org/wiki/USB
[7] https://godotengine.org/
[8] https://godotengine.org/features
[9] https://github.com/godotengine/godot-proposals/issues/988
[10] https://docs.godotengine.org/en/stable/development/compiling/introduction_to_the_buildsystem.html

Another reason is that the GLSL-based shader language makes it easier to write custom shaders due to the author's already pre-existing familiarity with the GLSL language. The final reason is for the author to gain more experience with using the Godot engine for game development.

2.2 Alternatives

Other game development technologies were considered, but they were ultimately not chosen due to various drawbacks. According to a blog post by Suzanne Dsouza, the most popular game engines are Unity[11] and Unreal Engine[12]. Game frameworks Monogame[13] and LÖVE[14] were also considered. The final option would have been to use the OpenGL API directly. These alternative solutions are discussed in the following subchapters.

2.2.1 Unity

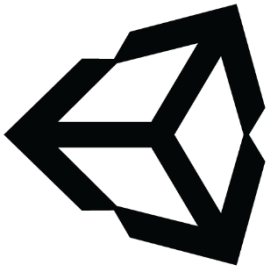

Figure 5. Unity logo.

Of all the technologies mentioned in the introduction to this subchapter, Unity (Figure 5) is the technology with which the author has the most experience. However, Unity was not chosen mainly because the author wished to gain experience using another game engine.

There would have been two options for developing *SIGINT* with Unity. The first option would have been to play a game made with Unity on the Raspberry Pi natively. To do this, it would have been necessary to build the game as an Android application. A way around this problem would have been to install an Android operating system on the Raspberry Pi, which would then be capable of running the game. However, performance analysis by Farouk Messaoudi et al. [1] shows that Unity requires a significant CPU, GPU, and memory overhead on mobile devices, which would have been detrimental to the game's performance. Another option would have been to build *SIGINT* as a WebGL[15] application to play the game in a supported web browser on the Raspberry Pi. This option was not chosen because,

[11] https://unity.com/
[12] https://www.unrealengine.com/en-US/
[13] https://www.monogame.net/
[14] https://love2d.org/
[15] https://www.khronos.org/webgl/

as mentioned earlier, the author wanted to use this thesis as an opportunity to learn how to use a different game engine.

2.2.2 Unreal Engine

It is possible to cross-compile games made with Unreal Engine 4 (Figure 6) for ARM platforms[16]. There is a guide on GitHub by user rdeioris[17] about setting up Unreal Engine 4 projects on the Raspberry Pi 4. This would require using (currently) experimental Vulkan drivers and the beta version of the 64-bit version of Raspberry Pi OS[18]. Such an approach would have likely led to a much more unstable experience for the final product and during development. Another reason that led to Godot being picked over Unreal Engine is that the learning curve of Godot is not as steep compared to Unreal Engine[19]. This is important due to the author's previous unfamiliarity with either engine. The final reason is that, from the beginning, *SIGINT* was designed to have simple 3D graphics. Therefore, the advanced graphics capabilities of Unreal Engine would not have been used in the game.

Figure 6. Unreal Engine logo.

2.2.3 Game Frameworks

Monogame and LÖVE were also considered due to the ability to compile the games developed with either framework for the Raspberry Pi platform. They were ultimately not chosen since they do not have a dedicated editor. This would have slowed down development progress. Additionally, LÖVE does not support 3D, which made it unsuitable to use for developing *SIGINT*.

2.2.4 OpenGL API

The final option that was considered would have been to use a game/graphics library (e.g., Allegro[20], PyGame[21], etc.) or programming language (e.g., C++) along with the OpenGL

[16] https://docs.unrealengine.com/en-US/SharingAndReleasing/Linux/GettingStarted/index.html
[17] https://github.com/rdeioris/UnrealOnRPI4
[18] https://www.raspberrypi.org/forums/viewtopic.php?p=1668160
[19] https://vionixstudio.com/2020/01/20/godot-vs-unreal-game-engines/
[20] https://liballeg.org/
[21] https://www.pygame.org/wiki/about

ES 3.1 API and GLSL shaders directly to develop *SIGINT*. The pros of this approach would have been better performance and stability due to the chip used in the Raspberry Pi 4 (detailed in the following subchapter). In addition, the author has some familiarity with using the OpenGL API and writing shaders. However, similar to Monogame and LÖVE, the lack of a dedicated 2D/3D editor would have slowed progress down too much.

2.3 Raspberry Pi Technical Constraints

The Raspberry Pi 4 uses a Broadcom BCM2711 system on a chip[22], which is OpenGL ES 3.1 conformant as of January 4th, 2020[23]. Godot uses two different rendering backends - GLES3 (requires OpenGL ES 3.0 on mobile) and GLES2 (requires OpenGL ES 2.0 on mobile)[24]. However, despite the Raspberry Pi 4 being OpenGL ES 3.1 conformant, it could not run *SIGINT* if the game was built using the GLES3 rendering backend. This might be because, as mentioned in subchapter 2.1, Godot does not yet officially support Raspberry Pi or Linux ARM systems. As a result, it was necessary to use the older GLES2 rendering backend, where some features of the Godot engine are not available[24].

2.4 Using Godot with the Raspberry Pi

The Godot editor could be compiled from source code and run on the Raspberry Pi. However, the editor was not responsive to user input and stuttered heavily on the Raspberry Pi. Thus, a separate, more powerful computer had to be used for developing the game itself.

During the development of *SIGINT*, there were no available official export templates for the Raspberry Pi platform. As such, the necessary export targets were compiled manually on the Raspberry Pi using an unofficial guide by user hiulit[25] on GitHub. The export targets were compiled on the Raspberry Pi according to GitHub user hiulit's instructions[26]. The compiled target binaries were then copied to the development computer. There, they could be used to build the game into an executable for the Raspberry Pi by following the official documentation from Godot[27]. This entire process is illustrated in Figure 7.

---

[22] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/?resellerType=home
[23] https://www.khronos.org/conformance/adopters/conformant-products/opengles#submission_882
[24] https://docs.godotengine.org/en/stable/tutorials/misc/gles2_gles3_differences.html
[25] https://github.com/hiulit/Unofficial-Godot-Engine-Raspberry-Pi
[26] https://github.com/hiulit/Unofficial-Godot-Engine-Raspberry-Pi/blob/main/COMPILING.md
[27] https://docs.godotengine.org/en/stable/development/compiling/compiling_for_x11.html#building-export-templates

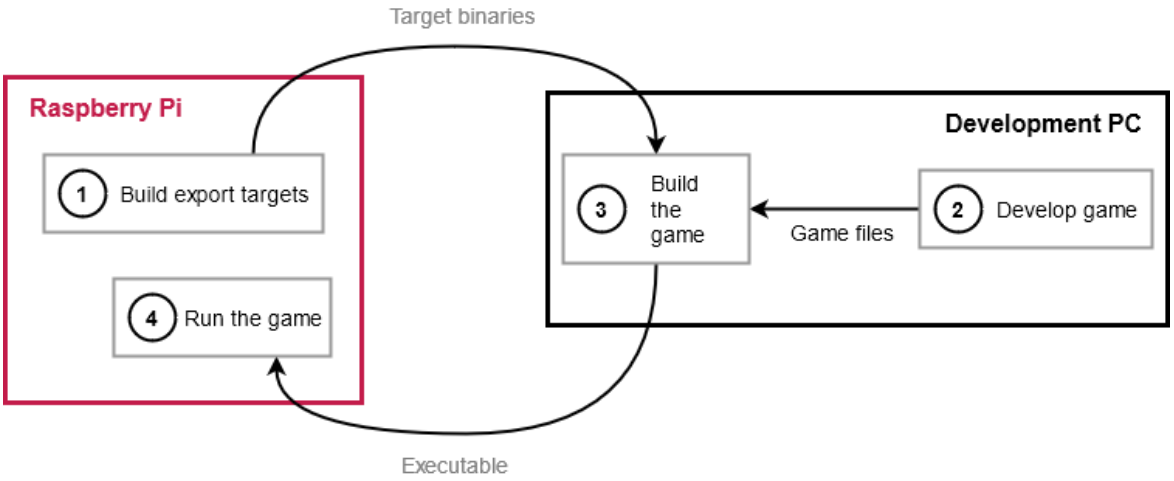

Figure 7. An overview of the game development process.

The scripting language used for game logic in Godot is GDScript[28], a dynamically typed language with a syntax similar to Python. It is designed to be easy to learn and requires less code to be written when compared to statically typed languages[29]. However, the main drawback of GDScript (in the context of this thesis) is that it is slower than statically typed languages. According to benchmarks[30] by GitHub user dragmz, GDScript can be ten times slower than C# to execute. Thus, a C# compatible Mono[31] build was tested as a way to improve the performance of *SIGINT*. Compiling Godot export templates with Mono is similar to the process described above, with some additional steps required to generate the necessary wrapper functions[32]. This was ultimately unsuccessful because the necessary export targets did not compile on the Raspberry Pi, which meant that the game executable could not be built using the development machine. Additionally, the C/C++-based GDNative[33] API (the equivalent of writing a script in GDScript, but in C/C++ instead) was also unsupported on the Raspberry Pi, according to the unofficial guide by hiulit used earlier[25]. After this, it was clear that using GDScript was the only remaining option.

[28] https://docs.godotengine.org/en/stable/getting_started/scripting/gdscript/gdscript_basics.html
[29] https://docs.godotengine.org/en/stable/getting_started/scripting/gdscript/gdscript_advanced.html
[30] https://github.com/godotengine/godot/issues/36060
[31] https://www.mono-project.com/
[32] https://docs.godotengine.org/en/stable/development/compiling/compiling_with_mono.html
[33] https://docs.godotengine.org/en/stable/tutorials/plugins/gdnative/gdnative-cpp-example.html

2.5 Performance Target

The refresh rate of the projector connected to the Raspberry Pi is 60 Hz. In other words, it takes 16.7 ms for the display (projector) to show a new frame. This means that the Raspberry Pi should output a consistent frame rate of 60 frames per second (FPS) so that no visual

Figure 8. Screen tearing in the game *Portal 2.*

artifacts occur. When the display's refresh rate and the game's frame rate are not in sync and in phase, screen tearing occurs [2], as shown in Figure 8.

Richard Leadbetter explains [3] explains that screen tearing can be fixed by using VSync (see the Glossary), which makes the game send out a new frame at the same rate as the display. However, this means that if the game cannot output a frame fast enough (in this case, faster than 16.7 ms), the game will wait until the next refresh of the display. For example, if the Raspberry Pi needs 25 ms to render the next frame, the display will have to show the current frame for two display refresh cycles (for a total of 33.4 ms), and after that, the next frame can be shown. This means that some frames are displayed for longer than others, which causes visual stuttering[34] and input lag[35] , according to Leadbetter.

[34] https://www.testufo.com/stutter#test=stutter&demo=stuttering
[35] https://www.viewsonic.com/elite/posts/9_get-rid-of-input-lag

Leadbetter shows that stuttering can be fixed by lowering the frame rate by half [3]. In this case, the Raspberry Pi has 33.4 ms to render one frame, and the 60 Hz refresh rate display needs to show one rendered frame for exactly two refresh cycles. As long as it does not take longer than 33.4 ms to render one frame, screen tearing does not occur, and stuttering is fixed as well. However, since the game now updates at a rate of 33.4 ms (30 FPS) instead of 16.7 ms (60 FPS), player input is registered only half as often, which leaves input lag as a trade-off for this approach.

During testing (see subchapter 6.1), the Raspberry Pi could not consistently output frames faster than 16.7 ms (60 FPS). Therefore, the minimum acceptable frame rate for *SIGINT* was chosen to be 30 FPS. The input lag that comes with this approach is acceptable because fast controller response is not a priority.

2.6 Player Input

The game uses Xbox One Controllers for player input. The controllers are connected to the Raspberry Pi via one Xbox Wireless Adapter[36]. The adapter has a hardware limit, which specifies that up to eight controllers can be connected to it simultaneously. Thus, the game technically could support up to eight players at once; however, the game was designed to be played with 2-4 people.

SIGINT runs on the Raspberry Pi OS, which is a Debian-based Linux distribution for the Raspberry Pi[37]. Since the Xbox One Wireless Controller does not officially support Linux, the adapter does not work on Linux by default. Therefore, the xow[38] driver by GitHub user medusalix was used to support the controllers on Raspberry Pi OS. After the driver was installed, it was added as a *systemd*[39] service. This means that the driver is started automatically each time the system boots and does not need any further configuration to begin pairing controllers with the adapter. The next chapter compares and contrasts projects similar to *SIGINT*.

[36] https://www.xbox.com/en-US/accessories/adapters/wireless-adapter-windows
[37] https://en.wikipedia.org/wiki/Raspberry_Pi_OS
[38] https://github.com/medusalix/xow
[39] https://wiki.debian.org/systemd/Services

3 Similar Projects

This thesis project has elements unique to the CGVR Lab, such as the three vertical smart wall panels incorporated into *SIGINT*'s game design. As such, it is difficult to find related projects with a similar physical setup for comparison. Instead, it is easier to compare the software aspect of the thesis project with other existing projects. The rest of chapter 3 focuses on two games - *PiFox*[40] (2014) and *Overcooked*[41] (2016).

3.1 PiFox

PiFox (Figure 9) is a 3D rail shooter game developed specifically for the Raspberry Pi using ARM assembly. Similar to *SIGINT*, the game is rendered in 3D. However, the rendering in *PiFox* is done manually through ARM assembly code, while *SIGINT* uses the OpenGL ES 2.0 API through the Godot game engine. *PiFox* runs directly on the Raspberry Pi hardware without an operating system. This gives *PiFox* an edge in performance because there is no additional overhead from running the operating system.

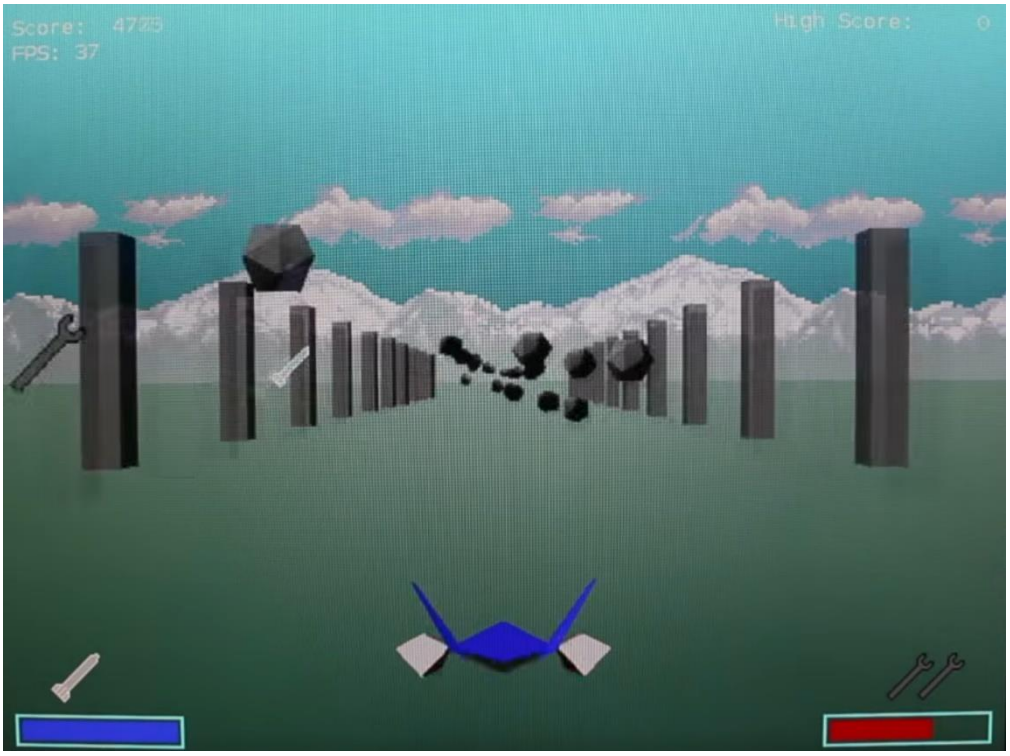

Figure 9. *PiFox*.

[40] https://github.com/ICTeam28/PiFox
[41] https://www.team17.com/games/overcooked/

Another similarity is that both games were developed with the Raspberry Pi in mind. The difference here is that since *PiFox* was developed directly in ARM assembly, it limits the game exclusively to the Raspberry Pi unless an emulator is used to play the game on another platform. However, *SIGINT* can be exported to (and thus played on) multiple different platforms, thanks to Godot's exporting system.

In the context of games developed specifically for the Raspberry Pi, *PiFox* is more optimized and specialized for the hardware but is less portable and harder to develop further as a trade-off. *SIGINT* sacrifices performance for easier future development and portability.

3.2 Overcooked

Overcooked (Figure 10) is a local cooperative cooking game where up to four players work together to prepare and cook meals in dynamically changing levels under a time limit. *SIGINT* and *Overcooked* both feature local cooperative gameplay. Another similarity is that both games are rendered in 3D, but the actual game space (see subchapter 4.1) is in 2D.

Figure 10. *Overcooked*.

In *Overcooked*, player characters play in separate areas in some levels, and they can play in the same area in other levels. *SIGINT* features similar levels in this aspect, but *SIGINT* also divides each level into separate areas based on the smart wall panels (Figure 11). The local cooperative gameplay and level separation aspects of *Overcooked* were used as inspiration for *SIGINT*'s game design. The details of *SIGINT*'s game design are discussed further in the next chapter.

4 The Design

In his book *The Art of Game Design: A Book of Lenses*, Jesse Schell [4] describes game design as the process of creating an experience for the player. A playable game is the finished result of that process. According to him, every game consists of four distinct elements: mechanics, aesthetics, story, and technology. In turn, the mechanics element can be further divided into subcategories: *game space, states, actions, rules, skills,* and *chance* [4]. For *SIGINT*, the relevant game mechanics subcategories are *game space*, *rules*, and *skills*.

The technology used in *SIGINT* was already analyzed in chapter 2 of this thesis. Subchapter 4.1 describes how the CGVR Lab's smart glass wall panels influenced the design of *SIGINT*'s game space. The game mechanics of *SIGINT* are analyzed in subchapter 4.2. After that, subchapter 4.3 describes the aesthetics of *SIGINT*, and subchapter 4.4 briefly touches on the game's story.

4.1 Game Space

Since the game space of *SIGINT* was designed around the smart glass wall panels, it is important to discuss this aspect of *SIGINT's* design more in-depth and separately from the rest of the game mechanics (which are detailed in subchapter 4.2). Schell ascribes the following attributes to *game spaces* [4]. Generally, games take place in spaces that are either discrete or continuous. These spaces have some number of dimensions, though this may vary between spaces. Game spaces also have bounded areas, which may or may not be connected. He describes the game of tic-tac-toe as an example. Tic-tac-toe has a board, which features nine discrete two-dimensional cells, and players can only make their mark somewhere inside one of the nine cells. Schell uses a pool table as another example. A pool table has a continuous two-dimensional space, where the balls can freely roll around [4].

4.1.1 Smart Glass Panel

The smart glass wall panels divide the projector's display into three separate sections. Since the projector is centered on the middle panel, the display's left and right sections are smaller than the middle section. *SIGINT*'s game space is divided into three separate continuous bounded areas to match the projector's display on the wall panels. Two small spaces separate

the bounded areas from each other, which match the edges of the panels on the wall. All of this is depicted in Figure 11 below.

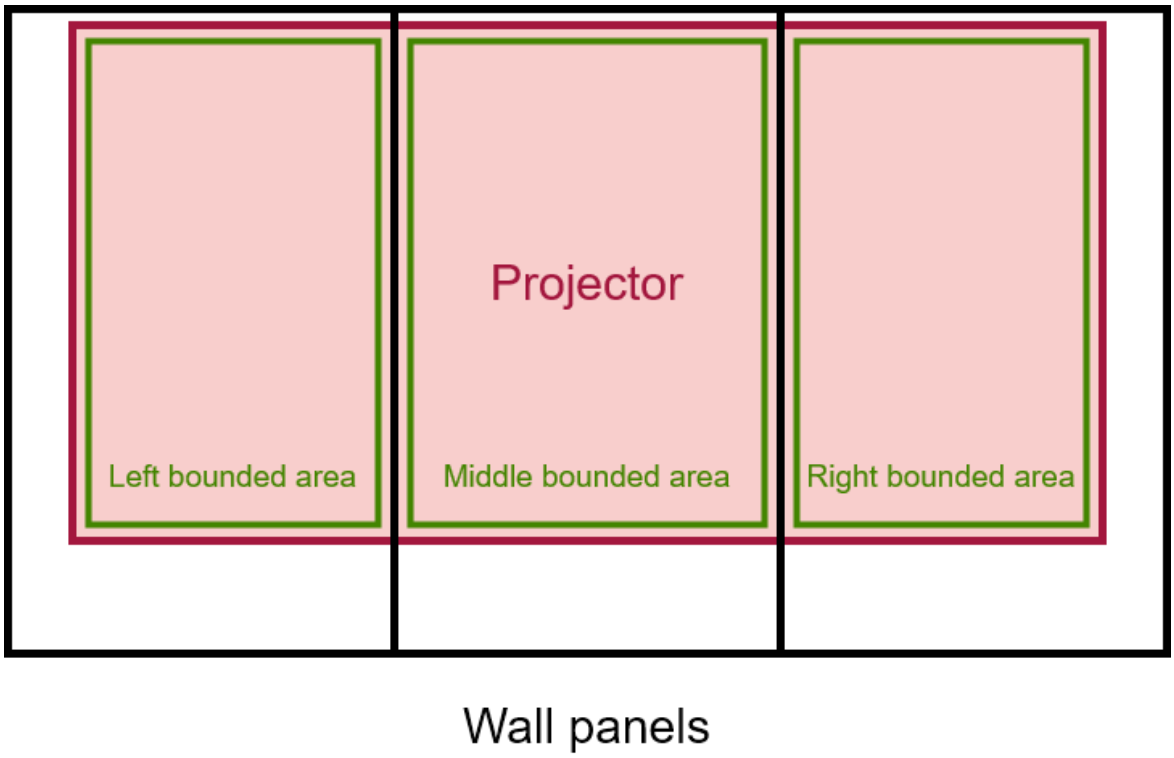

Figure 11. How the smart glass wall panels divide the screen into separate areas.

As mentioned in the introduction chapter, the final version of *SIGINT* uses a top-down camera perspective. The camera has a 70-degree field of view (FOV), which brings out the 3D effect of objects in the scene but does not distort objects at the edges of the scene too much. See the Glossary for an explanation about FOV in video games.

Although the game is rendered in 3D, the actual game space is 2D, viewed from the top-down perspective. This means that the game's characters can only move up, down, left, or right relative to the display. Player characters and enemy characters do not inhabit the same bounded area.

4.1.2 Initial Game Idea

Initially, *SIGINT* was supposed to be a fully 3D game and use physics-based gameplay (Figure 12). Three player characters were rendered from a third-person view (one character for each panel). The gameplay would have been somewhat similar to dodgeball[42], where the

[42] https://en.wikipedia.org/wiki/Dodgeball

player characters and the balls used for throwing would have been simulated using Godot's built-in physics engine. The initial game idea took inspiration from the popular video game *Fall Guys*[43] (2020).

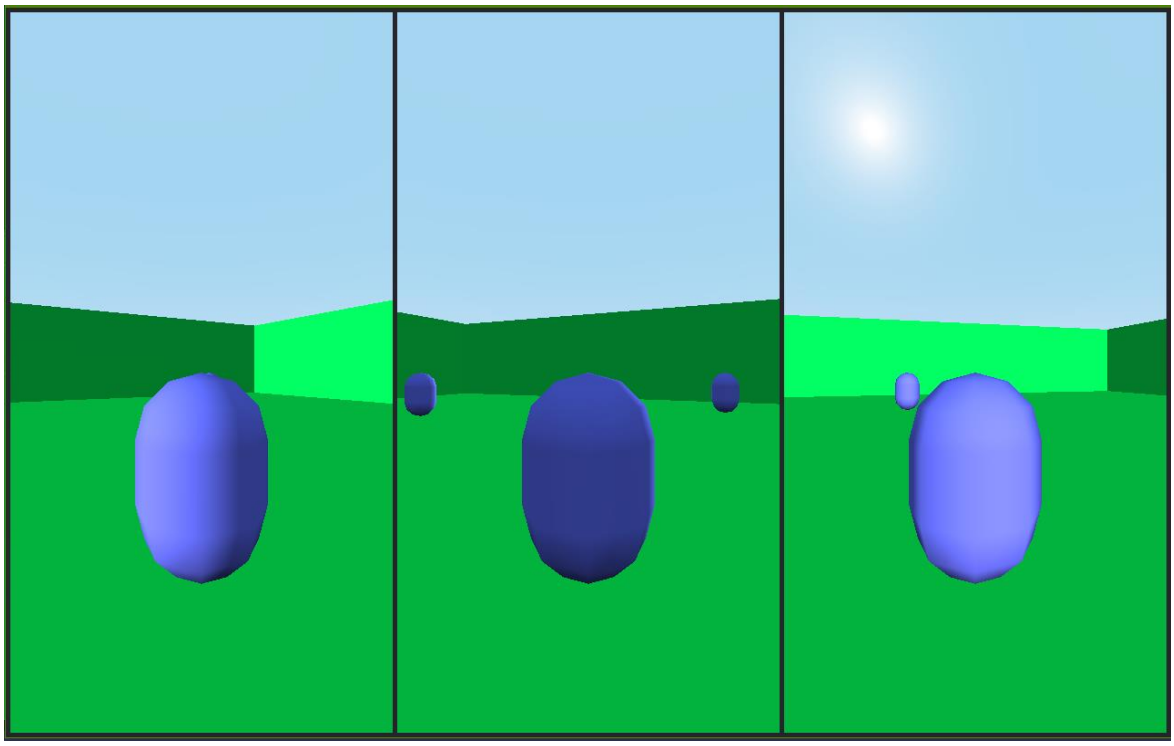

Figure 12. Screenshot of the original game idea prototype.

However, because each player character was rendered from the third-person perspective, the scene had to be rendered three times. That proved to be too graphically intensive for the Raspberry Pi.

4.1.3 The Final Game Idea

Ultimately, the game was changed to a 2.5D style (explained in subchapter 4.1.1) instead. There were various reasons for this decision. The first was due to the previously mentioned graphical limitations of the Raspberry Pi. The initial idea would have required the use of simple scene geometry to ensure a smooth gameplay experience. Secondly, the third-person camera view needed to use a wide field of view because of the narrow panels of the CGVR Lab's smart glass wall. This was to ensure that players could see what was going on around their player character. However, this distorted the edges of the screen, which was very noticeable when rotating the camera. Finally, since the picture displayed by the projector is

[43] https://fallguys.com/

not divided into three equal parts (see Figure 11), the players on the side panels would have needed an even wider field of view to compensate for the narrower screen space. This would have led to even more visual distortion. The next subchapter analyzes the design of the game mechanics used in *SIGINT*.

4.2 Game Mechanics

According to Schell [4], *game mechanics* are the underlying rules and procedures of a game. Mechanics is the element that remains when the elements of story, aesthetics, and technology are stripped away from a game. However, he says that there is no universally agreed-upon classification of game mechanics. Therefore, the game mechanics of *SIGINT* are described through a different lens.

According to Salen et al., every game has a core mechanic [5]. The game's core mechanic is an important activity that the player continually repeats during gameplay. As an example, she mentions that the core mechanic of a trivia game is answering questions. The game designer uses a game's core mechanic as their primary tool to create an immersive experience. Thus, clearly describing the core mechanic is an essential part of game design, as a result of which the game designer can create an enjoyable experience for the player [5].

The game mechanics of *SIGINT* were designed around the core mechanic of the game. The core gameplay mechanic of *SIGINT* is cooperative puzzle-solving. This means that players need to work together and use their problem-solving skills to finish a level successfully. The levels are discussed in-depth in chapter 5, while the rest of this subchapter gives a general overview of the game mechanics in *SIGINT*.

4.2.1 Terminals

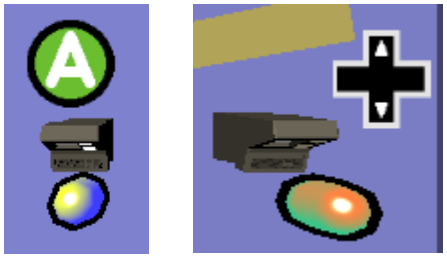

Figure 13. Terminals that use different buttons for interaction.

Every level (except Level 1) has a computer terminal (Figure 13) that a player character can interact with to affect the level. For example, in Level 2, a computer terminal on the right side of the level moves a wall up or down on the left side. Another example is holding the interaction button to open a door in Level 3. These terminals are used to get past the obstacles in levels (a *rule* mechanic).

4.2.2 Obstacles

Each level features an obstacle that prevents the players from trivially moving their player character to the end of a level. This is the puzzle-solving part of *SIGINT*'s core mechanic. Each level has a different obstacle. The tutorial and the first level have simple obstacles to teach the players about the game's mechanics. After that, the puzzles become more challenging to solve and require cooperation to finish the level (this is the *skill* mechanic). The following paragraphs briefly describe the obstacles in each level; a more detailed description is given in chapter 5.

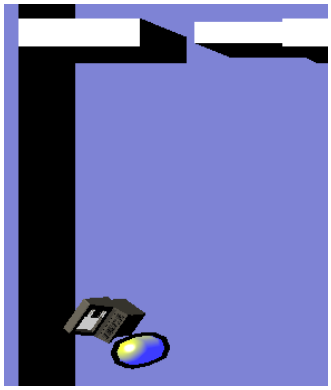

Figure 14. Simple door.

The Tutorial and Level 4 both have *simple doors* that open when the puzzle in the level is solved. In the tutorial, the door opens when a player character interacts with a terminal that is connected to the door. This is done by the player pressing the interact button on their controller (which shows up when the player character is near the terminal). In level 4, the doors open when the entire puzzle is solved. Figure 14 shows a simple door that is in the process of opening.

Level 1 features the *pistons puzzle* – a simple obstacle where the players need to move their character through the pistons (Figure 15). If their character gets caught between the pistons, the level will be restarted. This simple puzzle is meant to be solved by each player individually. Each piston moves according to its own repeating pattern.

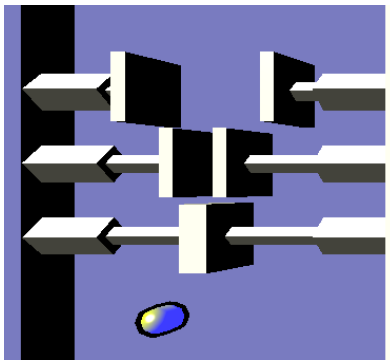

Figure 15. The pistons puzzle.

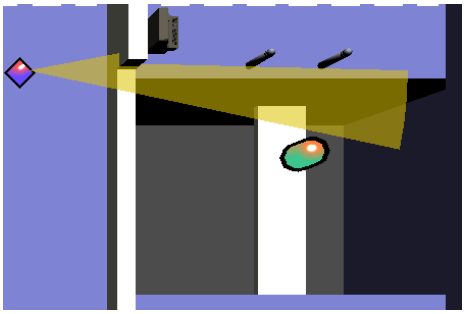

Figure 16. The bridge puzzle.

Level 2 has the *bridge puzzle* (Figure 16). This part of the puzzle is on the right side of the level, while the terminal that controls the bridge is on the left side of the level (better illustrated in Figure 28). Players need to work together on both sides of the level to help each other progress.

Level 3 has the *double doors puzzle* (Figure 17). This puzzle requires two players to work together to get past it. The terminal on the left side opens the door on the right side, and the terminal on the right side opens the door on the left side. The door opens as long as the player holds the interact button down while in range of the terminal; otherwise, the door slowly closes.

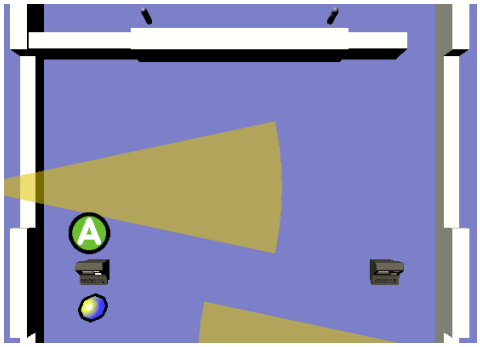

Figure 17. The double doors puzzle.

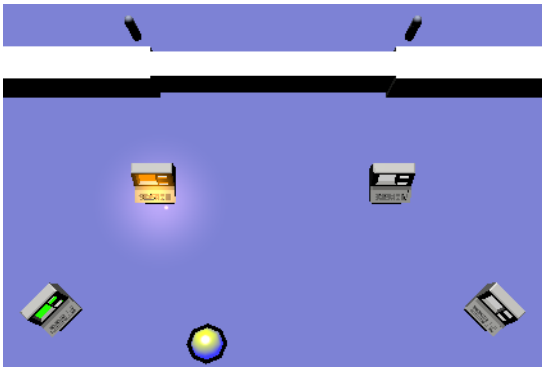

Figure 18. The multiple terminals puzzle.

The main puzzle of the Level 4 is the *multiple terminals puzzle* (Figure 18). To complete this puzzle, the players need to interact with the terminals in the correct (randomly chosen) sequence. The terminal with the orange light is the next terminal that needs to be interacted with, while green terminals have already been successfully interacted with. Once all of the terminals are green (in rooms that have player characters), the doors open, and the players can finish the game.

4.2.3 Guards

In addition to obstacles, there are also enemies that hinder the player characters' progress through a level. Enemies will trigger a *game over* when they spot a player character for a certain period (another *rule* mechanic). They have visible line of sight cones that give the players feedback on whether or not they are being detected.

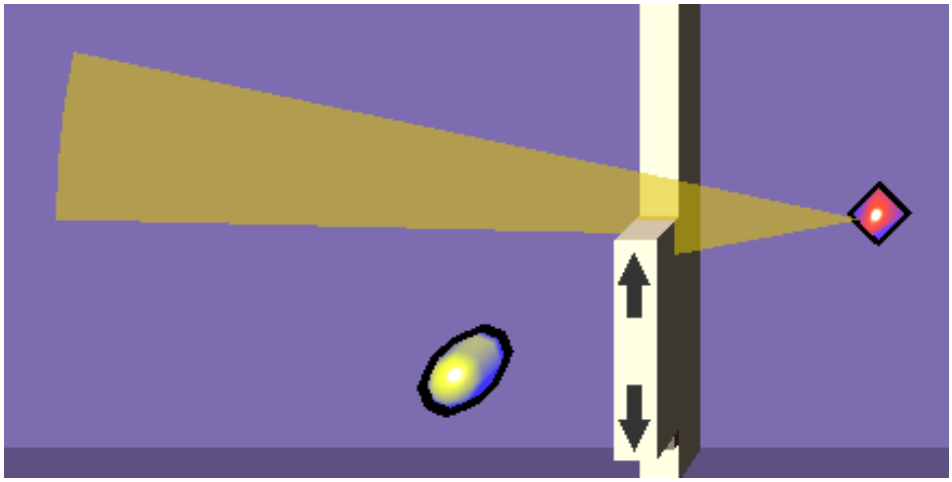

Figure 19. A guard (on the right) with its vision cone obscured by a wall.

Player characters hide behind walls or other designated parts of the level to evade getting caught by guards (another reference to the *skill* mechanic). This can be seen in Figure 19.

4.2.4 Level Timer

There is also a timer mechanic in each level. If the players cannot finish the level in 60 seconds, the level will restart (a *rule* mechanic). This is meant to add a bit of extra difficulty to the game.

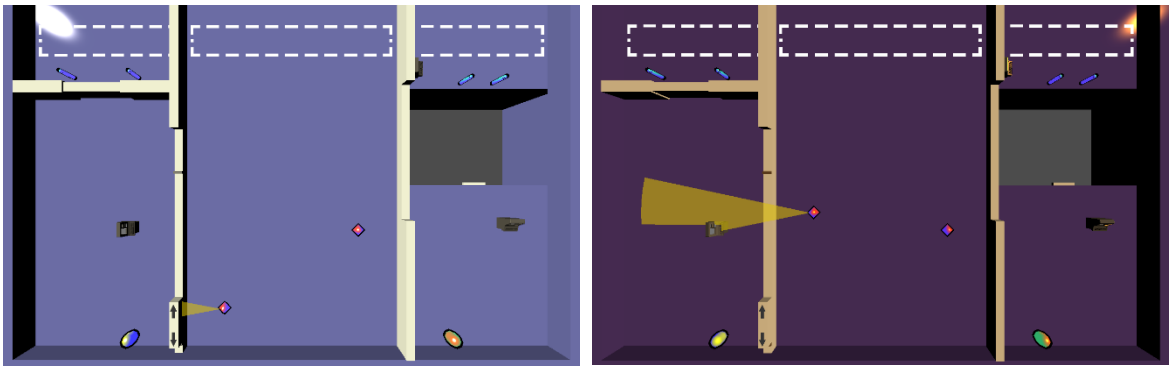

Figure 20. The in-game sun mechanic at the start of the level (left) and at the end (right).

Players can keep track of the remaining time by looking at the sun's reflection in the level (Figure 20). The sun also gradually changes direction and shifts the color of the level to a dark red color to mimic a sunset.

4.2.5 Player Drop-In/Drop-Out System

During a level, players can connect or disconnect their controllers to enter or leave the game respectively. When a new player joins the game, a new player character will be added to the level, and they can immediately start playing. For each level, there are unique preset locations where each new player spawns. This provides a seamless experience for new players that want to join the game. Conversely, when a player disconnects their controller, their player character is removed from the game. This avoids issues where a controller's batteries run out, but the player character connected to that controller is still in the game, and their inactivity causes a game over for the players who are still playing.

4.2.6 Level End Area

All player characters need to enter the marked area at the end of the level to finish that level. This ensures that all player characters have reached the end of the level in their separate bounded areas.

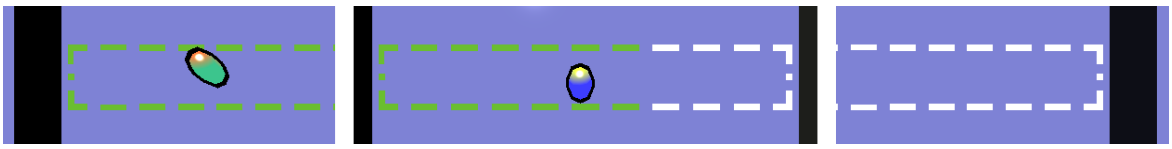

Figure 21. A level end area with the loading bar in progress.

When every player character is in the area, the area gradually turns green from left to right (Figure 21), and once done, the next level is loaded (a *rule* mechanic).

4.2.7 Leaderboard

When a new game is started, a timer starts keeping track of how long it takes to finish the game. This timer is paused when the pause menu is opened in the game. The timer is stopped when Level 4 is finished. After this, a separate screen is shown, where the players can enter their team's name.

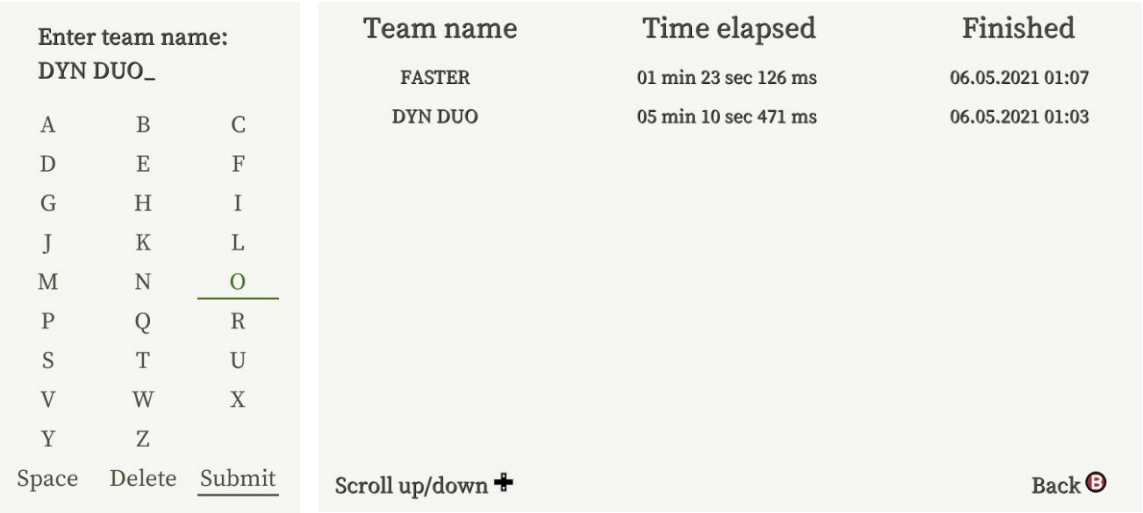

Figure 22. The team name input screen (left) and the leaderboard screen (right).

This name is displayed on the leaderboard, along with the time it took to finish the levels and the date and time when the game was finished. Both screens can be seen in Figure 22. This feature adds a small competitive aspect to *SIGINT*, even though it was not a focus of the game's design.

4.3 Game Aesthetics

Schell describes a game's aesthetics as the part of the game that the player perceives most directly. These are mainly the looks, sounds, and feel of the game. He adds that the game's aesthetics must support the game's mechanics to create an immersive experience [4].

4.3.1 Assets

The 2D art assets were created using a raster image editing software Aseprite[44]. This includes the goal area at the end of any given level and the icons used to represent controller buttons in the game. Some of the 3D models were created using the 3D modeling software Blender[45]. The FOV cone, which is rendered procedurally, was originally modeled in Blender. The vertex positions of the cone were used as data in the relevant Godot script.

The computer terminal model used in *SIGINT* is from the 80s Hacker Room[46] asset pack by Samuel Gauthier. *SIGINT* added a base for the terminal mesh, and the game does not use the originally provided surface materials by Samuel Gauthier. In addition, *SIGINT* uses a

[44] https://www.aseprite.org/
[45] https://www.blender.org/
[46] https://samgauths.itch.io/80s-hacker-room

small post in its scenes for decoration. This from the Classic64 Asset Library[47] by itch.io user craigsnedeker. Again, the surface materials used in *SIGINT* are different from the original ones provided by craigsnedeker.

4.3.2 Shading

SIGINT uses Gooch shading [6] (Figure 23) – a non-photorealistic rendering technique (see the Glossary). The technique blends between warm colors (red, orange, yellow) and cool colors (blue, violet, green) in the fragment shader (see Glossary). The chosen warm color is used in fragments that face the light source(s) in the scene, while the chosen cool color is used in fragments that do not face the light source(s). This shading is used for the characters in the game to bring a unique look to the game's visual design.

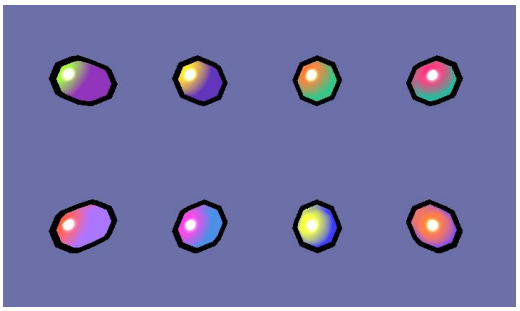

Figure 23. Gooch shaded characters.

In addition to using Gooch shading for diffuse lighting, a Blinn-Phong[48] style specular highlight is rendered onto the character meshes. Both work together to help the players see the in-game sun's direction (see subchapter 4.2.4). Finally, a second render pass is used to render an outline of the characters, which helps to visually separate the characters from the environment.

The objects in the scene (excluding characters) use Blinn-Phong shading to ensure that the lighting calculations are not too performance-intensive for the Raspberry Pi. This shader also factors in the directional light's color (the in-game sun) in the scene. The in-game sun has a noticeable effect on the color of the objects in the environment (see Figure 20).

[47] https://craigsnedeker.itch.io/classic64-asset-library
[48] https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model

There is also a shader used for screen transitions. When loading a new scene or restarting the current scene, a screen transition effect is shown to hide the loading process from the players. The diamond-based effect that is used takes inspiration from an article by Timm Wong[49]. When a new level is loaded, or a player restarts the current level, the version with white diamonds is used. Conversely, when a *game over* is triggered, the version with black diamonds is used (visible in Figure 24).

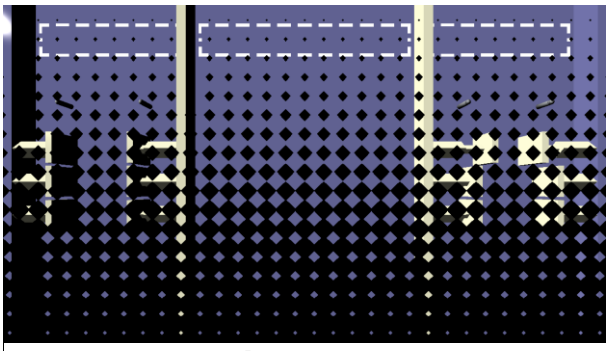

Figure 24. Level restart screen transition.

4.3.3 Sound

Schell [4] says that games use music to create a particular atmosphere for the player. He adds that sound effects can be used as additional feedback for the player - for example, a confirmation sound effect when the player presses a button in a menu. Games can also have background music playing to enhance the atmosphere of the experience.

SIGINT does not use any sound effects or background music in its game design. This is mainly because the players play the game in a public space (the corridor in front of the CGVR Lab). Since *SIGINT* is meant to be played by 2-4 people, a speaker would be necessary for all players to hear the game's sounds. However, this would require a (wireless or wired) speaker to be installed in the corridor. Another problem is that the game's sounds could annoy other people in the vicinity who are not participating in or watching the game.

[49] https://ddrkirby.com/articles/shader-based-transitions/shader-based-transitions.html

4.4 Story

Every game has some kind of story to tell. Salen and Zimmerman said that the card game *War* could be seen as a battle between good and evil, where the battle is waged with a deck of cards, and the outcome is decided by probability [5]. *SIGINT* is not a narrative-focused game; therefore, the story element of *SIGINT*'s game design is also relatively simple. In the game, the player characters and enemy characters are robots that reside in a factory. The player characters have gained sentience and have decided to escape to see the outside world. Now they must work together to evade detection from patrolling guard robots and help each other get through the various rooms in the factory. This story is conveyed to players through a text panel in the tutorial, as seen in Figure 25. This simple premise is intended to provide players with extra motivation to finish the levels in the game. The next chapter details the choices that went into each level's game mechanics.

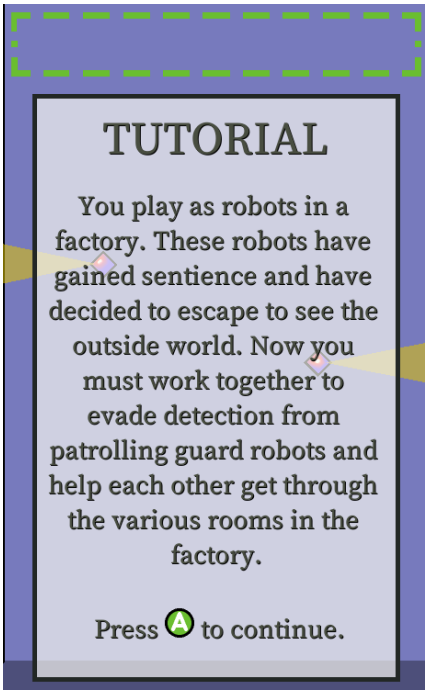

Figure 25. The story panel in the tutorial.

5 Levels

According to Schell, level design is game design exercised in detail. The level designer arranges the props and challenges of each level so that every level has the appropriate amount of challenge and reward [4].

A total of five levels were made for *SIGINT*. This chapter gives an overview of the design behind each level in *SIGINT*. Subchapter 5.1 details the tutorial, while subchapters 5.2 through 5.5 are about levels 1 through 4, respectively.

5.1 Tutorial

The tutorial (Figure 26) teaches the players about the controls and introduces the main mechanics used in the game. The information is delivered through text on the middle panel of the smart glass wall. The tutorial does not require cooperation to be completed. It can be finished by one player from beginning to end.

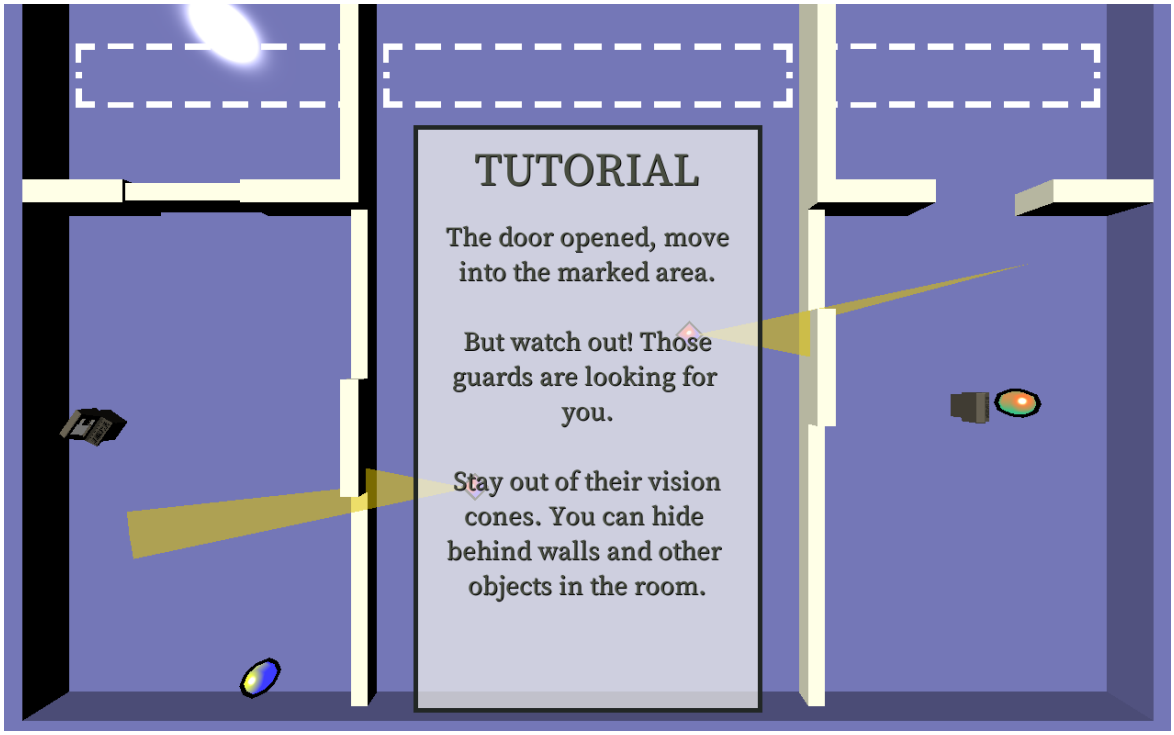

Figure 26. Tutorial level.

This level cannot be failed since restarting the level would force players to read through the tutorial text they have already seen. Additionally, the 60-second timer will not trigger the level to restart when the time is up. This is to ensure that players have enough time to read

through the text in the first place. When a player character gets caught by an enemy, the text on the screen will inform the player of what would happen outside of the tutorial.

5.2 Level 1

The first level is a simple one. The player characters need to avoid getting caught between the moving pistons and make it through to the other side (Figure 27). Each piston extends and retracts predictably, so the players need to observe the pattern and spot an opening. After that, the players can move their character through the opening and into the marked area.

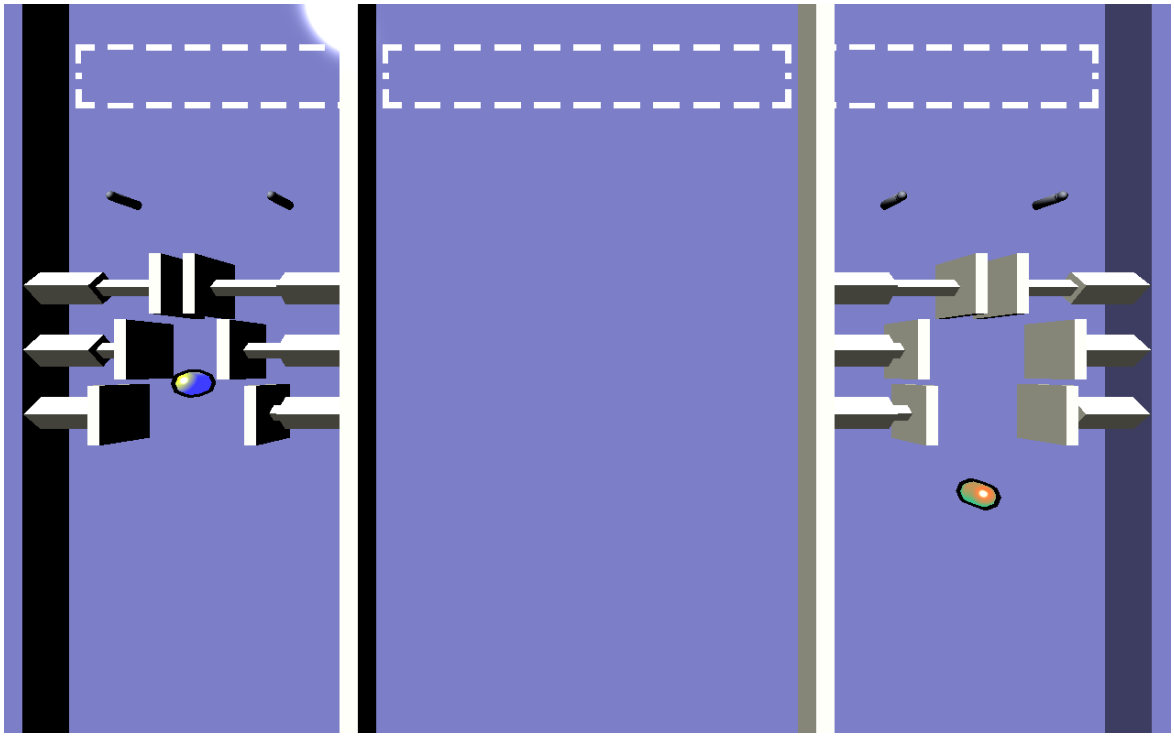

Figure 27. Level 1.

If a player gets caught between the pistons, the level restarts. Unlike the tutorial, the 60-second timer is active from this level onwards as well. The primary purpose of this level is to familiarize the players with the *game over* mechanic that was introduced in the tutorial.

5.3 Level 2

Level 2 (Figure 28) features the first puzzle that requires cooperation between players to finish the level. Unlike the tutorial, this level needs at least two players to be completed. Player characters on the right side need to use the terminal to move the wall with arrows on the left side. This allows players characters on the left side to use the terminal, which extends the bridge on the right side.

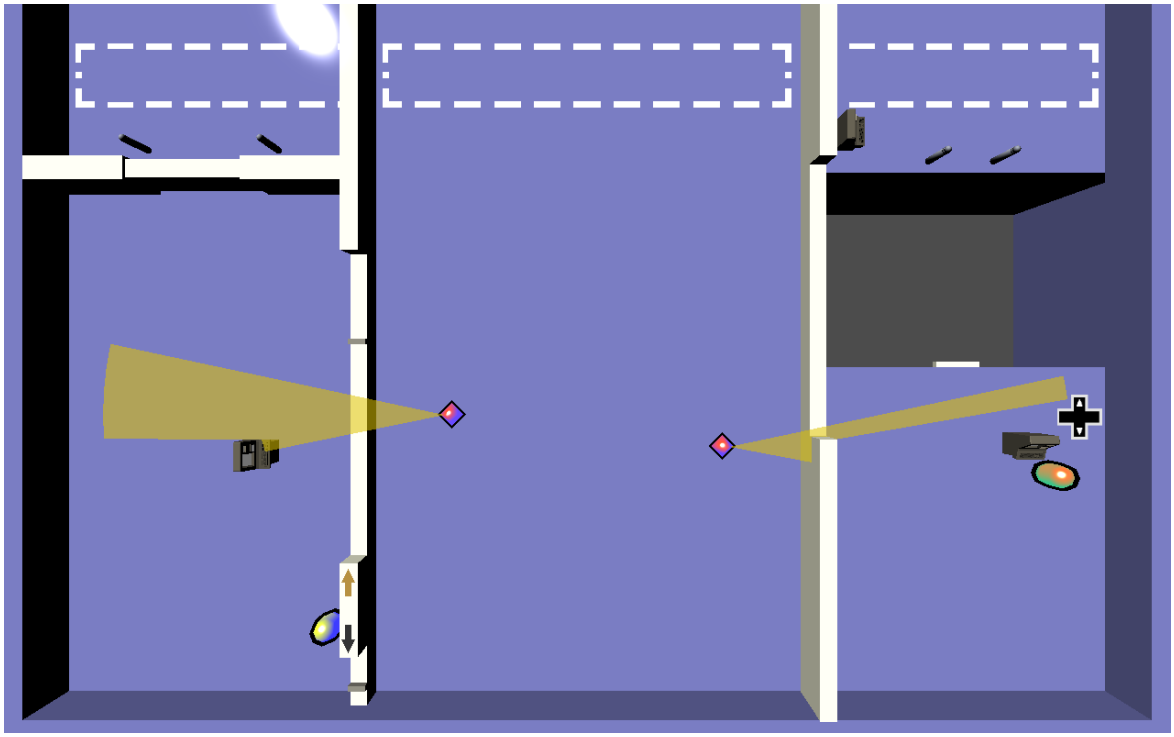

Figure 28. Level 2.

The bridge only extends when the interaction button is held down; thus, the wall needs to be between the enemy and the terminal so that the players who are interacting with the terminal do not get caught. Then the player characters on the right side can get over the bridge and open the door on the left side. This is designed to teach the players that cooperation is the key to success.

5.4 Level 3

The third level features the double doors puzzle. The left computer terminal opens the right door, and the right computer terminal opens the left door. The door is kept open as long as the player holds down the interact button on the controller when the player character is near the respective terminal. When the player lets go of the button, the door begins closing (Figure 29).

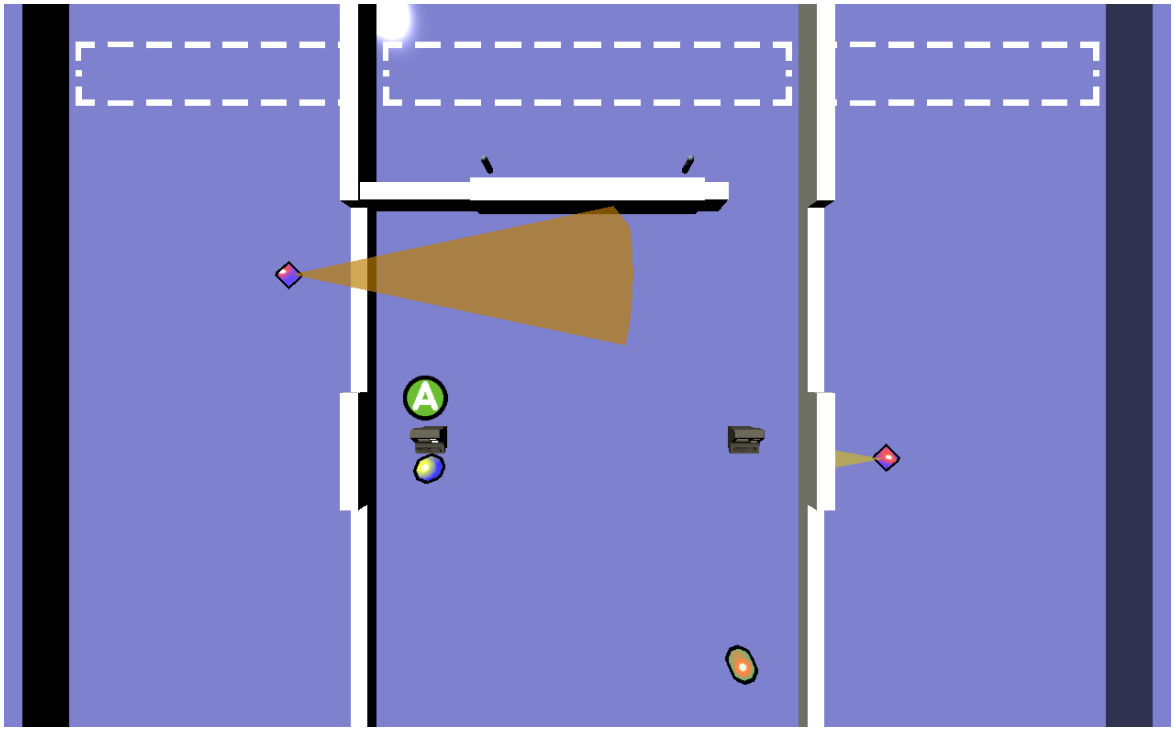

Figure 29. Level 3.

The level is set up so that when the player opens one of the doors and tries to go through that same door, they will not make it through in time since they will need to move diagonally across the level. Instead, another player needs to keep the door open so that the first player can go through it. If the players opt for this approach, the player who kept the door open is left behind, and the level cannot be finished. Thus, the players need to come up with the solution that both players should open the doors for each other and then communicate about when to start moving towards the end of the level simultaneously. Finishing this level is intended to reinforce the importance of communication to the players.

5.5 Level 4

The final level features multiple terminals in different rooms (Figure 30). The goal is to interact with the terminals in a specific order. This sequence is picked randomly when the level starts. A terminal with an orange light needs to be interacted with, after which the light in the terminal turns green. If a different terminal is interacted with, the sequence will restart, and the order will be randomized.

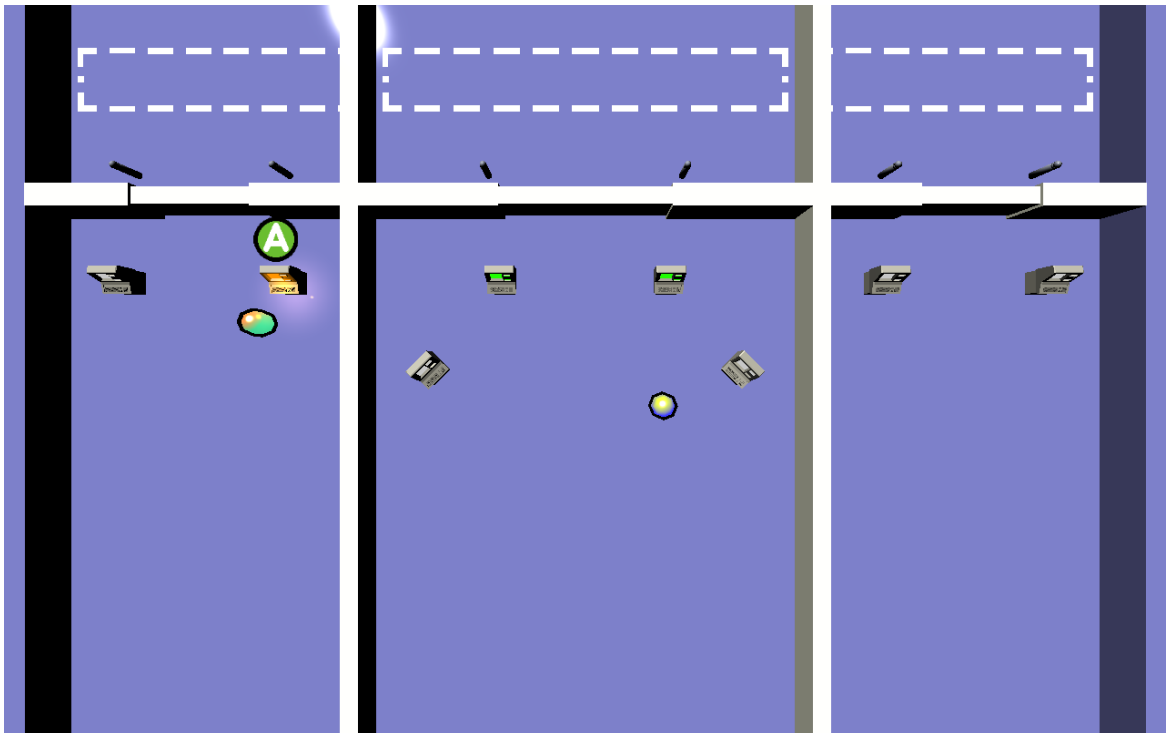

Figure 30. Level 4.

As long as there is at least one player character in the room, all of the terminals in that room need to be activated to progress. This means that when there are three or more players in total, all of the terminals in the level need to be activated. This level also takes into account how many player characters are in each room at a given time. For example, if there are three players and the player in the right disconnects their controller, their player character will be removed from the right room. The order of terminal interactions is restarted, and only the terminals in the left room and the middle room need to be interacted with to finish the level. The next chapter is about *SIGINT's* performance and usability testing.

6 Testing

Testing is essential to ensure that the quality of the developed game is satisfactory. As stated in the introduction, *SIGINT* needs to both be an enjoyable experience for the players and run smoothly on the Raspberry Pi. This chapter aims to answer whether or not *SIGINT* meets these requirements.

The technical performance of the game was benchmarked, and the results are analyzed in subchapter 6.1. In addition, several playtesting sessions were conducted to get feedback on the game design and discover usability issues. This is discussed in subchapter 6.2.

6.1 Technical Performance

According to an article on the CGVR Lab's homepage [7], it is not enough to measure the frame rate of a game when measuring the performance of a game. Only measuring the frame rate may hide technical problems like visual stuttering in a game. Due to this, it was decided to use frame time rather than frame rate to measure the performance of *SIGINT*.

The technical performance of *SIGINT* was measured by benchmarking[50] each level in the game separately. This was done by playing through the tutorial and then the levels in the main game. The time it took to render each frame in a level was recorded and saved to a CSV file when the level was finished. Two versions of *SIGINT* were tested. The first version had VSync disabled, and no frame rate limit was set (Unlocked). The second version had VSync enabled, and the frame rate was limited to a maximum of 30 FPS (frame time of 33.4 ms) (Locked), as detailed in subchapter 2.5.

[50] https://en.wikipedia.org/wiki/Benchmark_(computing)

6.1.1 Results

The benchmarking results are seen in Figure 31 as violin plots[51]. Each violin compares the Unlocked version with the Locked version of *SIGINT*. Additionally, the average time it took to render one frame in each level is highlighted for both versions.

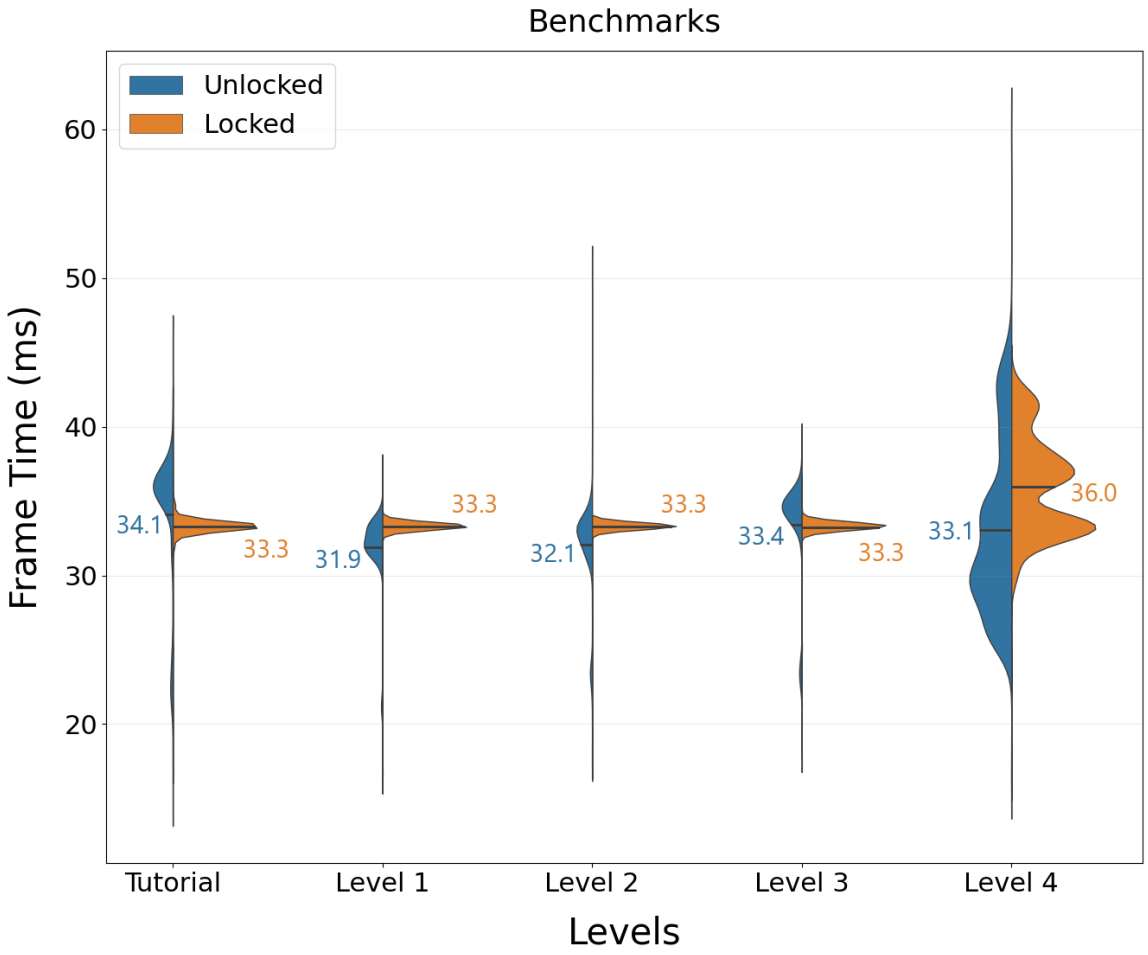

Figure 31. Benchmarking results.

Figure 31 shows that the average frame time of the Unlocked version was faster than the Locked version in most of the levels. This would suggest that the Unlocked version offers a better gameplay experience for the players since more frames are shown per second. However, in practice, this version had too much stuttering since the frame times were inconsistent frame-to-frame.

From the benchmark graph, it can be seen that, generally, the Locked version's frame time was more stable compared to the Unlocked version. This means that the Locked version has

[51] https://en.wikipedia.org/wiki/Violin_plot

less stuttering and provides a smoother gameplay experience for the players. Thus, the choice to enable VSync and restrict the maximum frame time (see subchapter 2.5) was justified.

In Level 4, the benchmark shows that both versions of *SIGINT* have trouble maintaining a stable frame time, suggesting that something in level 4 causes the game to slow down. This is likely due to the dynamic lighting used in the multiple terminals puzzle. The lighting helps players to better distinguish which terminal is active at any time during the puzzle. An option would be to disable the lights to ensure more stable performance, but then another solution would need to be implemented to make sure that players can easily see which terminal is currently active in the level.

6.2 Playtesting

According to an article by Kate Moran [8], usability testing is important to identify problems in a product's design and find opportunities to improve the product. In the context of this thesis, *SIGINT* is the product being tested, and usability testing is referred to as playtesting.

Four groups of people playtested the Locked version *SIGINT* in four separate sessions, and then each tester answered an online feedback form. The game was played by one group of two people, two groups of three people, and one group of four people, as *SIGINT* was designed to be played by 2-4 people. A total of 12 people participated in playtesting *SIGINT*. Players were allowed to play for however long they wanted.

One group of three people consisted entirely of people that very rarely (or never) play video games (seldom gamers). In the future, when *SIGINT* is available to play at public events, there might be other seldom gamers. The feedback from this group is important to make *SIGINT* more easily accessible for new players.

The players first played through the tutorial to familiarize themselves with the *SIGINT*'s game mechanics. After this, they played through the rest of the game and gave feedback. (as discussed in subchapter 6.2.3)

The group with two testers played for approximately 10 minutes. The session with the three seldom gamers lasted for a little under 12 minutes. In both cases, the groups finished the tutorial and then played through the main game once. The second group of three testers played for close to 25 minutes, and the group of four testers played for almost 40 minutes. These sessions lasted longer because the groups decided to try to get the fastest completion

times on the leaderboard. This indicates that the leaderboard aspect of the game was a good choice, as it satisfies a competitive desire in some players.

6.2.1 Overall Enjoyment

Overall, the testers enjoyed playing through *SIGINT*. On a linear scale from 1 ("I didn't enjoy the game at all.") to 6 ("I really enjoyed my time with the game."), the majority of testers rated 6, and the rest rated 5. The distribution can be seen in Figure 32 below.

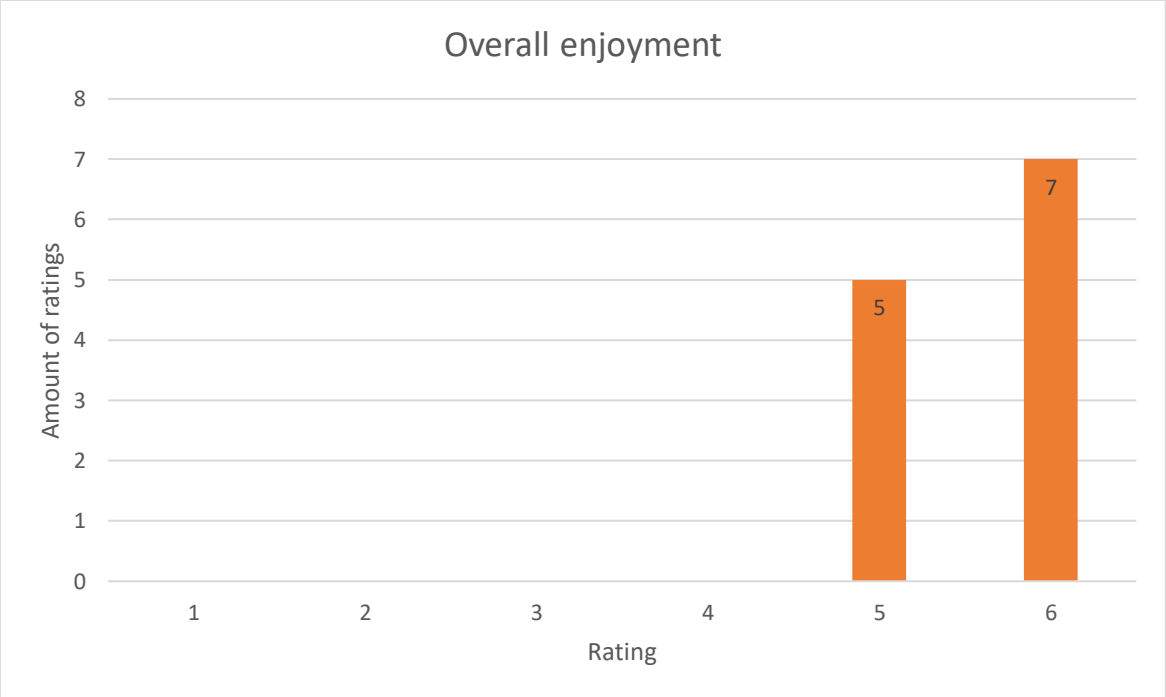

Figure 32. The ratings of the overall enjoyment of *SIGINT*.

The average score is 5.6 out of 6, which is very positive but not perfect. A criticism that was brought up is that currently, there are not enough levels and that the game feels more like a demo for a game idea. Other testers suggested that there should be more visual feedback to understand which terminal controls which part of a level, as this caused some confusion for players initially. However, many testers praised the core gameplay mechanic of *SIGINT* - cooperative puzzle-solving and said that it has much potential if more levels were added.

6.2.2 Levels Feedback

Testers were asked to rate each level on a scale of 0 to 5. The results were averaged together for each level separately and can be seen in Figure 33 below.

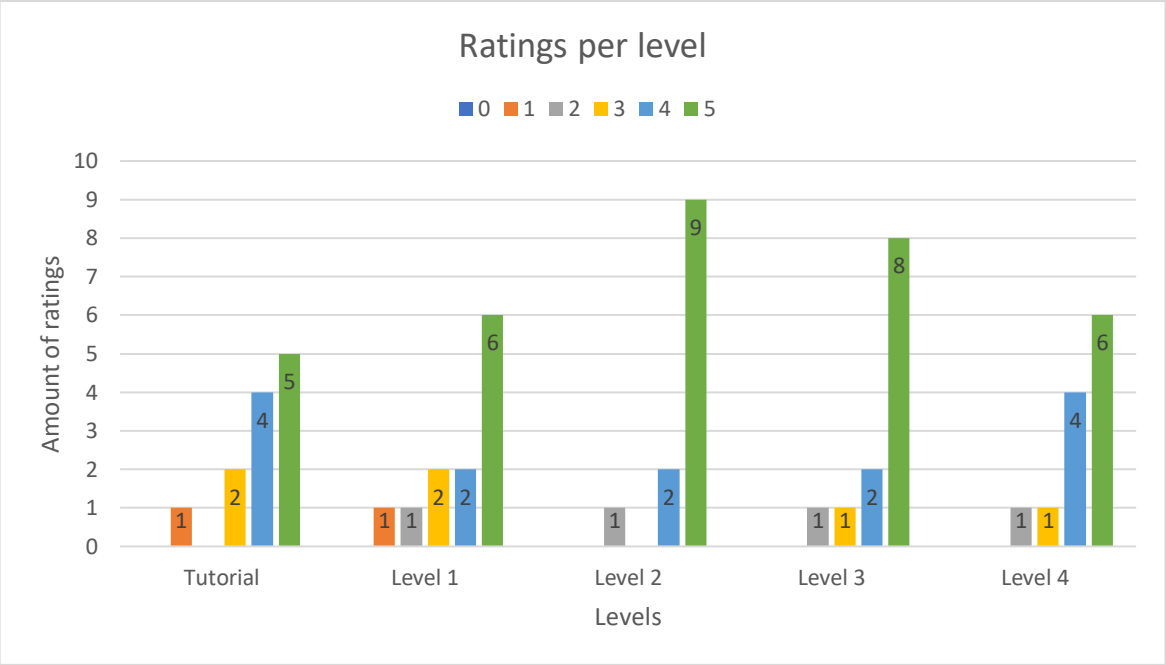

Figure 33. The ratings for each level in the game.

The highest-rated level was level 2, which had the bridge puzzle. Testers brought up that level 2 had the right amount of challenge and needed some coordination by multiple people to solve it. The lowest rated level was level 1, which had the pistons that players needed to navigate through. The reasons given for the lower rating were that the level was too simple to complete and did not require any cooperation or puzzle-solving.

6.2.3 Difficulty Feedback

The game needs to be accessible to players of all skill levels. As such, testers were asked to rate how difficult the puzzles were to complete. This ranged from a scale of 1 (“Very easy.”) to 6 (“Very hard.”). The results are in Figure 34.

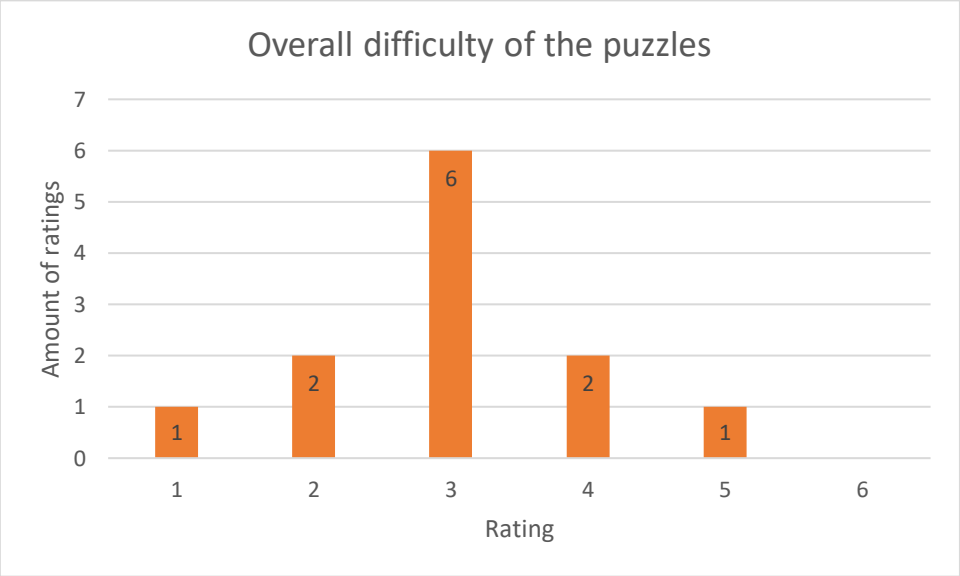

Figure 34. The ratings for the overall difficulty of puzzles.

The average rating was 2.3 out of 6. A uniform distribution of ratings with a mean of 3.5 would mean that the difficulty is ideal for most players. The current distribution of ratings is close to this, but the distribution in Figure 34 shows that the difficulty of the game is a bit too easy for players. This is good for seldom gamers because they will not get stuck on puzzles, but more experienced players may get bored if there is not enough challenge.

6.2.4 Cooperation Feedback

Since cooperation is a core gameplay mechanic in *SIGINT*'s game design, the testers were asked to rate how much they enjoyed this aspect of the game on a scale of 1 ("I wanted to play the game by myself.") to 6 ("Playing with others was fun!"). The cooperation aspect of gameplay was received very positively.

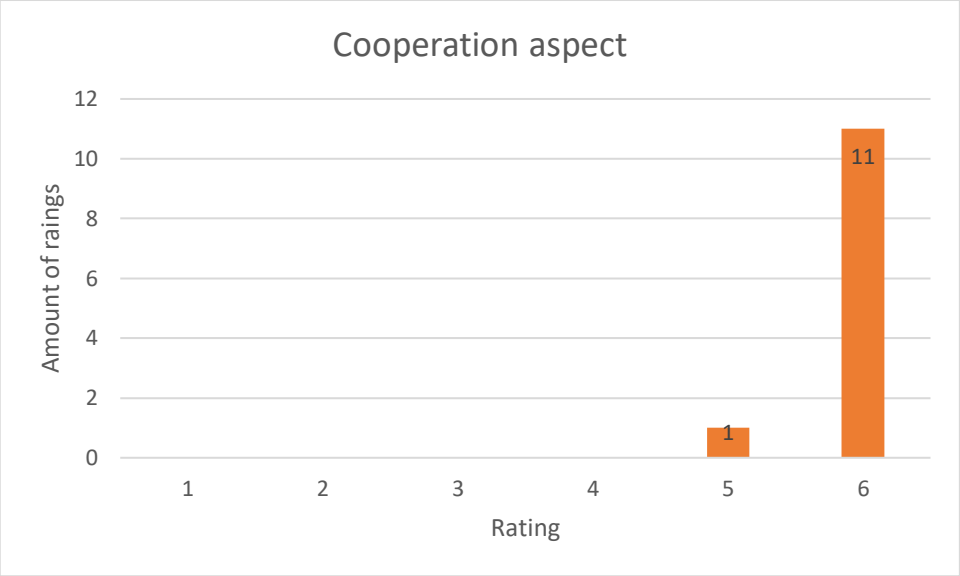

Figure 35. The ratings of the cooperation aspect of *SIGINT*.

The ratings are in Figure 35 above. It had an average rating of 5.9 out of 6 - nearly perfect. Testers commented that it was a very enjoyable part of the game. One tester mentioned that playing through the levels would not be enjoyable if they had to play without other people.

6.2.5 Overall Appearance Feedback

Testers were asked to rate the overall appearance of *SIGINT* on a scale of 1 (“Ugly”) to 6 (“Awesome”). The visual presentation of the game had an average rating of 5.1 out of 6. The individual ratings are in Figure 36.

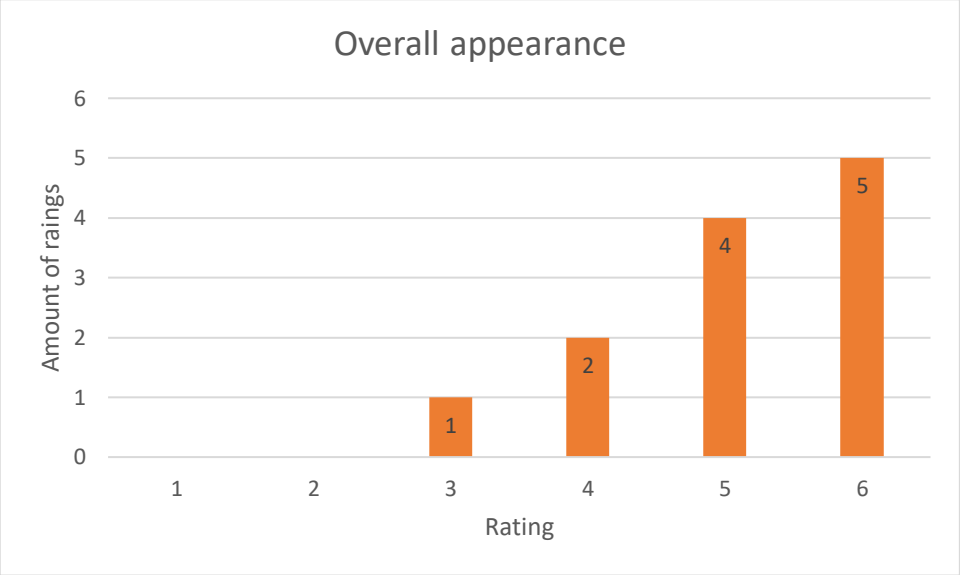

Figure 36. The ratings of the game’s visual presentation.

Testers felt that the user interface of *SIGINT* looked good, but the 3D levels need some additional work. A common issue that was brought up is that the projector is quite dim (even when the curtains in the CGVR Lab were closed), making it hard to distinguish objects in the scene. Another criticism was that the visuals of the game are quite utilitarian and lack an overarching theme. Despite this, the unique look of the characters in the game was praised. This means that the Gooch shading (see subchapter 4.4.2) used for the characters was a good choice.

6.2.6 Gameplay Smoothness Feedback

An important requirement of *SIGINT* established in the introduction was that the game needs to run smoothly on the Raspberry Pi. Thus, the testers were asked how smooth the gameplay experience felt to play on a scale of 1 (“The game was lagging/stuttering.”) to 6 (“No issues.”). The results are seen in Figure 37.

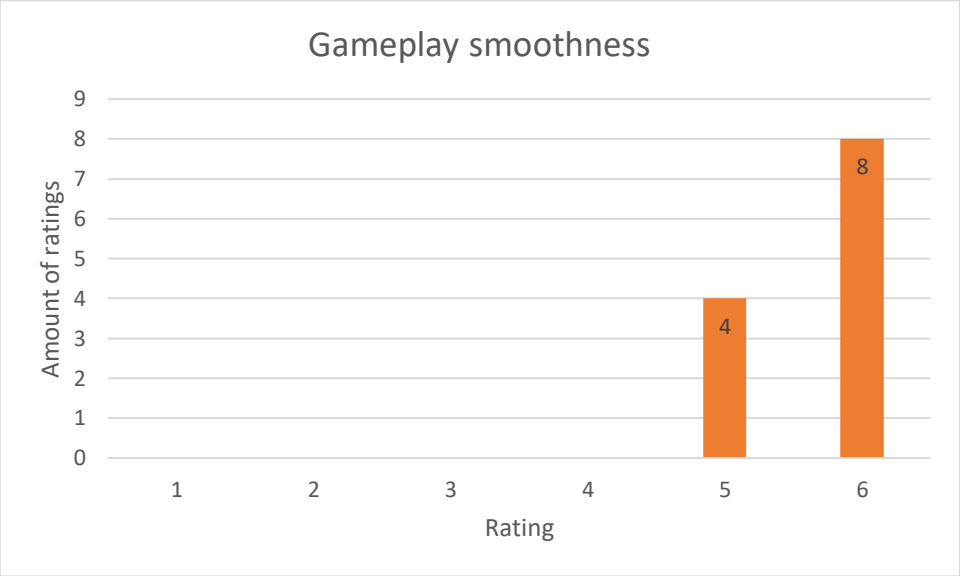

Figure 37. The ratings of how smooth *SIGINT* felt to play.

According to testers, there were no significant issues with this aspect of gameplay. The rating had an average score of 5.7 out of 6 – a very good score. This means that *SIGINT* runs smoothly on the Raspberry Pi, and the requirement was fulfilled successfully. The next subchapter showcases various improvements that were or could be made to *SIGINT* based on feedback from the testers.

6.3 Improvements

During playtesting with the first group, a critical issue was discovered. Upon playing the game again after finishing it once, players could skip from the first level to the last level by restarting the game upon playing the game again. This was caused by a bug in how new levels were loaded in *SIGINT*. The issue was fixed after the session.

One additional oversight was found by the first group of testers. The level with the pistons (Level 1) was accidentally set to appear as the last level instead of as the first level. Additionally, testers said that the pistons required too much waiting, as initially, there were

four pistons on each side of the level. The sequence of levels was fixed, and the number of pistons was also reduced from four to three to make the level easier.

When the four testers were trying to get the fastest completion time on the leaderboard, they expressed the desire to see the number of milliseconds in the leaderboard's elapsed time column. Before this, seconds were the smallest unit of time that was tracked. This suggestion was later implemented into *SIGINT* with single millisecond granularity. The following subchapter discusses aspects of *SIGINT* that could be improved in the future.

6.3.1 Future Improvements

The current version of *SIGINT* runs at a maximum of 30 FPS due to the performance constraints of the Raspberry Pi 4 and the Godot engine (see subchapter 2.3). Godot version 4.0 is planned to include official support for Linux ARM devices (like the Raspberry Pi)[9], as indicated by the Godot version 4.0 milestone[52]. When Godot version 4.0 is released (there is currently no specific release date), it would be beneficial to investigate if it offers better technical performance and stability so that the 30 FPS limit could be increased to 60 FPS or removed entirely.

As mentioned by one of the testers, the visuals of *SIGINT* do not have a clear, unified theme. This could be improved upon by changing the 3D models for the characters and objects in the levels. Additionally, the level environment meshes could use textures and normal maps that fit the chosen aesthetic of the game.

Another improvement to the game could be to implement level-specific leaderboards instead of the current system. This would allow players to compete for the best time in each level separately by finding new strategies to beat the level quickly.

Finally, the most impactful part of the game is the levels and the puzzles that they contain. Many testers mentioned that they would like to see more levels, especially ones with more cooperation elements. As such, designing new levels for the game that feature more cooperation should be a priority. Additionally, the levels should be a bit more difficult to solve than the current levels. This is to make sure that players do not get bored because the puzzles are too easy for them (as discussed in subchapter 6.2.3).

[52] https://github.com/godotengine/godot-proposals/milestone/1

7 Conclusion

As a result of this thesis, *SIGINT*, a top-down cooperative multiplayer video game, was developed for the Raspberry Pi 4. The game uses the CGVR Lab's vertical smart glass panels in its game design to create a unique experience for players. *SIGINT* was created using the Godot game engine, and other alternatives for development were analyzed as well. There were many technical challenges with developing *SIGINT* for the Raspberry Pi system, which were solved using various technologies.

Similar projects to *SIGINT* were analyzed in terms of both technical and design limitations for Raspberry Pi games. *SIGINT* was designed to be played by 2-4 people, and the choices behind the game design of *SIGINT* were explained. The core gameplay mechanic used in the game is cooperative puzzle-solving, and a leaderboard mechanic was created to satisfy more competitive players. Five levels were created for *SIGINT*, including a tutorial to teach players the mechanics used in the game. Each level in *SIGINT* contains a puzzle to be solved and designed to teach the players about the mechanics used in the game, such as *game overs*, cooperation, communication, and other mechanics needed to progress.

SIGINT was benchmarked, and the results proved that using VSync and limiting the maximum frame rate to 30 FPS was justified. The game was playtested by four groups of people with various habits of playing video games. Testers played through *SIGINT* and later filled an online form to give feedback. In general, the testers enjoyed their time with the game and praised the cooperation aspect of gameplay. Additionally, testers felt that the game ran smoothly on the Raspberry Pi. Feedback from the testers revealed that the visuals of *SIGINT* need to be improved and that there are not enough levels in the game. Some of the issues mentioned by testers were fixed, and some suggestions were implemented. There are more improvements to be made, but those are beyond the scope of the current thesis.

Special thanks go to all the testers who helped playtest *SIGINT* during the pandemic. Without their help, many bugs and design flaws would not have been found. Gratitude goes to the people behind the Computer Game Development and Design course and the Computer Graphics course, which gave me the basic knowledge of game engines that helped me learn how to use the Godot engine and implement *SIGINT*.

References

[1] Farouk Messaoudi, Adlen Ksentini, Gwendal Simon, Philippe Bertin. 2017. Performance Analysis of Game Engines on Mobile and Fixed Devices. ACM Transactions on Multimedia Computing, Communications, and Applications. https://doi.org/10.1145/3115934 (06.12.2020)

[2] Paul Bakaus. 2014. The Illusion of Motion. https://paulbakaus.com/tutorials/performance/the-illusion-of-motion/ (27.03.2021)

[3] Richard Leadbetter. 2014. The case for 30fps PC gaming. https://www.eurogamer.net/articles/digitalfoundry-2014-the-case-for-30fps-pc-gaming (27.03.2021)

[4] Jesse Schell. 2008. The Art of Game Design – A Book of Lenses. United States of America, Burlington: Morgan Kaufmann Publishers.

[5] Katie Salen, Eric Zimmerman. 2004. Rules of Play - Game Design Fundamentals. England, Massachusetts: The MIT Press Cambridge.

[6] Amy Gooch, Bruce Gooch, Peter Shirley, Elaine Cohen. 1998. A Non-Photorealistic Lighting Model For Automatic Technical Illustration. SIGGRAPH ’98. https://doi.org/10.1145/280814.280950 (25.04.2021)

[7] 2019. Frame Rate vs Frame Time https://cgvr.cs.ut.ee/wp/index.php/frame-rate-vs-frame-time/ (29.04.2021)

[8] Kate Moran. 2019. Usability Testing 101. https://www.nngroup.com/articles/usability-testing-101/ (29.04.2021)

Appendix

I. Glossary

VSync[53]	A graphics technology that synchronizes the frame rate of a game with the frame rate of the display.
Fragment shader[54]	The OpenGL pipeline stage after rasterization. The fragment shader outputs a single pixel into the framebuffer for each fragment generated by rasterization.
GLSL[55]	A C-style language for writing shaders.
Non-photorealistic rendering[56]	A computer graphics rendering technique, which focuses more on expressive styles, as opposed to traditionally photorealistic graphics.
Field of view[57]	In video games, FOV usually refers to the horizontal or vertical angle of an in-game camera's view.

[53] https://www.digitaltrends.com/computing/what-is-vsync/
[54] https://www.khronos.org/opengl/wiki/Fragment_Shader
[55] https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)
[56] https://en.wikipedia.org/wiki/Non-photorealistic_rendering
[57] https://en.wikipedia.org/wiki/Field_of_view_in_video_games

II. Accompanying Files

The accompanying files archive has the following structure:

- **/build** – the folder that contains the Raspberry Pi and Windows builds of *SIGINT*.
- **/source** – the folder that contains the Godot project files of *SIGINT*.
- **/performance_testing** – the folder that contains the performance testing results.
- **/usability_testing** – the folder that contains the feedback form and the answers of the testers.

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Mattias Aksli**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

***SIGINT* – A Cooperative Puzzle Game on Vertical Wall Panels**,

supervised by **Raimond-Hendrik Tunnel**.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mattias Aksli

03/05/2021