CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Individual game development with open-source software

A case study with guidelines for programmers and game designers

Master's Thesis in Computer Science and Engineering

Axel Ljungdahl

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Master's thesis 2020

Individual game development with open-source software

A case study with guidelines for programmers and game designers

Axel Ljungdahl

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Interaction Design
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Individual game development with open-source software
A case study with guidelines for programmers and game designers
Axel Ljungdahl

Supervisor: Michael Heron, Department of Computer Science and Engineering
Examiner: Staffan Björk, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Division of Interaction Design
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Individual game development with open-source software
A case study with guidelines for programmers and game designers
Axel Ljungdahl
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The focus of this project is individual (i.e., single-person) game development using open-source tools; specifically, the development of a side-scrolling, 2D platform game with a focus on players' enjoyment, and the creation of this thesis, which is intended to serve as a case study of individual, open-source game development and a limited guide for future developers, especially those working alone.

This report describes the development process for the aforementioned game (titled *CheckerSphere*), which includes both single-person game development and development using strictly open-source tools (aside from low-level proprietary software such as drivers for the graphics card used during development as well as proprietary hardware). Additionally, as a guide, this thesis contains items such as a list of personally recommended study material, mostly consisting of recorded lectures and video essays; a brief analysis of the movement systems of three 2D platformers; information regarding how to legally use existing assets and where to find them; in what ways *CheckerSphere* is designed for accessibility; and several lists of open-source tools that may be of use for a game developer.

Open-source distribution of games may not always be considered feasible, but it brings new possibilities, such as potentially causing volunteers to contribute to the game for free, encouraging players to create problem descriptions when they find apparent bugs, allowing those who are unwilling to install closed-source software on their computers to play the game, increasing the amount of trust and good will towards the developers, and helping future game development projects by contributing to the open-source community. Additionally, using only open-source tools brings benefits such as being able to trust software to not infringe on one's privacy or be otherwise malicious; being able to suggest, or personally implement, desired changes in a piece of software; being able to remove undesired parts of software; and having access to all software without payment, facilitating lower-budget development.

This project has personally strengthened the view that open-source game development is entirely feasible when extreme graphical fidelity is not required, both in terms of using only open-source tools and in terms of distributing the game under an open-source license; the latter does not prevent the developer from releasing the game commercially, since non-code assets may remain proprietary.

Keywords: interaction design, video game design, independent video game development, open-source software

Acknowledgements

I would like to thank my supervisor, Michael Heron, for his support throughout this project; my examiner, Staffan Björk; the three individuals who helped improve the game by testing it at various stages of development; the opponent for my thesis; and everyone who downloaded the game from `itch.io` and provided feedback.

Axel Ljungdahl, Gothenburg, June 2020

Contents

List of Figures

1 Introduction

1.1 Background

1.1.1 Video games

Video games[1], also known as *electronic games* and *computer games*, are interactive games operated by computer circuitry. The first video game, *Spacewar!*, was created in 1962 at the Massachusetts Institute of Technology; since then, video games have become a part of mainstream culture and commerce. [1]

Unlike film and literature, which are organized into genres based on their subject matter, genres for games are determined by gameplay – e.g., *sports* games, *shooter* games, *racing* games and *role-playing* games. Genres such as these can be further divided by subtype – e.g., sports games can be further divided into categories based on the type of sport being represented. However, an arbitrarily selected game cannot necessarily fit into a single game genre – for example, modern role-playing games tend to include challenges that test physical coordination, which was not initially a common aspect of the genre. [2]

Another type of genre is the *platform game* (or *platformer*), which consists of games for which gameplay is highly based on players controlling a character who can manually move and jump between platforms [3]. This project involves the development of such a game.

1.1.2 Platform games

Depending on the definition, the first platform game was either *Space Panic*, released in 1980, or *Donkey Kong*, released in 1981; the latter was the game which introduced a button dedicated to jumping, which would become a staple of the genre. *Super Mario Bros.*, released in 1985, is a platformer game which became highly influential for later platformers. The genre suffered a drop in popularity starting in the late 1990s, but has since grown. [3]

Two-dimensional[2] platformers (i.e., platformers in which movement is limited to a two-dimensional axis, but where visuals may be rendered as projections of three-dimensional objects) can be divided into different types based on a variety of criteria. For example, the following criteria significantly impact gameplay and their

[1] *Video game* is abbreviated to *game* in the remaining paragraphs.

[2] Abbreviated to *2D* in the remaining paragraphs; similarly, *three-dimensional* is abbreviated to *3D*.

implementation may therefore be subjectively experienced as positive or negative depending on the individual:

- The movement of the *camera* (i.e., the region of the game's world which is visible to the player, projected to two dimensions). For example, *Donkey Kong*[1] uses a static camera; *Elevator Action*[2] uses a camera which only moves vertically; *Super Mario Bros.*[3] uses a camera which predominantly moves horizontally; and *Sonic the Hedgehog*[4] uses a camera which frequently moves both vertically and horizontally.
- The perspective of the camera. Some games, such as *Super Mario Bros.*, position the camera so that the player and other entities are shown approximately in profile; others, such as *Kirby 64: The Crystal Shards*[5] use a slightly rotated perspective; and yet others, such as *Sonic 3D Blast*[6] use an isometric or otherwise pseudo-3D perspective without letting the player manually rotate the camera. This attribute does not necessarily determine whether the rendering is based on 2D images or 3D models – *Sonic 3D Blast* is a pseudo-3D game with 2D artwork, but *Kirby 64: The Crystal Shards* is a side-scrolling game which uses 3D models.
- The movement system. For example, *Super Mario Bros.* uses slight acceleration and deceleration when moving, responds almost immediately to the player's movement and allows them to adjust jump height and change direction while in the air, whereas *Castlevania*[7] has constant velocity and a fixed jump arc which cannot be adjusted or interrupted in any way aside from collision with another entity.

3D platformers can be divided differently (e.g., some use a first-person camera while others show the player's character from a third-person perspective). Additionally, some platformers use primarily 2D movement but also allow some movement along a third axis, such as *Mutant Mudds Deluxe*[8]; depending on the exact definition of the term, these may also be considered 2D platformers, despite not having strictly 2D movement.

Examples of 2D platform games released between 2017 and 2019 include *Super Mario Maker 2* [9], *Sonic Mania*[10], *Shovel Knight: Specter of Torment*[11], *Celeste* [12] and *Cuphead* [13].

[1]https://nintendo.com/games/detail/arcade-archives-donkey-kong-switch/
[2]https://arcade-museum.com/game_detail.php?game_id=7700
[3]https://nintendo.com/games/detail/super-mario-bros-3ds/
[4]https://sega.com/games/sonic-hedgehog
[5]https://nintendo.com/games/detail/kirby-64-the-crystal-shards-wii-u/
[6]https://store.steampowered.com/app/34278/Sonic_3D_Blast/
[7]https://nintendo.com/games/detail/castlevania-3ds/
[8]https://store.steampowered.com/app/247370/Mutant_Mudds_Deluxe/
[9]https://nintendo.com/games/detail/super-mario-maker-2-switch/
[10]https://sega.com/games/sonicmania
[11]https://yachtclubgames.com/shovel-knight/
[12]http://celestegame.com/
[13]https://store.steampowered.com/app/268910/Cuphead

1.1.3 Game development

Game development is unlike many other types of software engineering due to its focus on player engagement over code quality and stability – e.g., constant stability is not necessarily important as long as no significant amount of the player's progress is lost when an error occurs. Certain software bugs may even be remembered fondly – for example, a bug present in *Super Mario Bros.* which leads the player to a level unofficially called the *Minus World*[1] – and, occasionally, games such as *Goat Simulator*[2] attempt to recreate such experiences as a core aspect of their gameplay.

The creation of a game requires that several different roles are filled. Every aspect of a game (e.g., how the player can manipulate the game and how the game presents itself to the player) must be designed, regardless of whether a rigorous design document is followed, design is based on rapid iteration, or some other method is used. A game must be implemented and, generally, visuals and audio must be created and/or acquired. Finally, the game must be distributed, and, if the game is sold, various business-related problems (e.g., marketing, securing a channel for distribution and handling taxes) must be dealt with.

The barrier to entry for game development is continuously lowering due to the continued creation and refinement of general-purpose and genre-focused game engines, which results in several tasks which were previously required now being optional in certain cases – for example, by using a genre-focused game engine, depending on the game being developed, developers may be able to completely avoid programming or exchange traditional programming for simple text-based or visual scripting), and online stores such as *Steam*[3] and *itch.io*[4] can handle distribution, keeping records of transactions and other business-related issues. However, *design* of games remains highly necessary; while some aspects are more difficult (e.g., previously impressive technology may no longer be enough to convince people to play a game with mediocre design), others are less so (e.g., the number of freely available resources, ranging from vague guidelines to well-defined design templates, are continuously increasing and can be acquired through the Internet).

1.1.4 Individual game development

The larger and more complex a game is, the more people and types of skill are needed in its development team. The capacity for game development is proportional to these two factors, which means that smaller teams may be unable to create what larger teams can, due to a need for additional team members (or more time than what can feasibly be allocated), a need for particular skills or some other requirement.

However, smaller teams have certain advantages. Due to the number of different roles involved in a large project, in order to prevent inaccurate representation of one's thoughts – or misunderstanding of another's – clear communication between group members is required. For example, game designers' tasks are intertwined with each of the other roles – they must understand the limitations of programmers

[1] An unofficial description of the glitch is available at `https://mariowiki.com/Minus_World`.
[2] `http://goat-simulator.com/`
[3] `https://store.steampowered.com/`
[4] `https://itch.io/`

and the programming environment in order to design a game which can be feasibly implemented, they must ensure that the visuals and audio created for a game fits with its gameplay and narrative, they should consider any business-related information that may be relevant to the project (e.g., which features appeal to the project's demographic), and they must communicate with any other game designers working on related issues. These problems also exist in teams of a few people, but the risk increases with the number of people and the number of different roles.

Individual game development (i.e., development performed by a single person) leads to a unique situation in which the game designer, programmer, graphical artist, composer, etc. are the same person, which means that problems with communication cease to exist; additionally, a single person can be more willing to take financial risks since failure only affects them and anyone who depends on them. However, it also requires that the single person in charge possesses each of the skills required to create a complete game and that this individual is able to remain motivated enough to finish the project since, unlike in a team setting, if the sole developer loses motivation, there are no coworkers who can motivate them to continue. As a result of this, the choice of budget and team size is strongly dependent on the scope and risk factor of the game, and there are advantages and disadvantages inherent to each type of development.

An individual developer with few skills relevant to game development other than design and programming may be tempted to gather freely available assets and use them to create a game. However, using nothing but existing assets, whether they are freely usable or must be purchased, may cause a game to be seen as an *asset flip* – a game created through its developers acquiring assets and packaging them as a new game, after which the game is sold for a low price in order for the developers to be able to recover their costs through a small number of sales [4]. As shown by the accusations of being an asset flip made against *PlayerUnknown's Battlegrounds*, despite its creative director claiming that most of the game's assets were created by its developers [5], developers who intend to sell their games may need to be careful when selecting where to use existing assets in their games. Due to these potential problems, single-person game development for commercial purposes may seem to be a very difficult challenge if the goal is to produce a game of high quality in terms of design, implementation and aesthetics, unless the developer in question is a person who can not only design and program games, but is also able to create aesthetically pleasing graphics and audio.

Despite these difficulties, examples of commercially released games developed by a single person do exist, such as the following:

- *Axiom Verge*, a 2D game inspired by games such as *Metroid.* [6][7]
- The first commercially released version of *Stardew Valley*, a 2D role-playing game based on farming. [8]
- The first commercially released version of *Minecraft*, a 3D game involving "placing blocks and going on adventures". [9][10]

1.1.5 Open-source development

The *Open Source Initiative* defines open-source licenses as licenses which "allow software to be freely used, modified, and shared" [11][1].

Open-source licenses are primarily of two different types – *copyleft* or *permissive*. The former require that all derivative works are released under the same license as the original work; an example of such a license is the *GNU General Public License 3.0*[2]. The latter is any license which is not *copyleft* – i.e., derivative works may be released under any license, even a proprietary one, potentially with non-restrictive requirements such as attribution. [12]

Some open-source games also release their assets for free, such as *0 A.D.*[3], *The Battle for Wesnoth*[4], *NetHack*[5], *SuperTuxKart*[6] and *Xonotic*[7]. Others use proprietary assets, or assets available under non-commercial licenses, but release the code under an open-source license, such as *Angband*[8], *Frogatto & Friends*[9], *OpenTTD*[10], *osu!*[11], *Tales of Maj'Eyal*[12], *Warsow*[13]. Some games were originally released as proprietary, commercial games but were eventually released under an open-source license, with or without proprietary assets, such as *Doom*[14], the *Marathon* trilogy[15], *Star Wars: Jedi Knight - Jedi Academy*[16] and *Quake*[17].

1.2 Thesis

1.2.1 Premise

The focus of this project is individual (i.e., single-person) game development using open-source tools; specifically, the development of *CheckerSphere*, a side-scrolling, 2D platform game with a focus on players' enjoyment (rather than an engaging narrative, educating players or some alternative goal). The result of the project was intended to be the developed game (in the form of a short prototype of subjectively high quality) as well as this thesis, which is intended to serve as a case study of individual, open-source game development as well as a limited guide for future developers, especially those working alone. As a guide, this thesis was intended to contain items

[1]A more detailed definition is available at `https://opensource.org/osd`.
[2]`https://gnu.org/licenses/gpl-3.0.html`
[3]`https://play0ad.com/`
[4]`https://wesnoth.org/`
[5]`https://nethack.org/`
[6]`https://supertuxkart.net/Main_Page`
[7]`https://xonotic.org/`
[8]`https://rephial.org/`
[9]`https://frogatto.com/`
[10]`https://openttd.org/`
[11]`https://osu.ppy.sh/home`
[12]`https://te4.org/`
[13]Available at `https://warsow.net/`. *Warsow* also has a completely non-proprietary fork called *Warfork* (`https://warfork.com/`).
[14]`https://github.com/id-Software/DOOM`
[15]`https://alephone.lhowon.org/`
[16]`https://github.com/grayj/Jedi-Academy`
[17]`https://github.com/id-Software/Quake`

such as examples of advice from previous game developers and personal experience; examples of open-source tools and how they can be utilized to create games without a budget; examples of ways to acquire free assets for game development; and reasons to use open-source tools and release one's code under an open-source license.

The research questions associated with the project are the following:

As an individual developer using strictly open-source tools[1]*, what problems may appear – particularly in terms of game design, implementation, creation of visuals and creation of audio – during development of a side-scrolling, 2D platform game, and how can these problems be solved?*

While these questions are limited to a specific type of game, their answers are intended to be potentially useful to those interested in developing other types of games.

The demographic of the produced game, to which all testers would belong, was intended to consist of those who have some interest in playing games, allowing for varying preference with regard to difficulty; this demographic is wide, so the game must be simple to understand for people with little experience with games, but those who have never played a game and those who are disinterested in games are excluded, since the former are unlikely to choose a small independent game as their first experience with the medium and the latter seem unlikely to want to play any game whatsoever.

1.2.2 Goals

This project has three goals – two personal goals for the prototype, and one goal for the thesis. The personal goals include creating a short, high-quality game to serve as a demonstration of the skills acquired during prior studies, and showing general self-improvement in terms of game design and programming. The goal for the thesis is to create a set of guidelines for future developers working with an open-source toolset, focusing on those working individually, by providing a summary of the knowledge gained during the project; this may include examples of open-source tools, ways to create a visually appealing game with minimal artistic skills, game design techniques, reflections on how to improve a development process, etc.

1.2.3 Limitations

Limitations for the thesis include the following:

- A large percentage of the time was spent on development rather than research; a more comprehensive thesis could have been produced by a mainly research-focused project.
- The focus was mainly placed on tools and techniques; issues such as mental health, marketing and distribution were only briefly mentioned.
- Only one game engine was tested due to the amount of time seemingly required

[1]Some exceptions may need to be made to software which may be classified as tools but for which no feasible alternatives exist, such as drivers or firmware.

to learn and prototype using a new one and the lack of an obvious reason, in terms of providing value for this thesis, to do so.

Limitations for the prototype include the following:

- An existing game engine was used in order to avoid the time needed to create a custom engine (which is not directly related to the purpose of this project).
- The game's demographic excludes those who have never played a game or are not interested in playing games which are focused on providing enjoyment, in order to save the time required to accommodate such people.
- The support for non-keyboard controllers is very basic in order to avoid spending time on implementing prompts for the buttons on all major types of controller.
- Freely available graphical assets were used in parts of the game in place of graphics made specifically for the game; the same applies to sound effects.
- Freely available music was used; creation of custom music was planned, but higher priority was placed on visual fidelity and sound effects.
- The movement system and the level design were prioritized over visuals and audio, and the prototype can be finished in approximately one minute by an experienced player; the game was never intended to be appropriate for commercial release.
- Even if the code is deterministic, automated testing was not implemented.

2 Theory

2.1 Game development

To study general game design, a variety of material is available, such as lectures, interviews with developers, video essays, articles and developers' logs, but most appears to be *gray literature* – "research that is either unpublished or has been published in some non-commercial form" [13]. This is a result of factors such as the partially subjective nature of game design – an attribute which is present in anything that relies on a person's subjective interpretation of a creation – and the fact that the profession of game designer is relatively recent when compared to those present in other types of media (e.g., film).

Printed books which focus on game design do exist, such as Steve Swink's *Game Feel: A Game Designer's Guide to Virtual Sensation*[1], but the high-level research used to gather sources for this project – which generally consisted of web searches using appropriate search terms, including names of individual developers and keywords such as *game design* and *game development* – suggests that the majority of material is gray literature; however, disproving the existence of large amounts of non-gray literature within this area is outside of the scope of this project. Major primary sources of material related to game development include the *Game Developers Conference*[2], *Gamasutra*[3] and others; secondary sources include community-driven repositories of knowledge such as the `gamedev` community on *Reddit*[4].

With regards to implementation and asset creation, resources such as tools' documentation, tutorials and existing algorithms can be used. For information regarding other aspects of game development, such as dealing with mental health problems and how to start working as an independent game developer, lectures from the aforementioned *Game Developers Conference* may be helpful (e.g., *Mental Health and Making It: Succeeding Through the Struggles*[5]) but, as previously mentioned, subjects such as these are beyond the scope of this project.

2.1.1 Individual game development

Research on the differences between individual game development and game development in small teams is apparently lacking; the bulk of the knowledge seems to be

[1]`http://game-feel.com/`
[2]`https://gdconf.com/`
[3]`https://gamasutra.com/`
[4]`https://reddit.com/r/gamedev/`
[5]`https://gdcvault.com/play/1024979/Mental-Health-and-Making-It`

available through informal sources such as logs, lectures and the like from developers.

Using search terms such as *solo game development*, *individual game development* and *one-person game development* results in lists of commercially successful games developed by individuals; using such developers' names as search terms provides some sources of information specific to individual game development (e.g., in an interview, one developer revealed that he temporarily avoided loss of motivation by switching between different roles [14]), but most of the information appears to also apply to smaller teams (e.g., advice regarding how to effectively market a game with a minuscule budget).

Easily accessible sources (at least those found during this project) such as lectures and developers' *YouTube* channels do not seem to frequently discuss the differences between one- and two-person development – a counterexample is the lecture *No Time, No Budget, No Problem: Finishing 'The First Tree'*[1] by developer David Wehle, and various videos by game developer Thomas Brush[2]. Developers' *text-based* logs are potential sources of such information, but the logs found were highly unstructured – for example, mixing thoughts about everyday life with press releases and occasional mentions of their development process, as with the logs for *Axiom Verge*[3]. It is entirely possible that some potentially useful lectures, excerpts of developers' logs or the like have been missed due to faulty assumptions and poor decisions during the research for this project, but finding these hypothetical sources would require a project with a greater scope or different goals.

Auteur theory, in filmmaking, refers to the notion that a film's director is a major creative force [15], but a similar idea is also present in the video game industry. Examples of auteurs may include people such as *Hideo Kojima*, who worked on the *Metal Gear Solid* franchise, and *Hidetaka Miyazaki*, who is part of the team of developers responsible for the *Souls* franchise; however, it is not always apparent if these individuals truly have the highest position of creative input in their respective teams, or if the team simply produce games that are similar to their previous titles to improve the odds of success [16]. The structure of a development team determines how ideas are presented – any team of at least two people cannot have an auteur if the potential auteur is not able to relay their ideas. Depending on the definition of the term "auteur theory", a developer working alone will either be the auteur of their project regardless of their intention, or cannot be an auteur because they have no team members who can implement their creative vision – regardless, their games may share elements in terms of gameplay and/or aesthetics because they were wholly designed and implemented by the same individual. An example of research into auteur theory for games is [17].

2.1.2 Open-source game development

As mentioned in chapter 1, open-source software is licensed under conditions which allow the software to be "freely used, modified, and shared", according to the *Open*

[1]`https://youtube.com/watch?v=g5f7yixtQPc`

[2]`https://youtube.com/user/thomasmbrush`

[3]The blog in question is available at `https://axiomverge.com/blog`, and information about *Axiom Verge* is available on the same website.

Source Initiative [11]. A list of various open-source licenses can be found on their website.

Source-available is a term which refers to any code for which source code is publicly available, including open-source software, but the term may also include software which imposes restrictions which conflict with the above definition of open-source software, such as prohibiting commercial use or prohibiting any redistribution whatsoever. As an example of source-available licensing, the *Commons Clause* is a legal clause which can be appended to a permissive open-source license in order to restrict commercial use without substantial addition to the original work [18]. Examples of non-open-source, source-available games include *Airline Tycoon Deluxe*[1], *Aliens versus Predator Gold*[2], *Anodyne*[3] and *Receiver*[4].

Assets which are not code, such as 3D models and music, generally do not use the same licenses as those used by code. Examples of licenses for such assets include the *Creative Commons*[5] licenses for general assets, the *SIL Open Font License*[6] for fonts and the *GNU Free Documentation License*[7] for documentation. As with open-source licenses, these licenses may be permissive (e.g., *CC BY 4.0*[8]) or copyleft (e.g., *CC BY-SA 4.0*[9]), and they may have additional restrictions such as prohibiting commercial use (e.g., *CC BY-NC 4.0*[10]).

When accepting third-party contributions to an open-source project, in order to prevent issues with the intellectual property rights of individual contributors, a *Contributor License Agreement* (*CLA*) – a legal agreement which states that the contributor gives the project's maintainers an irrevocable license to distribute the contributed work as part of the project, or that the contributor completely assigns the copyright of the contributed work to the project's owner – may be of use. A CLA must be clear with regard to what rights are granted – for example, whether the project's license can be changed without the consent of contributors. [19] An example of a CLA is *Apache*'s *Individual Contributor License Agreement*[11].

Examples of existing research into open-source game development include [20] and [21].

2.1.3 Accessibility

In order to accommodate potential players with disabilities, it may be in a developer's best interest to attempt to add certain accessibility-related features, especially those which can be implemented relatively easily and quickly, such as those listed in the *basic* set of guidelines from `http://gameaccessibilityguidelines.com/`. These

[1]`https://gog.com/game/airline_tycoon_deluxe`
[2]`https://gamefront.com/games/aliens-vs-predator-3/file/avp-gold-complete-source-code`
[3]`https://github.com/analgesicproductions/Anodyne-1-Repo`
[4]`https://github.com/David20321/7dfps`
[5]`https://creativecommons.org/`
[6]`https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL%2F`
[7]`https://gnu.org/licenses/fdl-1.3.en.html`
[8]`https://creativecommons.org/licenses/by/4.0/`
[9]`https://creativecommons.org/licenses/by-sa/4.0/`
[10]`https://creativecommons.org/licenses/by-nc/4.0/`
[11]`https://apache.org/licenses/icla.pdf`

basic guidelines include, among others, allowing remapping of controls; having a clearly readable user interface with large fonts; using clearly contrasting colors for separate entities; not including flickering images or repetitive patterns; having modifiable audio volume; displaying subtitles for all important speech; and providing multiple levels of difficulty [22].

Depending on the demographic and the available resources, it may be financially beneficial to implement items from the *intermediate* list – e.g., supporting multiple input devices, allowing customization of interfaces, allowing players to skip nonessential parts of the game and allowing players to adjust the field of view in a 3D game [23] – and/or from the *advanced* list – e.g., providing a control scheme which is compatible with assistive devices such as eye tracking, allowing all instructions and narrative to be replayed, allowing font size to be adjusted, and including every relevant category of functional impairment during playtests [24]. While some instructions on each list are likely irrelevant for certain types of games (e.g., there's no need to provide subtitles for a game without speech or any other essential sounds), they nevertheless contain generally useful advice for development teams of any size.

2.2 Open-source tools

There exist many open-source tools which can be used for game development as feasible alternatives to closed-source tools. These tools enable the creation of video games while using a minimal amount of proprietary software; however, depending on the hardware required to develop the game, closed-source firmware or drivers may be required (e.g., in order to use a graphics card to its full potential), and closed-source websites are difficult to avoid when marketing and/or distributing a game.

As mentioned previously, some may choose to use strictly open-source software, or possibly any source-available software whose license has restrictions which are acceptable to the user. Reasons for using open-source software may include being able to trust software to not infringe on one's privacy or be otherwise malicious; having software repositories, such as those for the operating systems *Debian GNU/Linux*[1] or *Arch Linux*[2], which is only legal since the code within can be freely redistributed; being able to suggest, or personally implement, desired changes in a piece of software; being able to remove undesired parts of software through recompilation (and possibly without changing the source code, if the software in question is designed to be able to be compiled without particular features); having access to all software without payment (although donations may be requested by a developer in order for them to be able to continue supporting a project); being able to select an operating system that can run on old hardware with low specifications; and not being forced to read (or ignore) proprietary usage agreements which may contain hidden, undesired restrictions or requirements in countries where doing so is legally valid.

[1]`https://debian.org/distrib/packages`
[2]`https://archlinux.org/packages/`

2.2.1 Operating systems

There appear to be two groups of open-source operating systems with which modern games can be efficiently developed and played. The first group consists of the open-source operating systems based on the *Linux kernel*[1], also known as *Linux distributions* [25]. The second group consists of the open-source derivatives of the *Berkeley Software Distribution*, an operating system released in 1978 [26], which includes, among others, *FreeBSD*[2] and *DragonflyBSD*[3]. These groups seem to encompass the most capable open-source operating systems with regard to drivers for graphics cards and software for interacting with graphics cards (e.g., *Mesa 3D*[4]), which is essential for performance-heavy games. All of the software listed above can be run using Linux distributions, and some can run using BSD derivatives. Additionally, *Wine* – a compatibility layer used to run Windows applications on, among other operating systems, Linux distributions and BSD derivatives [27] – allows consumers who typically play games on closed-source operating systems such as *Windows 10*[5] to gradually switch to open-source platforms; the software which requires Wine to run may generally be closed-source, since it could otherwise have been ported to avoid the use of Wine (although some exceptions exist for which ports do not yet exist due to, for example, a lack of resources), but *Wine* reduces the amount of closed-source software which has to be used to play many commercially available games.

ReactOS[6] is an open-source, Windows-like operating system which is designed to be able to run Windows software, but it is still in an early stage (version 0.4.12 [28]) and is currently "mostly intended for programmers to expand and improve on" [29].

2.2.2 Game engines, frameworks and libraries

Development frameworks include general-purpose engines such as *Godot*[7] and *LÖVE*[8]; genre-specific engines such as *Ren'Py*[9] and *Adventure Game Studio*[10]; and lower-level development libraries such as *SDL*[11] and *SFML*[12]. Game engines may have corresponding tools meant to be used specifically with that engine, as is the case with *Godot*, which includes a code editor, a level editor (i.e., an editor used to create and modify in-game levels), an animation editor, and various other features [30].

[1]`https://github.com/torvalds/linux`
[2]`https://freebsd.org/`
[3]`https://dragonflybsd.org/`
[4]`https://mesa3d.org/`
[5]`https://microsoft.com/en-us/windows/get-windows-10`
[6]`https://reactos.org/`
[7]`https://godotengine.org/`
[8]`https://love2d.org/`
[9]`https://renpy.org/`
[10]`https://adventuregamestudio.co.uk/`
[11]`https://libsdl.org/`
[12]`https://sfml-dev.org/`

2.2.3 Tools for creating and editing visuals

2D graphics can be created directly as *raster graphics*, storing image data as a grid of rectangular pixels [31]; as *vector graphics*, storing image data as mathematical formulae which can be scaled without losing detail [32]; or as 3D models which can be used to generate raster graphics.

Examples of tools include *Krita*[1] and the *GNU Image Manipulation Program*[2] (*GIMP*) for raster graphics; *Inkscape*[3], *LibreOffice Draw*[4] and, to a lesser extent, *GIMP*, for vector graphics; and *Blender*[5], *Dust3D*[6] and *FreeCAD*[7] for 3D modelling.

2.2.4 Tools for creating and editing audio

Examples of tools to create music include *LMMS*[8], *Ardour*[9], *Radium*[10], *MusE*[11], *Qtractor*[12], *Bosca Ceoil*[13], *SoundTracker*[14] and *OpenMPT*[15]. These tools vary with regard to features but can all be used to create music; *Audacity*[16] is an example of a tool which is focused on *editing* audio rather than creating it, but it is technically able to create new audio by manipulating existing data.

Bfxr is a tool which can be used to rapidly create sound effects for computer games. However, it appears to be unable to run without either the closed-source *Adobe Flash Player*[17] or a closed-source operating system. [33] To circumvent this, *Jfxr*[18], which was inspired by Bfxr [34], can be used as a replacement. Other alternatives include the browser-based tool *BeepBox*[19] and the aforementioned *Audacity*, as well as more specialized tools such as *Power Station Industrializer*[20] and *Explodomatica*[21].

2.2.5 Coding tools

Integrated development environments – tools which, in addition to allowing users to edit code, typically integrate features such as debugging, file browsing and version

[1]`https://krita.org/`
[2]`https://gimp.org/`
[3]`https://inkscape.org/`
[4]`https://libreoffice.org/discover/draw/`
[5]`https://blender.org`
[6]`https://dust3d.org/`
[7]`https://freecadweb.org/`
[8]`https://lmms.io/`
[9]`https://ardour.org/`
[10]`https://users.notam02.no/~kjetism/radium/`
[11]`https://muse-sequencer.github.io/`
[12]`http://qtractor.org/`
[13]Available at `https://boscaceoil.net`. It is open-source, but it requires the *Adobe Integrated Runtime* (`https://get.adobe.com/air/`), which is not open-source.
[14]`http://soundtracker.org/`
[15]Available at `https://github.com/OpenMPT/openmpt`. It needs Wine to run without Windows.
[16]`https://audacityteam.org/`
[17]`https://get.adobe.com/flashplayer/`
[18]`https://jfxr.frozenfractal.com/`
[19]`https://github.com/johnnesky/beepbox`
[20]`https://sourceforge.net/projects/industrializer/`
[21]`https://github.com/smcameron/explodomatica`

control [35] – include those created for use with a specific game engine (e.g., the editor included with the *Godot* engine[1]), *Eclipse*[2], *NetBeans*[3], *Code::Blocks*[4] and others.

Examples of less feature-packed *code editors*, which may be able to be extended to gain the functionality of an integrated development environment, include, among others, *Visual Studio Code*[5], *Atom*[6], *Brackets*[7], *Vim*[8] and *Emacs*[9].

2.2.6 Level editors

As mentioned above, level editors may be included in a game engine, but there exist external tools for 2D – e.g., *Tiled*[10] and *Ogmo Editor 3*[11] – and 3D – e.g., *GtkRadiant*[12], *TrenchBroom*[13], the *ATF LevelEditor*[14] and the aforementioned *Blender*.

[1]`https://godotengine.org/features`
[2]`https://eclipse.org/`
[3]`https://netbeans.org/`
[4]`https://codeblocks.org/`
[5]`https://code.visualstudio.com/`
[6]`https://atom.io/`
[7]`http://brackets.io/`
[8]`https://vim.org/`
[9]`https://gnu.org/software/emacs/`
[10]`https://github.com/bjorn/tiled/`
[11]`https://github.com/ogmo-editor-3`
[12]`https://github.com/TTimo/GtkRadiant`
[13]`https://github.com/kduske/TrenchBroom/`
[14]`https://github.com/SonyWWS/LevelEditor`

3 Methodology

3.1 Software development

Due to the user-dependent nature of game design, some models of software development are less feasible than they may be for other types of software (e.g., productivity tools). For example, the *waterfall* model consists of a linear sequence of steps (requirements, design, implementation, verification and maintenance) carried out in a single iteration [36]; as a result, if a flaw is found late enough in the process, it cannot necessarily be resolved without deviating from the model. On the other hand, *iterative* models consists of several iterations, where each iteration adds new functionality [36]; this results in the ability to solve problems that appear in one iteration during the subsequent iteration. Additionally, iterative models allow frequent testing with potential users to be integrated into the development process.

The *agile* methods of development consist of a subset of the *iterative* methods. An example of an agile method is *Scrum*, which is based on a series of short iterations (generally between one and six weeks) called *sprints*, each of which is executed based on a *sprint backlog* (a list of tasks meant to be finished within the sprint) with the goal of producing a deliverable piece of software; at the end of each sprint, feedback is gathered from stakeholders in order to determine how the product can be improved during the next sprint. [36]

3.2 General interaction design

A/B testing involves comparison between "two versions of the same design" to see which performs better, statistically, with a random group of participants [37].

Competitive testing involves analyzing competitors' product(s) from the point of view of a user, and using the gathered information to evaluate one's own product(s) [37].

The *Critical Incident Technique* consists of finding *critical incidents* – important moments where a choice is made while using a product, leading to a positive or negative outcome – by retroactively asking test participants to describe a situation that ended well or poorly. This method is intended to provide quantitative data and therefore needs a significant number of participants (e.g., fifty to one hundred). [37]

Crowdsourcing occurs when a large group of people help with a project by completing various tasks voluntarily in exchange for some form of compensation [37].

Evaluative research, also known as *user testing* or *product testing*, is a method in which a prototype is tested by real users in an attempt to gauge expectations against the prototype being tested. Ideally, the process is iterative, with each iteration being based on the tests performed on the previous iteration; exclusively evaluating very late in the design process, where changes may be complicated and expensive, does not fit with this method. [37]

Experiments are used to measure the effect of an action by showing a causal relationship between the action and an event, or by clearly demonstrating that the action directly resulted in a particular event [37]. When testing video games, experiments can be performed in order to determine what caused, for example, a loss of interest in a particular section of a game or a sudden spike in difficulty.

There are multiple types of *observation* which can be used during tests. For example, they may or may not involve communication with a participant; may be one-time occurrences or occur repeatedly (regularly or irregularly); and may be formal or informal. [37]

Participatory design involves active engagement from users and potential stakeholders throughout the research and design processes, including co-design activities which are ideally performed face-to-face [37].

Prototyping consists of creating versions of a product with varying levels of fidelity – for example, for a software interface design, an low-fidelity prototype may be a *paper prototype*, with pages representing different screens, whereas a high-fidelity prototype may be a digital, interactive version of the intended interface. *Experience prototyping*, unlike passive viewing of static prototypes, is based on active interaction with a low-fidelity prototype, either internally or with potential users. [37]

Rapid Iterative Testing and Evaluation (*RITE*) is a method used early in a design process to evaluate and identify problems with an interface, rapidly resolve them and then empirically verify effectiveness of the solution, using a rapid sequence of testing and fixing. This method allows designers to quickly explore a space of design solutions and prevents focusing on large issues in the beginning of a design process. [37]

Role-playing involves acting in the role of a user in an attempt to find problems that would pertain to such a user without performing tests with external participants. This method is particularly useful when direct observation is not feasible or ethical. [37]

Secondary research is a method of research in which information is collected and organized from existing data, as opposed to obtaining first-hand research material [37].

Surveys are used to collect self-reported personal information from participants. Questionnaires and interviews are the two dominant types of survey. [37]

A *think-aloud protocol* involves test participants verbalizing their thoughts and actions while completing a task in order to reveal positive and negative aspects of an interface [37].

User-centered design is an iterative design process during which focus is placed on the potential users by involving them during the different stages of the process. Designers use *generative* methods (e.g., brainstorming) and *investigative* methods (e.g., interviews) in order to better understand users' needs. The process can typically be divided into four stages: define and context in which the prospective users may use the system; specify the users' requirements; design solutions based on the previous stages; and evaluate the created solutions against the outcomes from the first two stages. If the results are unsatisfactory after the fourth stage, return to one of the previous stages. [38]

3.3 Game development

The *MDA* (*Mechanics, Dynamics, and Aesthetics*) framework is a formal methodology which attempts to "bridge the gap between game design and development, game criticism and technical game research" [39]. Within the framework, the term *mechanics* refers to the components of game game at the level of algorithms and data representation; *dynamics* refers to the behavior of the mechanics as they act on players' input and the input from other mechanics while the game is running; and *aesthetics* refers to the emotional responses which the developer desires to evoke in players while playing the game. Aesthetics may include, among others, *sensation* (sensory pleasure), *fantasy*, *narrative*, *challenge*, *fellowship*, *discovery*, *expression* and *submission* (playing a game to pass time); for example, the physical game of *charades* may embody *fellowship*, *expression* and *challenge* to varying degrees. A game's desired aesthetics can be used to guide the selection of dynamics and mechanics – for example, *challenge* can be created through time pressure and opposing players; *fellowship* can be encouraged by sharing information with team members or using winning conditions that promote or require collaboration; and *expression* can result from letting players leave *marks* on the game (e.g., by allow them to design and construct parts of their environment). [39]

Another framework, described in the book *Game Research Methods*, uses *primitives* (the basic building blocks of games) and their relations to analyze games. The primitives include *components* – entities which can be manipulated by the player or the game system; various types of *actions*, including *player actions* (initiated by the player), *component actions* (perceived by the player to be initiated by components) and *system actions* (*not* perceived to originate from the player or components); the *goals* within the game system (not personal goals outside of the game, such as enjoyment); and *rewards*, which are granted by achieving goals. Different levels of analysis require different levels of description: describing primitives and their relations, describing the principles of design, and describing the role of the primitives and principles of design. The third description depends on the second, and the second depends on the first. [40]

4 Planning

Due to describing plans made before and during the first four weeks of the project, this section is written in the future tense.

4.1 Concept

The game is intended to be a side-scrolling platform game with 2D movement and a focus on highly responsive controls and level design which facilitates moving at high speeds. Describing the game in terms of the *MDA* framework, the primary intended *aesthetics* are *sensation* (in that the movement system is intended to be enjoyable to use) and *challenge* (in that the game is focused on continuous high-speed movement, depending on reflexes and awareness of each entity's movement speed and patterns).

The main inspirations for the game include other high-speed platformers such as *Mega Man X*[1], *Super Meat Boy*[2] and, both directly and indirectly, many of the games in the *Mario*[3] franchise. In order to better understand why these games are as engaging as they are, a basic analysis of the movement system of each game – in the case of the Mario franchise, a single entry, *New Super Mario Bros. 2*[4] – will be created and used as a partial guide during development of the prototype. The goal of these analyses is to find unique or otherwise interesting elements of the games' movement systems which may be used as inspiration for this project. The analyses will be performed by playing through approximately a third of each of the games and testing the movement systems in various different situations, noting aspects such as the size of the player-controlled character relative to the size of the camera, the speed of the player-controlled character, the use of horizontal and vertical movement, and any abilities which are useful for increasing the overall speed of the player-controlled character. The analyses are not intended to yield purely quantifiable data; rather, the games' movement systems will be subjectively examined in order to, hopefully, find some ideas which are not immediately obvious yet may be used as guidelines when create an engaging movement system in this project and in future projects.

The following list details all intended possible interactions involving the player-controlled character (*PCC*). Actions are denoted by *italic text* and buttons are denoted by **bold text**. This convention will be used for the rest of the report, although italic text is also used for general emphasis. Figures 4.1 and 4.2 show two

[1]`https://nintendo.com/games/detail/mega-man-x-wii-u/`

[2]`https://store.steampowered.com/app/40800/Super_Meat_Boy/`

[3]`https://mario.nintendo.com/history/`

[4]`https://nintendo.com/games/detail/new-super-mario-bros-2-3ds/`

state machines of varying complexity which describe part of the player's movement.

- *Run left* by holding **left**, and *run right* by holding **right**. After releasing either button, the PCC stops immediately (unless in a state where movement cannot be stopped in this manner) – i.e., the velocity is constant – and remains facing in the corresponding direction.
- Temporarily *dash* – which approximately doubles the movement speed while forcing the player to *run* regardless of whether or not they are holding **left** or **right** when initiating the *dash* – left or right for a fixed period of time by pressing **dash** while facing the corresponding direction. *Dashing* can only be initiated while the PCC is grounded, and can only be deliberately interrupted by colliding with a wall. If *dashing* is performed while the PCC is grounded but not *running*, the PCC will start *running* in the direction in which they are facing. The velocity while *dashing* is constant. This action is heavily inspired by the *dashing* in *Mega Man X*; the main difference between the two systems is that the latter does not have a fixed duration.
- *Jump* a fixed distance upwards, by pressing **jump**. The PCC's horizontal movement speed can be adjusted, by holding **left** and/or **right**, any number of times while airborne. If *jumping* while *dashing*, the *dash* is extended until the PCC lands.
- *Fall* downward, after a *jump* ends or by moving the PCC off a ledge.
- *Slide* in the current direction by pressing **slide**; this causes the PCC's sprite and collision shape to shrink vertically, letting the PCC move underneath vertically narrow spaces. Like *dashing*, a *slide* has a fixed duration and can only be initiated when the PCC is grounded. Moving off a ledge, hitting a wall or *jumping* causes a *slide* to be canceled.
- *Crouch* after a slide by remaining under a vertically narrow space within which the PCC cannot fit at normal size. This will happen whenever such a situation occurs, unaffected by the player's input, and cannot be interrupted without moving out of the space.
- Perform *wall sliding* by colliding with a wall and holding **left** or **right** (corresponding to the direction from the PCC to the wall). *Wall sliding* causes the gravity affecting the PCC gravity to decrease in strength, which means that the PCC can slide upwards for a short time if jumping immediately before *wall sliding*, and that *sliding* down occurs with a downward velocity lower than when falling.
- Perform a *wall jump* by jumping while *wall sliding*, either back toward the wall (potentially scaling the wall) or away from the wall. A *wall jump* is oriented in the direction opposite to the wall but ends after a few hundred milliseconds, after which the player is able to reverse direction or continue moving along the direction of the jump. This system is inspired by the systems of both *Mega Man X* and *Super Meat Boy*, mainly with regard to the ability to scale any wall due to the short amount of time elapsed between the initiation of a *wall jump* and when the player regains control.
- *Jump* when slightly off a ledge, in order to prevent the player from feeling as if their *jump* did not register despite them believing that it should have.
- Touching any damaging obstacle will cause the level to be restarted; the PCC

cannot remove any entities from a level.

These mechanics were chosen based on the games analyzed, primarily *Mega Man X* (from which the *dashing* system was greatly inspired), and personal preference.

For a somewhat experienced player, the core gameplay loop is intended to be as described in figure 4.3. Graphical assets will be created with the 3D modelling tool *Blender*[1], due to experience with the tool but a lack of skill with regard to directly creating 2D art (whether raster-based on vector-based); since Blender supports both animation and rendering [41], it can be used to generate both static and animated 2D graphical assets without requiring specific knowledge needed to create detailed 2D artwork, such as shadows and perspective.

4.2 Software development methodology

In order to ensure that feedback from potential players can be accounted for during development, and to allow the design to be updated as new ideas appear, an *iterative* method of development (as described in chapter 3) will be used, loosely inspired by the *Scrum* method – *sprints* will be used to maintain focus and *sprint backlogs* will be used to track what has been done and what remains to be done in a particular sprint, sorting features based on priority (which will be set based on the goals of the project, criticism from testers and observations from tests); while agile frameworks are typically intended for multiple-person teams, some of their concepts can be adapted for individuals, an example of which can be seen in [36]. Since a single developer cannot work on multiple parts of a project simultaneously, and frequently switching roles to create a game, as with a larger team of developers, is not feasible due to the test-driven nature of the development process (since a longer time would be required between tests to ensure that the prototypes are different enough to warrant testing), each sprint will focus on a particular set of features; this is intended to result in a prototype which is significantly different from the previous sprint's and which can be used during user tests.

The first sprint will focus on the game's movement system, emphasizing the *sensation* aesthetic; the second will focus on level design, emphasizing the *challenge* aesthetic; the third will focus on visual and auditory polish, further enhancing the *sensation* aesthetic; and the fourth will be used to resolve the most important remaining issues, prioritized based on the amount of time left. The order of these sprints was chosen based on the idea that creating visuals and audio without having a player seemed likely to force abandonment of some art for the sake of better gameplay, or vice versa; and creating levels before having a movement system would force the movement system to be designed around the levels rather than being designed to be enjoyable to use on its own, or to change the levels significantly after creating the movement system.

[1]`https://blender.org/`

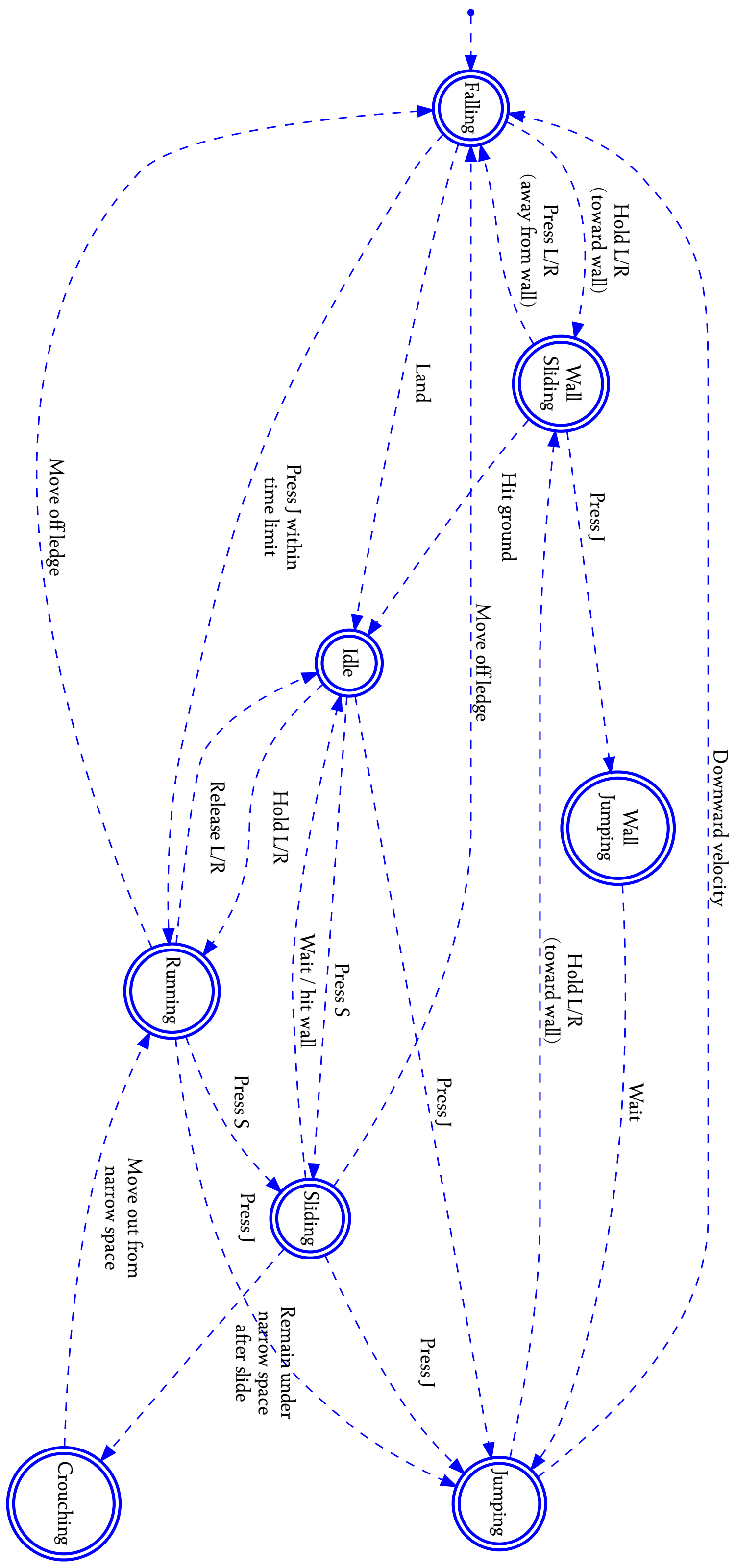

Figure 4.1: A state machine diagram describing the PCC's intended main actions, without accounting for *dashing*, power-ups or the direction in which the PCC is facing.

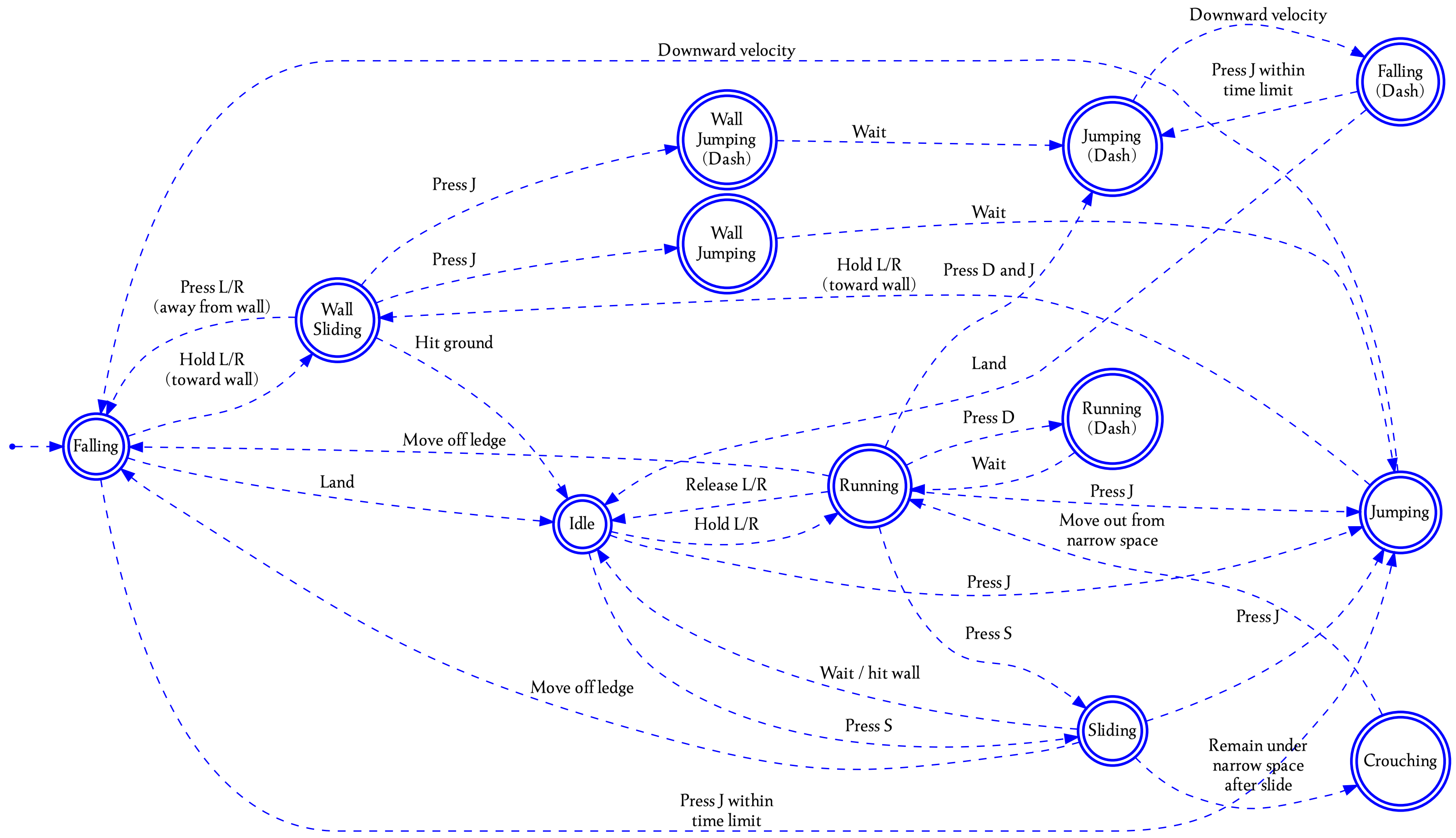

Figure 4.2: A state machine diagram describing the PCC's intended main actions, accounting for *dashing* but not for power-ups or the direction in which the PCC is facing. As shown, adding just four states causes a significant increase in complexity, as would adding a single power-up; each power-up would cause the size of the state machine to expand greatly, especially if multiple power-ups can be active at once.

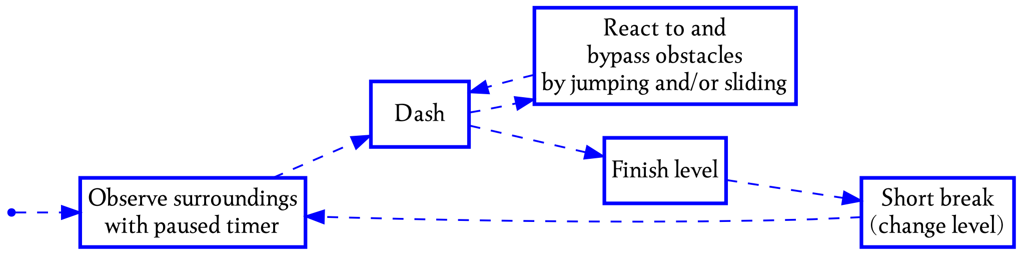
React to and bypass obstacles by jumping and/or sliding
Dash
Finish level
Observe surroundings with paused timer
Short break (change level)

Figure 4.3: The intended core gameplay loop for an experienced player, based on short levels and fast reactions to obstacles.

4.3 Interaction design methodology

The design process will be based around rapid prototyping (specifically, prototyping in which goals are divided into smaller features which can be implemented in, ideally, at most a few hours of work), in order to be able to quickly perform subjective tests of what aspects of gameplay are enjoyable and dismiss or set aside any additions or changes that do not seem to clearly improve the game; additionally, such a methodology is well-suited for an agile type of software development. Dedicating long periods of time to a single feature may makes it mentally difficult to abandon it, regardless of its quality, which may negatively impact a game's design and wastes development time which could have been spent on prototyping multiple smaller features; in a worst-case scenario, an entire game may need to be abandoned (temporarily or permanently) because of unmaintainable code, poor design and/or a lack of money to continue development, as occurred after four years of development for developer *Adrián Novell* and his team [42]. While adding major features may be unavoidable, the development process can be optimized by testing lesser versions of these features in order to partially understand whether the complete change will be worth the estimated time required to implement it. Additionally, forcing oneself to have regular deliverable prototypes to be tested may facilitate the limitation of features, keeping the developer focused on a relatively clear goal.

For each of the three first sprints, three personal user tests will be conducted, with the focus of each test being on the prototype created during that sprint. Additionally, attempts to collect additional feedback by distributing each prototype on game-focused online forums will be made before the first tests on the prototype in question. Both methods are intended to generate purely qualitative feedback, since the former cannot provide enough data to be statistically significant and the same is expected to apply for the latter; additionally, this qualitative data will not necessarily be indicative of the types of people who tend to find and play independent platform games, it will be highly limited by the small number of tests and participants, and personal connections with the testers may cause conscious or unconscious bias. The tests will mainly consist of observation, with the player being asked to reveal any thoughts they want to convey beforehand; if the player becomes unable to proceed, their movements will be observed more closely and they will be asked to describe their actions in order to locate potential issues. After each test, a short interview will be performed, during which the player will be asked questions formulated *prior* to the test as well as any questions which may appear *during* the test. For the first tests, these questions will include the player's opinion on the order of priority for the first three sprints.

Very simple variations of various methodologies (i.e., new methodologies using some of the core principles of existing methodologies) will be used:

- *Competitive testing* will be used, in a sense, through the analyses of games which will be performed during the first month of the project – in the broad sense of the word, these games *are* competitors, as any product is, although their existence only very *slightly* affects the chance of an individual playing the game produced by this project.
- A variation of the *Critical Incident Technique* and *experiments* will be used when participants appear to have difficulty advancing from a specific spot; in such a situation, the participant will be asked to describe their actions and, if the test is conducted with physical proximity, their interaction with the keyboard will be observed.
- Very simple *role-playing* will be used when playing the game in order to find in what way each level can be modified so that the level is enjoyable to play through regardless of the proficiency of the player.
- *Secondary research* will be used throughout the project as an alternative to first-hand research, since the latter would require more participants and time than this project can afford.
- The process will use elements of participatory design – by inviting testers to provide their own insight into the design of the prototype being tested – and user-centered design – by testing the results of each sprint with three people, at least once per person and sprint.

A/B testing and *crowdsourcing* will not be used because they would require far more participants than was can be feasibly expected to be available for this project. *RITE* will not be used because it is based on empirical verification of solutions, which will not be practically possible in this project. *Surveys* will not be used because, without access to a large number of participants, they appear to have very little use; experience from a previous course revealed that almost nothing valuable was obtained from approximately twenty survey participants. A *think-aloud protocol* will not be used because it interrupts any potential feeling of flow that a tester might experience and thereby prevents observation of where and when such moments may occur.

4.4 Tools

As the operating system, a Linux distribution will be used. Various BSD operating systems may be usable for developing games and using the remaining tools; however, due to a lack of experience with any BSD operating system and significant experience with Linux distributions, the former were dismissed.

As the game engine, text editor, level editor, etc., *Godot* – an open-source game engine available for Linux, Windows, macOS and other platforms [30] – will be used. The reasons for choosing this engine include prior experience with it; its support for development on Linux [30]; its ability to export to Linux, Windows, macOS, Android and other platforms [30]; and its general-purpose functionality [30].

As mentioned previously, *Blender* will be used to create visual assets. If the game will use 2D graphics, *The GNU Image Manipulation Program*[1] may be used to perform various graphics-related modifications.

To create music, a tool such as *Bosca Ceoil*[2] or *LMMS*[3] will be used. For sound effects, *Bfxr*[4], or a program with similar functionality, will be used.

Finally, the versioning system *Git*[5] will be used to track different versions of the game for the sake of testing, to keep a record of the changes made and to allow inspection of previous versions in order to facilitate finding the sources of software bugs.

4.5 Time plan

Weeks 1-4:

- Create a basic analysis of the movement system and level design of three existing successful platformers – *Mega Man X*, *Super Meat Boy* and *New Super Mario Bros. 2*. These platformers were chosen because they embody qualities which were personally wanted for the game developed during this project, including high-speed movement and highly responsive controls. *New Super Mario Bros. 2* was selected as a representative of the *Mario* franchise because it possesses, subjectively speaking, one of the most engaging movement systems of the 2D games in the series.
- Perform basic research on game development processes as described by experienced game developers, focusing on those who worked individually.
- Create the planning report.
- Perform basic research on accessibility problems in order to find out what can be implemented relatively easily.
- Create a basic prototype in Godot that showcases the movement system of the game.

Weeks 5-8:

- Perform user tests using the prototype generated by the previous sprint.
- When distributing the first and second prototypes online, the focus of the prototype will be clearly stated to prevent wasting testers' time on providing feedback which would not be useful (e.g., feedback on the visuals while using temporary art).
- Create a preliminary non-playable design for two short, linear levels. This design will be limited to focus on content in terms of gameplay (e.g., power-ups, terrain layout and obstacles) rather than appearance; the latter aspect of the level design will be added later during the process in order to limit the initial focus.
- Implement the designed level in Godot.

[1] `https://gimp.org/`
[2] `https://boscaceoil.net/`
[3] `https://lmms.io/`
[4] `https://bfxr.net/`
[5] `https://git-scm.com/`

Weeks 9-13:

- Perform another set of user tests, and then update the design document and the prototype accordingly.
- Create simple graphics for the player, enemies and some environmental objects (e.g., platforms), and add them to the prototype. Using unique graphics for the game's essential visuals ensures that the game is differentiated from other games with very low budgets.
- Gather graphical assets for less important elements, such as terrain and trees, which are available under a license that allows commercial use (e.g., *CC BY 4.0* [43]), and add them to the prototype.
- Create and/or gather sound effects for relevant events in the game, and add them to the prototype.
- Create a single piece of music to play during the level(s) using a composition program which does not require advanced knowledge of musical theory (e.g., the aforementioned *Bosca Ceoil*), and add it to the prototype. As with models, creating unique music helps make the game unique.

Weeks 14-16:

- Perform another set of user tests, and then update the design document and the prototype accordingly.
- Resolve as many of the remaining issues as possible (e.g., bugs and design problems).
- If a significant amount of time is left when the above tasks have been completed, increase the amount of content without making significant changes to the basic design or existing levels.

Weeks 17-20:

- Finalize the report.
- Prepare for and perform the presentation and the oppositions.

Throughout the project, the report will be updated; additionally, study material from various sources, especially from successful independent video game developers, will be regularly studied (i.e., read, listened to or watched) in order to improve the quality of the product and the effectiveness of the development process.

Figure 4.4 summarizes the time plan, excluding the aforementioned continuously occurring tasks.

4.6 Success criteria

The first goal – creating a high-quality game to serve as a demonstration of the skills acquired during prior studies – is highly subjective, so it will be considered successful based on personal opinion and the opinions of testers. Testers' opinions will be based on the final prototype tested – they will be asked whether they enjoyed the prototype, and their reactions will be observed throughout the tests in order to determine whether they enjoyed the game before asking the question, in order to reduce the possibility of a positively biased answer resulting from personal connections.

The second goal – to show general self-improvement in terms of game design and programming – is less subjective. Quantitative criteria based on surveys are not particularly indicative of success (e.g., measuring average playtime or the amount of

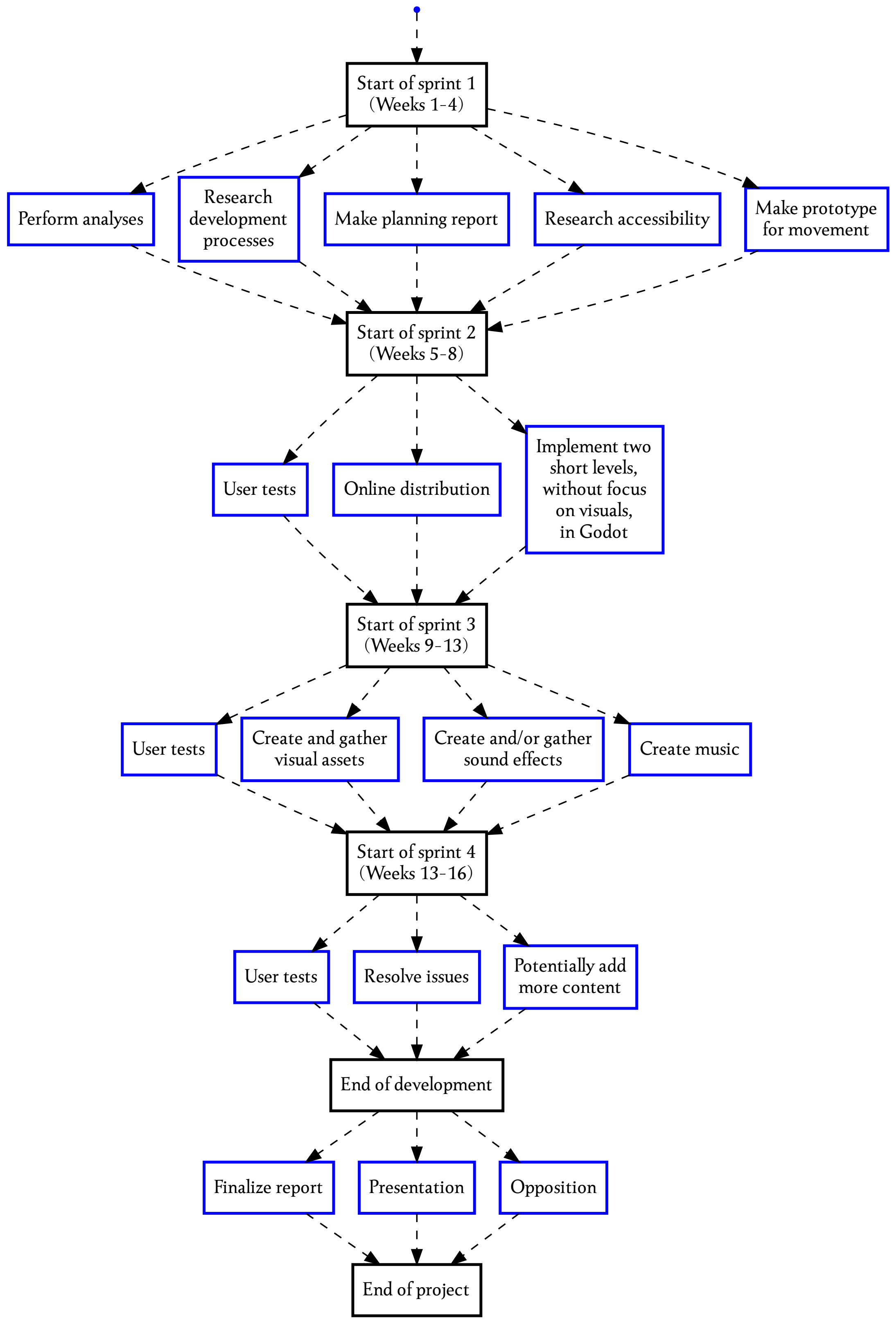
Start of sprint 1 (Weeks 1-4)
Perform analyses
Research development processes
Make planning report
Research accessibility
Make prototype for movement
Start of sprint 2 (Weeks 5-8)
User tests
Online distribution
Implement two short levels, without focus on visuals, in Godot
Start of sprint 3 (Weeks 9-13)
User tests
Create and gather visual assets
Create and/or gather sound effects
Create music
Start of sprint 4 (Weeks 13-16)
User tests
Resolve issues
Potentially add more content
End of development
Finalize report
Presentation
Opposition
End of project

Figure 4.4: A diagram summarizing the time plan, excluding the work which will be done continuously during the project.

people who claimed that they enjoyed the game), and the same applies to objective criteria (e.g., the combined size of the levels). As a result, the success criteria for this goal will be based on improvement shown in logs of design, programming and the general development process; if clear improvement is shown in all three areas, this goal will be considered reached. As an example of using reflective logs, in the article *Writing a research paper: reflections on a reflective log*, the author describes his use of logging in order to describe the method by which he formulated a philosophical paper, claiming that it resulted in a personal increase of confidence in his process and that it revealed certain components of the process which may be useful for future work [44].

The third goal – to create a set of guidelines for future developers working with an open-source toolset, focusing on those working individually – will be informally determined by publishing them online and gathering any provided feedback.

5 Execution and Process

Unless otherwise mentioned, the plan listed in chapter 4 was followed.

5.1 General process

No formal design document was used, mainly because there was no need to communicate ideas to another team member, the game did not have a particular intended style with regard to visuals and audio, and there was no story to consider – the only important part during early development was to create a game with enjoyable mechanics and engaging level design. Instead, plans and ideas were written down as informal notes and lists of tasks, many of which were part of the sprint backlogs.

The feasibility of creating graphics using Blender was tested during the first and second prototypes to ensure that another method would not need to be found. The second prototype introduced some higher-fidelity art to attract more feedback online, but the majority of the work on visuals would be done in the third prototype.

The movement system was reworked in sprint 2 and slightly in sprint 3. Because of this, and due to an unexpected amount of time needed for work on this thesis, the third sprint was extended by one week.

5.2 Game design

Slight acceleration and deceleration was added to the player-controlled character's movement due to personal testing showing that the addition would lead to less precise input being required from the player when the player-controlled character was airborne and attempting to land – with the new system, holding a button for a short time causes smaller, more precise movements than with constant horizontal velocity, which facilitates landing within a specific area.

Dashing was changed from discrete to continuous dashing, because the former would force players aiming for consistent high-speed movement to press **dash** every second; this issue was revealed through both personal tests and user tests.

Sliding was changed so that, instead of having constant horizontal velocity, it decelerates from dashing speed to standing still, and it can be interrupted by releasing **slide**, which rewards players who stop *sliding* according to their location in the level but does not force them to exit the state by jumping, colliding with a wall or falling off a ledge. Based on personal testing, the old system significantly lowered the overall speed of the game in stages where *sliding* was necessary, which conflicted with the goal of consistently high-speed movement.

Wall jumping was made more accessible – the player-controlled character can *wall jump* without the player holding a directional button (in the direction of the wall in question) as long as they are airborne while *dashing* towards the wall; alternatively, the player can press the directional button which would move their character away from the wall slightly after jumping and still be able to *wall jump*, rather than falling off the wall, as in the initial system. This change was largely based on feedback from tests, which showed some participants experiencing difficulty with a level focused on *wall jumping.*

The player is able to press **jump** slightly before their character lands and still *jump* when the player touches the ground; this was implemented in order to prevent players from feeling that their input was not correctly registered.

5.3 Programming

Nearly all problems related to programming were caused by difficult-to-find bugs as well as the implementation of the initial code for the player (during the first sprint) – in order to make maintainable code for the player, a *finite state machine* (*FSM*) structure was used for the player's different states, but a single FSM wasn't necessarily enough, because some concurrency was needed in order to easily implement powered-up states (e.g., being able to double jump after grabbing a specific power-up). Initially, each state in which *dashing* could occur was duplicated (i.e., one without *dashing* and one with), which lead to the code being difficult to maintain and expand upon – power-ups were not included in the initial state machine diagram, and adding more than one would have been a lengthy and error-prone process. An attempt was made to transition the existing code to a *pushdown automaton* instead, but this did not help, and the concept was abandoned. Eventually, a simpler FSM was used, separate code was used to handle *dashing*, and another finite state machine was used to handle power-ups, with the three systems communicating through a central `Player` class.

Some minor difficulties occurred, usually due to trying to implement too much in a single commit to the *Git* repository, but these were eventually solved, and no apparent bugs remain.

5.4 Creating visuals

The models used were mostly generated procedurally with basic shapes – the player character is a sphere with a procedural texture; the sawblades were freely available online and were given two procedural textures; the portals which move the player to another level were made using Blender's *displace*[1] modifier, as were the portals which emit sawblades (both of which can be seen in figure 5.1); and the tiles (i.e., ground, walls and roofs) are boxes with textures (procedural for the brown tiles, and a freely available metallic texture for the white tiles). The entity which required

[1]`https://docs.blender.org/manual/en/latest/modeling/modifiers/deform/displace.html`

significantly more time to create was the central obstacle of the sixth level, which can be seen in figure 5.2 and was created using the following techniques:

- The *skin* modifier[1] allows the creation of a character and armature (essentially, the skeleton of the character which can be used to position limbs and other parts of the body) by placing the points of the armature and adjusting the width and thickness of the regions between each pair of points. This was used in addition to the *subdivision surface* modifier[2] to create rounded shapes for the limbs of the entity.
- *Inverse kinematics* (which, for a non-advanced user of Blender, is simply a way to create animations such as for walking without manually positioning the relevant bones in the armature) was used to create the entity's animation for walking.
- A procedural, checkerboard-like texture was used to cover the outer surface of the entity.

Additionally, the *solidify* modifier[3] was used to create outlines for all entities but the tiles, and a custom cartoon-inspired shader was used to create simple shading for the generated raster graphics.

5.5 Creating audio

Bosca Ceoil, the tool intended to be used to create music for the game, was found to be dependent on the *Adobe Integrated Runtime*[4], which is not open-source and does not seem to have an open-source alternative. No equivalent tool which was both seemingly easy-to-use *and* open-source was found during brief research; additionally, features related to visuals and sound effects had been assigned higher priority, which lead to the creation of music being postponed and eventually abandoned. As replacement, three freely available pieces of music were gathered – one for the title screen, and two as alternatives for the music playing during gameplay.

A similar problem occurred for *Bfxr* earlier in the project – as mentioned in chapter 2, it appears to require either the proprietary *Adobe Flash Player*[5] or a closed-source operating system [33] – but a browser-based derivative, *Jfxr*, was quickly found and used as a replacement to create each of the sound effects used in the game.

5.6 Testing

The following is a summary of the feedback received from the tests performed throughout this project.

[1]`https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/skin.html`

[2]`https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html`

[3]`https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/solidify.html`

[4]`https://get.adobe.com/air/`

[5]`https://get.adobe.com/flashplayer/`

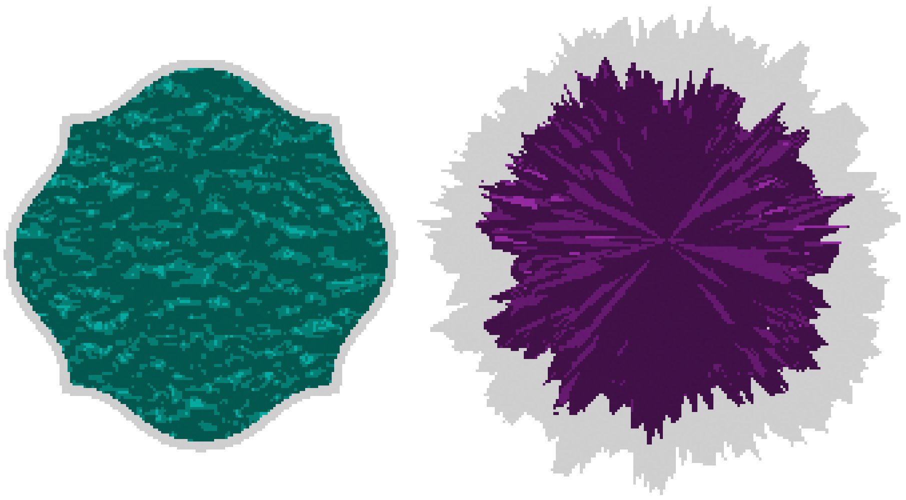

Figure 5.1: The initial frames of the two *portals* used in the game – the left is used as each level's goal, and the right is used as an emitter of moving *sawblades*. Each contains a particle system which causes a circular pattern to form within it.

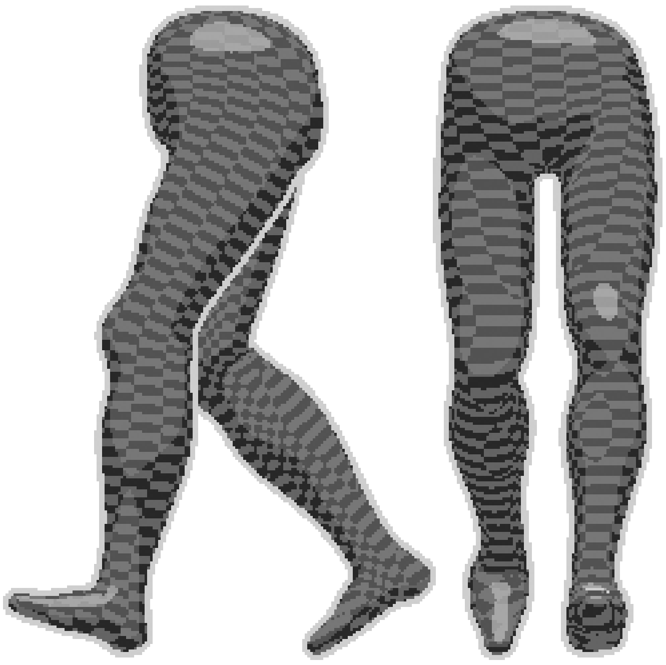

Figure 5.2: A side view (left) and front view (right) of the central obstacle in the sixth level. Its model was created entirely in Blender.

- Participant A claimed they enjoyed the game but believed that it may have been more enjoyable with the ability to perform a *jump* while airborne. (Tested using a very early version, before they were numbered.)
- `(v0.1.0)` Participant A claimed that they would prefer being able to continuously *run* over the current *dashing* system; preferred the visuals without the pseudo-3D effect; wanted dynamic *jump* heights (i.e., *jumping* whose height depends on how long **jump** is held), or at least a short *jump* and a *long* one; wanted to be able to scale walls more quickly; wanted greater *jump* speed; and found some instructions in the tutorial unclear. Each of these issues were resolved in later versions.
- `(v0.1.2)` Participant B preferred dynamic *jumping* or the current *jumping* system over a system with two *jump* heights. Additionally, they found part of the tutorial difficult to understand – the same part for which problems occurred in the previous test.
- `(v0.1.2)` Online feedback revealed, again, that the aforementioned part of the tutorial was difficult to understand. Additionally, one person suggested that transitions between the colors of the player should be gradual rather than immediate. Other positive reaction to the use of colors to indicate state resulted in this aspect remaining in later versions.
- `(v0.1.13)` Participant C provided no negative feedback.
- `(v0.2.0)` Participants A and B provided no negative feedback.
- `(v0.2.3)` Importantly, this prototype added acceleration and deceleration to movement. Participant A was unsure as to whether they preferred the movement system with or without acceleration and deceleration – it varied depending on the stage, as some parts were easier without the changes – but they did not want the acceleration and deceleration to be significantly lower (i.e., they did not want the time needed to turn around to increase).
- `(v0.2.0)`[1] Online feedback revealed some issues with regard to leniency for those who used unexpected sequences of button presses when attempting to perform a *wall jump* while *dashing*; as a result, the *wall jumping* system was modified to be less strict without simplifying the game for existing styles of play. Additionally, concern over the high pitch of the sound effect played when the player-controlled character *jumps* was raised; to accommodate this, the default pitch was slightly lowered and the ability to adjust the volumes of individual sound effects was added.
- `(v0.2.58)` Participant C provided some criticism regarding the positional sound effects on level 6 (specifically, that the volumes did not lower enough in volume when the entities creating them were no longer visible); this was adjusted in a later version. In general, the participant claimed that they enjoyed the game, and observation of their behavior seemed to agree with this opinion.
- `(v0.2.58)` No substantial feedback was received in response to online distribution of the prototype.
- `(v0.2.65)` Participant B found nothing to criticize and claimed that they

[1]Some of this feedback was given for a later version, before `v0.2.58`, but the exact version number was not recorded.

enjoyed the game; again, this seemed to fit with their observed behavior.
- `(v0.2.65)` Participant A criticized the color of the white tiles in comparison to the brown tiles and wanted an option to skip the transitional screen shown after clearing a level; they claimed that they (and seemed to have) generally enjoyed the game, although they did not notice many of the changes made since the prototype they tested previously (`v0.2.3`).

A detailed list of the prominent changes for each version can be found as part of the project's Git repository[1].

[1] `https://gitlab.com/ael-dev/mtgame`

6
Results

6.1 The game

The game, *CheckerSphere*[1] is a 2D side-scrolling platformer with a focus on high movement speed, and contains, among others, the following significant features:

- Pre-rendered 2D visuals generated by Blender, and procedural visual effects generated in Godot.
- Seven short levels. The game can be cleared within a minute, but tests have shown that it typically requires at least ten minutes on the first attempt.
- Dynamic difficulty – many levels have paths of varying difficulty, rewarding more skilled players with a lower level completion time.
- An optional pseudo-3D effect gives the illusion of three-dimensional rendering of tiles.
- A visually bland yet extensive settings menu which provides many accessibility-related options.
- Music for the title screen, and a choice between two pieces of music to play during gameplay; custom music was deprioritized, so all music was made prior to this project by unrelated musicians.
- Sound effects for the majority of the entities' actions. The sound effect played when the player-controlled character *jumps* uses random changes in pitch in order to reduce repetition.
- Very short loading times, partially due to asynchronous loading of levels on the title screen.
- The player can temporarily slow down gameplay in exchange for the level's timer counting real time; using this feature therefore allows players to more easily pass parts which may otherwise be very difficult for them, at the price of a worse level completion time.
- Visual effects, including particle systems, *screen shake* (randomly moving and/or rotating the camera's position during a short period of time) and visual trails placed after the player-controlled character when moving at high speeds.

The following list details all possible interactions involving the player-controlled character (*PCC*). As mentioned in chapter 4, actions are denoted by *italic text* and buttons are denoted by **bold text**. Figure 6.1 shows the states present in the finite state machine used for the character's actions.

- *Run left* by holding **left**, and *run right* by holding **right**. Horizontal movement

[1]The game is available in executable form at `https://ael-dev.itch.io/platformer` and as a Godot project, including source code, at `https://gitlab.com/ael-dev/mtgame`.

has very slight acceleration and deceleration to facilitate maneuvering while airborne.

- Temporarily *dash* – which approximately doubles the movement speed while forcing the player to *run* regardless of whether or not they are holding **left** or **right** when initiating the *dash* – left or right while **dash** is held. *Dashing* can only be initiated while the PCC is grounded, and can be deliberately interrupted by releasing **dash** while grounded, reversing direction while grounded or colliding with a wall. If *dashing* is performed while the PCC is grounded but not *running*, the PCC will start *running* in the direction in which they are facing.
- *Jump* with **jump**, with the height depending on how long **jump** is held.
- *Fall* downward, after a *jump* ends or by moving the PCC off a ledge.
- *Slide* in the current direction by pressing **slide**; this causes the PCC's sprite and collision shape to shrink vertically, letting the PCC move underneath vertically narrow spaces. A *slide* occurs until **slide** is released or after approximately one second has passed.
- *Crouch* and *wall slide*, as described in chapter 4.
- Perform a *wall jump* by jumping while *wall sliding*, either back toward the wall (potentially scaling the wall) or away from the wall. A *wall jump* is oriented in the direction opposite to the wall but ends after a few hundred milliseconds, after which the player is able to reverse direction or continue moving along the direction of the jump. This system is inspired by the systems of both *Mega Man X* and *Super Meat Boy*, mainly with regard to the ability to scale any wall due to the short amount of time elapsed between the initiation of a *wall jump* and when the player regains control.
- *Jump* when slightly off a ledge or slightly before landing, in order to prevent the player from feeling as if their *jump* did not register despite them believing that it should have.
- Touching any damaging obstacle will cause the level to be restarted; the PCC cannot remove any entities from a level.

The game contains the following damaging obstacles:

- *Sawblades*, which are of various sizes and do not move.
- *Sawblades* moving along a fixed path.
- *Sawblade portals*, which emit moving *sawblades* that are destroyed upon contact with a wall. The *portals* themselves, like other obstacles, harm the player upon contact, as do the *sawblades* emitted by them.
- The entities shown in figure 5.2, each of which walks at a constant velocity until it comes close enough to a wall or damaging obstacle, at which point it reverses its horizontal direction and then resumes movement.

Each obstacle which changes position starts moving only when the player first moves to within a fixed distance to the obstacle, in order to ensure that the player can predict the starting position of each obstacle without needing to be mindful of the amount of time spent to reach it.

The game contains a single power-up – the *air jump*, which allows the player to perform a jump while airborne, losing the ability after its first use, when touching a wall or when landing.

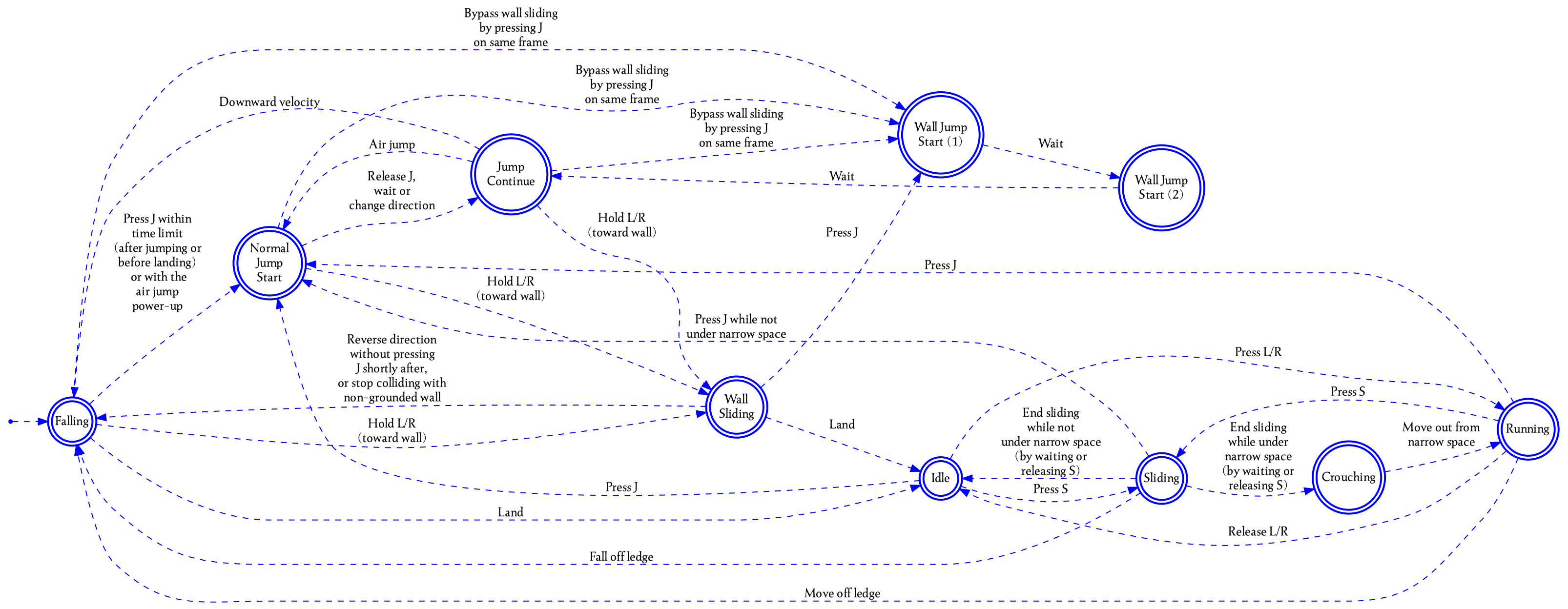

Figure 6.1: A state machine diagram describing the PCC's main actions. Unlike in the plans, *dashing* is handled by separate code (equivalent to a two-state FSM) and power-ups are handled by another FSM; the three systems communicate through a central `Player` class; this class is also used to store variables such as the direction in which the PCC is facing. A version without labeled edges can be seen in figure 6.2.

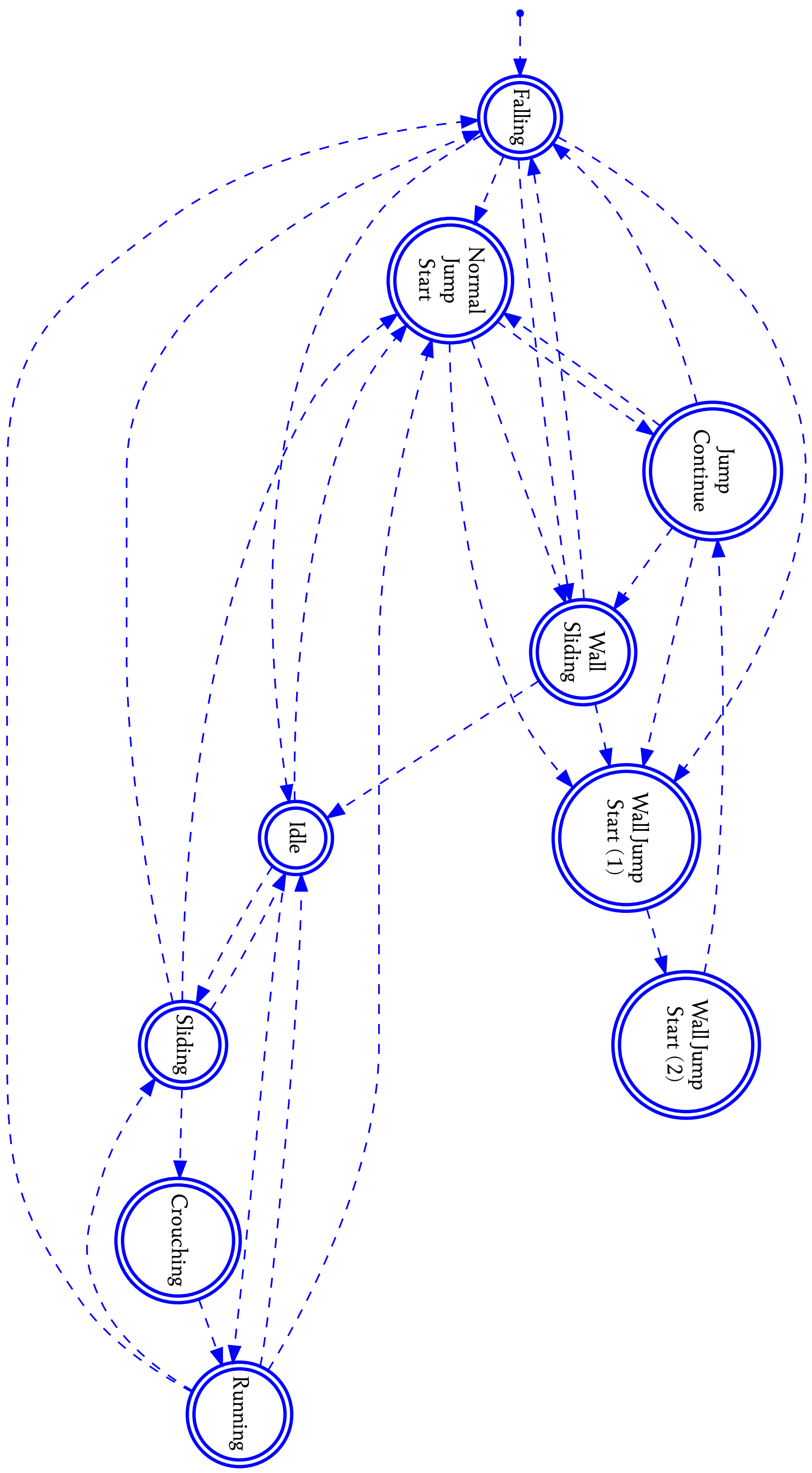

Figure 6.2: A version of figure 6.1 without labeled edges.

6.1.1 Accessibility

Accessibility is not inherently related to open-source and/or individual game development; however, improving the accessibility of a game can lead to an increase in the number of players (and therefore greater revenue in the case of a commercial release), and can solve some arguably ethical issues by accounting for potential players with disabilities. Many changes that improve accessibility are relatively easily implemented, such as allowing players to choose which buttons activate particular actions; even without considering ethics, since more players causes greater revenue for commercial releases (with exception for potential illegal downloads), implementing simple accessibility-improving features seems to be largely beneficial to developers.

The following is a list of accessibility-related features present in *CheckerSphere*. The items are highly based on the *game accessibility guidelines*[1] mentioned in chapter 2, especially the *basic* set of guidelines. Therefore, each feature will be followed by a reference to the relevant guideline in the order *list*, *section*, *item number*. If a guideline occurs twice, only the first occurrence (from *basic* to *advanced* and from top to bottom) will be referenced.

The following accessibility-related options can be modified within the game's settings menu:

- Every in-game action can be remapped. (*Basic*, *Motor*, 1)
- The master volume for music, and the individual volume of each track, can be set to between 0 and 500%, in increments of 10%. (*Basic*, *Hearing*, 2)
- The master volume for sound effects, and the individual volume of each of the three effects caused by the player, can be set to between 0 and 500%, in increments of 10%. (*Basic*, *Hearing*, 2)
- The base time scale can be set to between 100% and 50%, in increments of 10%; additionally, the temporary slow-down feature can be set to between 10% to 90%, in increments of 10%. The player can also choose whether to have slow-down be toggled when a button is pressed or enabled while a button is held. (*Intermediate*, *Motor*, 5)
- All particle effects can be disabled.
- The pseudo-3D factor, which causes the tiles to be repeated twice in the background and twice in the foreground for a 3D-like effect, can be increased, decreased or removed entirely.
- Any shaking of the camera can be disabled.
- The smoothing of the camera can be disabled, or its speed can be adjusted.

Additionally, the following accessibility-improving features were integrated into the core design of the game:

- Controls are very simple, requiring only five buttons; the game can be played with one hand if the actions are remapped appropriately. (*Basic*, *Motor*, 4)
- The game can be started without the need to navigate through multiple levels of menus – the player only needs to press a single button to start. (*Basic*, *Cognitive*, 1)
- An easily readable default font size is used. (*Basic*, *Cognitive*, 2)
- Simple, clear language and text formatting is used. (*Basic*, *Cognitive*, 3-4)

[1] `https://gameaccessibilityguidelines.com/`

- Interactive tutorials are included. (*Basic*, *Cognitive*, 5)
- Flickering images are avoided. (*Basic*, *Cognitive*, 7)
- Clear contrast between text/UI and background is provided. (*Basic*, *Vision*, 6)
- Color is not essential, and the game has been thoroughly tested with filters used to simulate four types of colorblindness (*protanopia*, *deuteranopia*, *tritanopia* and *achromatopsia*). These particular types of colorblindness were tested due to being part of a plugin for Godot[1]. (*Basic*, *Vision*, 1)
- Audio is not essential, and the game has been thoroughly tested with all audio muted. (*Basic*, *Hearing*, 3)
- Speech input is not required. (*Basic*, *Speech*, 1)
- Visuals in the foreground and the background are clearly distinguishable; this was tested with the different colorblindness filters.
- Reading is not strictly required, if the player is willing to experiment with the different possible buttons. (*Intermediate*, *Cognitive*, 7)
- A wide choice of difficulty levels is offered dynamically, by allowing players to take different paths through each level as well as to slow down time if a section is found to be too difficult. (*Basic/Intermediate*, *General*, 1)
- All interactive elements in the user interface are stationary. (*Intermediate*, *Motor*, 2)
- Button mashing (i.e., high-frequency input) is never encouraged or required. (*Intermediate*, *Motor*, 6)
- Windowed mode is supported and the game can be resized, with black bars appearing when the intended aspect ratio must be enforced. (*Intermediate*, *Motor*, 7)
- Reminders of controls can be found in the settings menu. (*Intermediate*, *Cognitive*, 3)
- A buffer of approximately 100 milliseconds is used for *jumping*, both before landing and after moving off a ledge or wall. A similar buffer is used for *dashing*, both before landing and after *jumping*.
- The player does not need to use a computer mouse at any point other than in the settings menu, and this is only a requirement because creating a keyboard-driven settings menu was expected to take a significant amount of time. (*Basic*, *Motor*, 2)

6.2 Guidelines

This section presents a list of guidelines and related information, specifically focusing on open-source and/or individual game development, largely based on personal experience from this project. In addition, a list of potentially useful material which was studied during this project can be found in appendix A.

The primary guidelines in this section are denoted by **bold text**.

[1]`https://github.com/paulloz/godot-colorblindness`

6.2.1 Overcoming individual difficulties

If you experience great difficulty with some aspect of game development, try to find an alternative approach.

As an individual developer, a particular part of game development may seem far more difficult than the rest – for example, during this project, drawing two-dimensional art for the game did not seem to be an option due to a lack of experience and an apparent lack of time to learn. To solve such a problem, unless the problem in question is one which cannot be avoided (e.g., general game design or some form of scripting or programming, whether visual or text-based), an alternative may be found:

- For *CheckerSphere*, 3D modelling was used in place of 2D raster graphics or vector graphics; another alternative may be to use a very simple art style (e.g., small sprites with few colors) that does not rely on correct perspective, shading or any other seemingly difficult aspect of drawing[1].
- Depending on the type of game, level design can be partially avoided by using procedural generation; for platformers specifically, this can be seen in research such as [45], [46] and [47], and has been showcased in games such as *Cloudberry Kingdom*[2].
- Creation of music can be avoided by using freely available music or creating a game which uses only ambient background noise.

6.2.2 Analyzing existing games

Analyze successful games in your chosen genre to find guidelines and inspiration for design.

The following personal guidelines were gained from the analyses with regard to development of high-speed 2D platform games; they are not necessarily applicable to *all* such games, but they were helpful in this project.

- Allow people to recover from small mistakes in some way (e.g., by allowing the player-controlled character to jump back up from a wall, as in all three of the tested games).
- Have the field of view be wide enough, relative to the player-controlled character, to allow the player to react properly. (This was occasionally an issue in *Mega Man X*, but never in *New Super Mario Bros. 2* or *Super Meat Boy*.)
- Make harmful entities stand out from the non-harmful ones. (This was also a slight problem in some areas of *Mega Man X*.)
- When introducing some new concept, do so in a way which allows the player to test the concept without losing significant progress if they make a mistake.
- Sound effects and visuals are very important for conveying movement; movement is not as satisfying if the player-controlled character seems to be stationary.
- Allow the player to move at different speeds (e.g., through a *running* mechanic) and allow those who prefer to move slowly do so (for the most part).

[1] An example of such art can be viewed at `https://0x72.itch.io/16x16-industrial-tileset`.

[2] `https://ubisoft.com/en-us/game/cloudberry-kingdom/`

Additionally, a potentially important mechanic which was missed in the analyses is the use of slight acceleration and deceleration to increase the precision of airborne movement; this was eventually implemented, but if this had been realized earlier, it may have had a greater impact on the quality of the movement system and the level design.

Informal notes made as part of the analyses can be found in appendix B.

6.2.3 Using open-source code and other free assets

When using freely available assets (e.g., code, visuals and audio), the following guidelines may be of use:

- **Ensure that each of the assets used has a clear and legally valid license that was issued by the asset's creator.**
  If an asset is released under an obscure or home-made license, unless it is very short and clear, it may be safer to find a similar asset released under a more well-known license. For code, a list of many open-source licenses can be found at the website of the *Open Source Initiative*[1]; for other types of assets, the *Creative Commons*[2] licenses may be used.
- **Ensure that each asset's license fits with the purpose of the project.**
  If the game is to be released commercially, avoid any license which prohibits commercial use, such as *Creative Commons BY-NC 4.0*[3] unless it is possible to distribute assets separately from the game and to market the game with screenshots showing used assets without conflicting with the non-commercial restrictions. The same applies to code available under *copyleft* licenses – in some cases, such as with the *GNU General Public License*, the license may force derivative projects to be licensed under the same license [48], which may not be feasible depending on potential contractual obligations (e.g., with publishers) or other code used in the project whose licenses are incompatible with a particular copyleft license.
- **Be careful when selecting freely available visuals and audio to use.**
  If a resource is frequently found in a variety of other games, if a player recognizes it, the perceived uniqueness of the game may be lessened, or they may otherwise lose interest – in the worst case, the game may be seen as an *asset flip* (which, as mentioned in chapter 1, is a low-budget game which consists mostly of freely available or paid-for assets and is sold at a low price to gain profit), even if every other aspect of the game was created during development. However, using existing assets may be the only feasible way to complete an otherwise ambitious project, and some parts of a game may even use such assets without most players noticing, such as a generic chair placed in a larger scene – as stated by Peter O'Reilly, the global head of the *Unity Asset Store*[4], "Even a chair can take a really good, talented artist three to four days to do well" [49].
- **Modify existing visuals and audio to make them less recognizable.**

[1]`https://opensource.org/`

[2]`https://creativecommons.org/`

[3]`https://creativecommons.org/licenses/by-nc/4.0/`

[4]`https://assetstore.unity.com/`

As a partial countermeasure against being perceived as an asset flip, personal modifications may be made to downloaded assets – for example, 3D models can be changed through the use of rendering shaders and/or custom textures.
- **Ensure that all visuals and audio fit the style of the game.**
  If an asset seems out of place in the surrounding environment or the game as a whole, a player's level of immersion may decrease.

A list of resources used or found during this project can be found in appendix A.

6.2.4 Direct or indirect use of 3D models

The use of 3D models, or rendered 2D versions of such models, is relevant for both 3D and 2D games, and repositories of freely available 3D models should not be discounted as sources of visual assets for 2D games.

An example of a *commercially* released game which uses animated sprites generated from 3D models is *Dead Cells*. Thomas Vasseur, one of the two artists who created visual assets for Dead Cells, revealed that using 3D models saved significant amounts of time when animations needed to be changed with regard to timing – unlike animations originally created as raster graphics, the keyframes of 3D animations can simply be adjusted to solve such issues – and allowed easy reuse of animations or other parts from older models. [50] An animated example of a model and its raster graphics counterpart in Dead Cells can be seen in [50].

As mentioned previously, due to a personal lack of skill in terms of directly creating two-dimensional art, a similar method was used in this project, albeit with different software (i.e., Blender); this provided many benefits, including the following:
- It allows visual assets to be created without requiring the creator to be capable of drawing with correct perspective or shading.
- Compared to using 3D models in the engine, 2D art may significantly improve performance and allows the use of features of a modelling tool which cannot be exported to a model or is not supported by the game engine being used (e.g., Blender's simulations of physics).
- It enables the creation of non-trivial animations (e.g., an animation representing walking for the obstacle shown in figure 5.2) relatively quickly.
- Procedural modifications to meshes can be used to generate models without, or in addition to, manual changes in topology.
- Procedural textures and freely available image textures can be used to create textured surfaces.
- Add-ons, or external tools that export to a format usable by the 3D modelling program being used, can facilitate the creation of specific types of models – e.g., to create a model of a human, the tool *MakeHuman*[1] can be used to export a model compatible with Blender.
- Freely available 3D assets can be found and modified to suit a game's art style (e.g., by using a custom shader in the modelling software) before using them to generate 2D art; this may also serve to make some who would otherwise remember a particular model from another work not recognize it, which partially alleviates the problem of being perceived as an asset flip. This can also be

[1]`https://github.com/makehumancommunity/makehuman`

done with raster graphics, but the set of possible changes that can be applied to 3D models is significantly greater – e.g., applying textures is easy for 3D models but cannot be done for raster graphics without manual reconstruction of each frame.

6.2.5 Visual effects and sound effects

Development of this project has shown that, despite having no effect whatsoever on a game's mechanics, visual effects can positively affect the enjoyment experienced when playing a game, and removing them from an existing game can result in the game being less enjoyable to play; this effect can be seen by configuring *CheckerSphere* to disable all particle effects. This resulted in the following guidelines regarding the use of visual effects and sound effects:

- **Try adding particle effects where appropriate**.
  Experiment with appearance and duration until a suitable effect is found, assuming that the engine being used has a visual editor for particles or some external editor can be used to generate code for a shader usable in the engine.
- **Use tests to determine if certain visual effects overwhelm players.**
  An overuse of visual effects may be distracting or otherwise annoying to certain players.
- **Do not overuse particle effects**.
  In addition to overwhelming the player, particle effects can serve as a bottleneck for a player's graphics card; countermeasures for performance-related issues may include using larger but fewer particles, using additively or subtractively blended particles to reduce the need for sorting, and generating particles on the graphics card where possible [51]. In some cases, particle effects may be partially or entirely replaced with pre-rendered raster graphics without significant loss in visual fidelity or apparent variety of particles (e.g., a 2D fire may use raster graphics for the flame and particle effects for sparks).
- **Instead of playing the exact same sound repeatedly, shift the pitch randomly within an appropriate range before playing it**.
  This may reduce perceived issues with repetition, which is especially important for frequently played sound effects.
- **Consider providing alternative versions for extremely frequent sounds.**
  In such a case, the game may benefit from alternative versions instead of, or in addition to, pitch changes – an example of the latter can be seen in *The Witness*, which is focused on walking and therefore includes over a thousand alternative sound effects for footsteps [52].
- **Use tests to determine if certain players perceive specific sound effects to be unpleasant.**
  Some sound effects (e.g., high-pitched sounds) may cause players to become irritated or unwilling to continue playing.

7 Discussion

7.1 Process

7.1.1 Goals

The first goal was fulfilled due to testers appearing to, and claiming to, enjoy the final tested prototype.

Subjectively speaking, the second goal was reached – personal improvement has undoubtedly occurred in terms of design, programming and the general development process, as evidenced by an increased knowledge of game design, especially with regard to creating an enjoyable movement system; programming, especially using state machines and various features of Godot (e.g., particle effects and tweening); using Blender, especially with regard to the use of procedural generation to make simple models more appealing; and working with an adaptation of an agile methodology for a single person. However, excluding the logs for changes made to the game during development (which are detailed and available as part of the project's Git repository[1]), the logs which were intended to be kept rigorously were often lacking in entries due to a general lack of interesting decisions or quandaries – as a result, it is more difficult to prove that the second goal has been reached. The lack of thorough logging was a significant regret – since logs were only written when they seemed subjectively worthy of being written, interesting points which may have appeared if daily logs had been made may have been lost. The logs which do exist have been integrated into chapters 5 and 6.

The third goal was fulfilled due to a set of guidelines having been created; whether these will be useful to other game developers remains to be seen, but they will be published online for the scrutiny of interested parties.

7.1.2 Methodology

If this project were to be restarted, further focus would have been placed on the movement system before beginning to work on level design. Due to the development of the first prototype sharing time with several other tasks, mainly the writing of the planning report, the first sprint was significantly shorter than the subsequent ones and therefore encroached on the subsequent prototype, and there were some subjectively unsatisfying parts remaining even after the second sprint, which were mostly resolved in the beginning of the third sprint. Additionally, further research

[1]`https://gitlab.com/ael-dev/mtgame`

regarding different types of movement systems and how to implement them well would have been performed, rather than trying to implement one from scratch, since this would have saved time and may have further improved the movement system, especially in its initial stages.

The choice of order for prototypes seemed fitting at the start, and it appears to have been a good choice based on feedback from testers – some claimed that all alternatives were inferior, and others mentioned that an alternative may have been to swap or combine the first two sprints. As mentioned, these sprints were not strictly limited to their intended scopes since the movement system needed to be modified in the second and third sprints based on personal and external testing, and some visual polish was added during the second prototype in order to increase the number of people willing to test it through online distribution, but any other exceptions to the plan were minor. Dedicating the third sprint to visuals prevented having to abandon art that was no longer relevant (or using worse alternative level design and/or mechanics to keep old art). Personally, the only feasible alternative to this order would have been to merge the first and second sprints into one so that the movement system and levels could be designed simultaneously, which may have resulted in either of the two being better than they currently are – for example, the levels may have been improved if they were designed around the latest version of the movement system, including features such as different jumping heights and easier aerial maneuvering.

Even with a relatively small project like this, temporary lack of motivation was experienced several times, and this issue could only partially be circumvented by switching between roles; an example of such an occasion was when the movement system was found to be partially lacking and needed to be greatly redesigned to reach the intended level of quality – at this time, it seemed that the only solution was to completely rewrite the code for the movement system, but, after calming down and reconsidering, a compromise was found which only required modifying *parts* of the code rather than abandoning everything and starting anew. For a developer wanting to release a commercial game, a potential solution to this problem would be to push past the motivational barrier until the next fully playable prototype and decide whether to continue at that point, but this will only be feasible in some situations (e.g., if relatively little time has been spent on the project at that point and if the developer can afford to abandon or halt the project); if the issue with the project is that the developer does not think it will result in a satisfactory product, the best solution may involve a significant redesign or completely abandoning a project.

7.1.3 Design

Few *interesting* decisions were made during the development of *CheckerSphere*; rather, almost every decision was small in scope and based on testing. A feature was quickly implemented and tested, and it was kept only if it improved the game or could do so with further changes; doing so appeared to be a more effective approach than attempting to theorize as to whether the addition of a feature would be beneficial to the game. However, some changes, such as the following, warranted more consideration.

The initial implementation of the *sliding* action was going to be removed due to personal tests revealing that it hindered the overall speed of the game, but it was reworked into its current system rather than being removed; this was partially due to the movement system seeming too simplistic without it, and because a level had been designed around the mechanic. In the final prototype, the mechanic is required in all levels but one after its introduction.

The modification of the movement system to include acceleration and deceleration required some deliberation in order to justify whether the changes would make a significant enough addition, since the implementation was expected to take a significant amount of time; eventually, attempting to maneuver the player-controlled character to land on a single tile during personal tests revealed that that the current system made any attempts at only *slightly* adjusting the direction highly difficult, so the feature was implemented and adjusted until the airborne movement was more manageable.

7.2 Open-source development

Aside from the use of a proprietary graphics driver, proprietary hardware and likely websites running closed-source software, using strictly open-source tools caused very few problems in this project.

7.2.1 This project

Godot, the engine used to create the game, was generally easy-to-use; personal experience with the *Unity*[1] engine was also mostly positive, but Godot personally provided a better experience as a result of features such as its easy-to-use messaging system and the ease of writing decoupled code (i.e., where code does not depend on code from other files unless it is necessary).

Some issues were experienced when creating levels – the state of the 2D scene editor in Godot is not ideal with regard to tile-based levels due to lacking features such as moving groups of tiles. The external level editor *Tiled*[2] was tested along with an extension which allowed importing the files exported from Tiled into Godot, but this process did not work immediately and was postponed indefinitely; however, if further levels were to be added into the game, integration between Tiled and Godot would be of high priority.

Comparing *Blender* to other 3D modelling tools is not currently possible due to a lack of experience with anything else, aside from tests of smaller tools such as *Dust3D*. Blender is a powerful tool capable of complex animation and modelling, but only a small subset of its features were used in this project, as mentioned previously. Blender is capable of generating animation which would be extremely time-consuming to draw manually – a basic example is rotation around the vertical axis, which cannot be done by rotating a two-dimensional image.

The creation of music was the only area of development during which the use

[1]`https://unity.com/`

[2]`https://mapeditor.org/`

of strictly open-source tools was somewhat unfortunate – as mentioned in chapter 5, *Bosca Ceoil* could not be used to create music for the game as it was reliant on closed-source software, which partially caused the creation of custom music to be postponed until all higher-priority features had been implemented.

7.2.2 General open-source game development

For game developers who intend on publishing their game commercially, the notion of publishing the source code for free may seem to be a poor choice, but open-source games do not need to distribute non-code assets for free; for example, a game can be sold in its compiled form with the assets separate from the executable (possibly in addition to those same assets bundled within the executable, depending on the functionality of the tool used to generate the executable) and those who have purchased the game may choose to compile the game themselves using the open-source code and the proprietary assets. Of course, this may prevent developers from hiding content in their games (e.g., secrets and/or details related to narrative); prevents any form of piracy prevention from being implemented, since dedicated employers of piracy may be able to modify the source code to remove them; allows others to clone one's game and release new assets; and lets malicious individuals in countries with loose copyright laws (or in any country other than the developer's if the developer does not have enough money to sue them) and illegally publish the game without changes. If any of these points is a serious worry for a developer, open-source or otherwise source-available software may not be appropriate, although the second scenario can be removed as a potential risk through the use of a non-commercial, source-available license. Similar issues may occur even if only the *concept* for a game is publicly announced, or a prototype is released, especially if the game in question is relatively simple – this happened for game development studio *Vlambeer*'s *Ridiculous Fishing* – but releasing source code, especially under an open-source license, undoubtedly *facilitates* cloning.

However, open-source game development does not only cause problems – it also brings new possibilities, such as potentially causing volunteers to contribute to the game for free, encouraging players to create problem descriptions (e.g., *issues* in a *GitHub* repository[1]) when they find apparent bugs, allowing those who are unwilling to install closed-source software on their computers to play the game, increasing the amount of trust and good will towards the developers, and helping future game development projects by contributing to the open-source community. A non-open-source, source-available license may also bring these benefits provided that the game's potential community is willing to accept the license's restrictions. These positive and negative aspects cause the perceived feasibility of releasing software as open-source or otherwise source-available to vary between projects and individuals.

In the case of a non-commercial game such as *CheckerSphere*, releasing the source code under an open-source license has no apparent negative consequences unless the developer simply does not want to let others see or use their source code (in which case no source-available license will fit). An alternative business model for those who wish to create open-source or source-available games but still earn money through

[1] `https://guides.github.com/features/issues/`

them is to request donations.

For expensive games created by large teams, the feasibility of using purely open-source tools and/or distributing one's game under an open-source license will vary depending on the project's requirements. For example, the latest technology with regard to drivers and algorithms for highly demanding visuals may be closed-source and/or require closed-source drivers; in such a case, using only open-source tools may not be an option. As another example, a large publisher may not be willing to let its developers release code under an open-source license, or require that they sacrifice something significant (e.g., part of their salary) in order to do so, in which case open-source distribution may not be considered feasible. However, in many other cases, open-source alternatives exist – as shown in chapter 2, 3D game engines, modelling tools, audio workstations and other types of software are available under open-source licenses, and this project has shown that it is undoubtedly feasible to, at the very least, create a 2D platformer without using closed-source tools (with the possible exception of drivers and hardware, depending on the hardware used), and the feasibility of distributing code under an open-source license simply depends on the openness or fears of the individuals in charge of such matters.

7.3 Ethical issues

7.3.1 Accessibility

When developing any video game, issues of accessibility become apparent. Some are typically relatively easy to solve (e.g., color blindness, problems with hearing or missing fingers) unless the game's design principles require that such problems are not present (e.g., a stealth-focused game which relies on hearing footsteps may require clear visual hints in order to allow those who cannot hear them to play the game effectively). Others may require more attention or physical solutions, such as dyslexia, problems with memory or severe motor impairments. In this project, disabilities which were relatively easy to account for were considered, whereas problems which would be difficult to solve were disregarded in the interest of time; if the prototype resulting from this project were to be developed into a commercially released game, more time would have been spent on accessibility and, for problems where no software-based solution could be feasibly implemented by a single person, alternative solutions such as the *Xbox Adaptive Controller*[1], would be explicitly recommended for potential players. Ideally, players with disabilities should be involved in testing in order to ensure that one's implementation is actually good enough for such players.

Whether disregarding a subset of one's potential players due to disabilities is considered *unethical* or not is arguable, and it may not be reasonable to expect developers, especially individual developers, to be able to cater to every type of disability; however, from a commercial viewpoint, it can certainly be beneficial to implement solutions to the most common accessibility-related problems in order to increase the number of sales and garner support from those who may have been

[1]`https://microsoft.com/en-us/p/xbox-adaptive-controller/8nsdbhz1n3d8?activetab=pivot%3aoverviewtab`

unable to play the game without these solutions.

7.3.2 Privacy

If a game uses network communication for any purpose, players should be allowed to know what the game is sending and the game should not send any information which could compromise players' privacy unless they give explicit permission through an in-game interface while being fully aware of the information being sent. This includes personally identifying information such as locale settings and the name of the user as entered into the operating system, and it also includes information such as a name and score uploaded to an online scoreboard – even if it appears to be obvious that such information would be sent, the user should be made concretely aware of everything that is sent in order for the game to be completely ethical in terms of privacy. Additionally, due to the European *General Data Protection Regulation*[1] of 2016, privacy is now also a *legal* requirement when handling the data of any person within the European Economic Area [53].

Source-available software can still have issues with privacy, both unintentional (i.e., bugs) and intentional; for example, a source-available communication client can claim to be *encrypted* without having *end-to-end* encryption (i.e., being encrypted by the sender and only being decrypted by the receiver), and those who do not contribute to the project may never find out about this security flaw. However, source-available software enables independent auditing of the code and allows users to remove any parts of the software which they consider to infringe on their privacy; additionally, if the code is also *open-source* and encourages contributions, otherwise unrelated individuals may contribute to the project, which increases the chance of security flaws being discovered – the more contributors and fewer lines of code a project has, the easier it is, generally speaking, to examine whether specific features work as intended or expected.

Similarly, *telemetry* is impossible to hide from a dedicated user in a source-available project, but developers who want to include telemetry as part of their software may need to consider various points, such as the following from the article *Six questions to answer before implementing a telemetry feature* [54]:

- Why is the data being collected, and is it necessary to collect it?
- What kind of data is being collected, and will the data provided by the collection process justify its intrusiveness in the eyes of the users?
- Is the data collected truly anonymous?
- How will the data be collected (or, if a third party's telemetry service is used, where will they handle the data)?
- Is the data open? If not, who has access to it?
- Is telemetry opt-in or opt-out? Opt-in is more privacy-friendly but may result in significantly less data than opt-out.
- When will data be deleted after becoming less relevant?

[1] `https://gdpr-info.eu/`

7.3.3 Digital rights management

Digital rights management (*DRM*) is the “protection of copyrighted works by various means meant to control or prevent digital copies from being shared over computer networks or telecommunications networks” [55]. For video games, this may consist of requiring players to connect to the Internet to verify their purchase whenever they want to play a particular game. This is arguably a question of ethics, especially if it punishes legitimate customers while rewarding illegitimate ones who bypass it, legally or illegally. A severe example of DRM is the type which requires a constant or near-constant Internet connection – this may be effective in terms of preventing piracy, but it punishes those who have connection problems, have limited bandwidth or live without consistent Internet access. DRM is not possible to enforce in source-available software since users are free to remove unwanted pieces of code.

7.3.4 Availability of source code

The availability of the source code of any piece of software is partially a question of ethics due to factors such as the relative ease of hiding malicious instructions in a closed-source program (as opposed to an open-source or source-available program). Other aspects of open-source code which involve ethics include the denial of code reuse inherent to closed-source software, which forces other developers to spend time solving problems which have already been solved; large companies using permissively licensed code and not providing any contributions to the open-source community in return (which can be avoided by using *copyleft* licenses such as the *GNU General Public License* [48]); and using open-source code for malicious purposes (e.g., when the United States’ *Immigration and Customs Enforcement* used open-source code as part of operations involving separating children from their parents by the country’s border [56]).

7.3.5 Miscellaneous

Other issues which are arguably related to ethics include not spreading misinformation or hate speech – unless it is part of a game’s narrative – and ensuring that the player is aware of content which may be considered inappropriate for children. Additionally, political and religious propaganda are perhaps best not included in a game, although this may be a question of alienating users as opposed to being ethically problematic.

8 Conclusion

8.1 The game

This project involved the creation 2D side-scrolling platformer titled *Checker-Sphere*. The game places great focus on high-speed movement and implements many accessibility-improving features, including configurable input, the ability to adjust the volume of individual pieces of music or sound effects, and the ability to slow down time. The game contains four types of obstacles, one power-up and seven short levels with slightly branching paths. The level of difficulty can be adjusted dynamically by slowing down time, moving less quickly or taking easier routes through the levels. The game was developed using an iterative development process with nine primary user tests (spread evenly among three participants) and two smaller tests; additionally, various prototypes were distributed online in order to gain further feedback.

The game was developed using only open-source tools – primarily, the *Godot* engine and the 3D modelling tool *Blender*. Discussions on open-source game development, and lists of relevant tools and resources, can be primarily found in chapter 2 and 7.

The movement systems of three different games (*Mega Man X*, *Super Meat Boy* and *New Super Mario Bros. 2*) were analyzed; the results of the analyses can be seen in chapter 6 and appendix B. In addition, several lists of guidelines for future independent game developers were created, as shown in chapter 6.

Future work for the game may include the following:

- Adding further content.
- Updating the visuals to be more appealing.
- Adding at least one piece of music created specifically for the game.
- Implementing non-abrupt transitions in terms of both visuals and audio between different screens.
- Adding online leaderboards; this was not implemented partially due to potential issues with the GDPR which needed to be researched, and partially because it appeared to require significant time to implement.
- Adding further accessibility settings, such as being able to change the player's sound effects.
- Adding a level editor.
- Optimizing unnecessarily performance-heavy code.
- Refactoring some of the less well-written parts of the code.
- Adding support for non-keyboard controllers, with icons representing the buttons of different types of controller.

- Reworking the *wall jumping* and *wall sliding* systems.
- Adjusting the normal *jumping* system to allow for greater control with regard to the height of a jump.

8.2 Research questions

The project's research questions were the following:

As an individual developer using strictly open-source tools, what problems may appear – particularly in terms of game design, implementation, creation of visuals and creation of audio – during development of a side-scrolling, 2D platform game, and how can these problems be solved?

Both questions are somewhat vague due to the use of the words "may" and "can", and there is no pretense that *all* problems and potential solutions will be presented; therefore, the length of the answers can vary from extremely short (e.g., "an example of a problem is finding music for a game, and a possible solution is to use repositories of freely available music such as `freesound.org`") to an entire report such as this one (or one of greater scope). This report presents problems that occurred during the development of *CheckerSphere* and how they were solved, as well as issues that may occur in other problems and potential solutions; therefore, while there is certainly room for improvement and expansion, this project has undoubtedly answered the research questions, albeit in a manner limited to a project of relatively small scope.

8.3 Contribution of knowledge

This thesis is largely a case study of individual, open-source game development, primarily presented in chapters 4 to 6; due to the lack of rigorous logging, the thesis is not as granular as may be desired, but it nevertheless provides information regarding the development process for *CheckerSphere*, which may be of use for future developers with similar circumstances.

Chapter 6 also contains a set of guidelines, briefly summarized below:

- If you have a particular shortcoming, try to find an alternative approach – for example, if you lack the skills to draw detailed two-dimensional art and do not believe you can learn to do so within a reasonable amount of time, try 3D modelling or using a very simple visual style.
- Analyze successful games in your chosen genre to find guidelines and inspiration for design.
- Ensure that you understand the license of every asset you use, including code, and that the license's restrictions are acceptable for your project's purpose.
- Be careful when using existing non-code assets in order to prevent your game from being regarded as an *asset flip*. This does not mean that a large amount of such assets cannot be present in your game; however, you may need to apply creative modifications to those assets (e.g., rendering shaders and textures) in order to make them fit with your game's visual style and seem less generic.

- The use of 3D models, or rendered 2D versions of such models, is relevant for both 3D and 2D games, and repositories of freely available 3D models should not be discounted as sources of visual assets for 2D games.
- Changes in terms of visuals and/or audio, even seemingly minor ones, can have surprising impact on the enjoyment experienced by players.
- When using visual effects, be careful not to overwhelm the player or create a bottleneck with regard to performance.
- When creating or finding sound effects to use in your game, be considerate to the users – for example, rapidly repeating a single sound effect without changes in pitch may be obnoxious, and high-pitched sounds may disturb certain players.

Finally, chapters 1 and 2 provide an introduction to the topic of the thesis, and chapter 7 contains discussions of various issues related to individual, open-source game development.

This project has personally strengthened the view that individual, open-source game development is entirely feasible when extreme graphical fidelity is not required, both in terms of using only open-source tools and in terms of distributing the game under an open-source license; the latter does not prevent the developer from releasing the game commercially, since non-code assets may remain proprietary. Whether this applies to very expensive games made by enormous game development studios remains to be seen; however, at the very least, this project has shown that it is entirely feasible for a simple 2D platformer.

Bibliography

[1] H. E. Lowood. (Mar. 1, 2019). “Electronic game,” [Online]. Available: https://www.britannica.com/topic/electronic-game (visited on 03/24/2020).

[2] E. Adams. (Jul. 9, 2009). “The Designer’s Notebook: Sorting Out the Genre Muddle,” [Online]. Available: https://www.gamasutra.com/view/feature/132463/the_designers_notebook_sorting_.php?page=2 (visited on 04/09/2020).

[3] K. T. Jensen. (Oct. 25, 2018). “Run, Jump, and Climb: The Complete History of Platform Games,” [Online]. Available: https://www.geek.com/games/run-jump-and-climb-the-complete-history-of-platform-games-1748896/ (visited on 12/02/2019).

[4] J. Bailey. (Nov. 9, 2017). “Asset Flipping: The Ethics of Reuse in Video Games,” [Online]. Available: https://www.plagiarismtoday.com/2017/11/09/asset-flipping-ethics-reuse-video-games/ (visited on 02/13/2020).

[5] M. Handrahan. (Jun. 18, 2018). “PUBG Corp. defends the use of asset stores as the only way to "work smart",” [Online]. Available: https://www.gamesindustry.biz/articles/2018-06-18-pubg-corp-defends-the-use-of-asset-stores-as-the-only-way-to-work-smart (visited on 02/13/2020).

[6] Thomas Happ Games LLC. (Mar. 14, 2016). “Award-winning Axiom Verge Announced for Xbox One and Wii U,” [Online]. Available: https://www.axiomverge.com/press-releases (visited on 11/23/2019).

[7] ——, “Developer – Axiom Verge,” [Online]. Available: https://www.axiomverge.com/author (visited on 11/23/2019).

[8] ConcernedApe LLC. “Stardew Valley - FAQ,” [Online]. Available: https://www.stardewvalley.net/faq/ (visited on 11/23/2019).

[9] Mojang. “What is Minecraft?” [Online]. Available: https://www.minecraft.net/en-us/what-is-minecraft/ (visited on 11/23/2019).

[10] Markus Persson. (Jul. 31, 2012). “I am Markus Persson aka Notch, Creator of Minecraft - Ask me Anything!” [Online]. Available: https://www.reddit.com/r/Minecraft/comments/xfzdg/i_am_markus_persson_aka_notch_creator_of/ (visited on 11/23/2019).

[11] The Open Source Initiative. “Licenses & Standards,” [Online]. Available: https://opensource.org/licenses (visited on 03/25/2020).

[12] ——, “Frequently Answered Questions,” [Online]. Available: https://opensource.org/faq (visited on 04/28/2020).

[13] University of New England. “Grey literature,” [Online]. Available: https://www.une.edu.au/library/support/eskills-plus/research-skills/grey-literature (visited on 03/25/2020).

[14] Academy of Interactive Arts & Sciences. (Dec. 17, 2019). “Lucas Pope’s Return of the Obra Dinn - The AIAS Game Maker’s Notebook,” [Online]. Available: `https://youtube.com/watch?v=quNbkpzhx1E` (visited on 02/10/2020).

[15] The Editors of Encyclopaedia Britannica. (Dec. 27, 2017). “Auteur theory,” [Online]. Available: `https://www.britannica.com/technology/raster-graphics` (visited on 04/29/2020).

[16] M. Hetfeld. (Aug. 20, 2018). “Auteur Theory and Games,” [Online]. Available: `https://unwinnable.com/2018/08/20/auteur-theory-and-games/` (visited on 04/29/2020).

[17] A. Donnelly, “Auteur theory in video games: recognizing Hideo Kojima and Thatgamecompany as auteurs in the video game medium,” Ball State University, Jul. 21, 2018.

[18] FOSSA. “The Commons Clause,” [Online]. Available: `https://commonsclause.com/` (visited on 04/29/2020).

[19] R. Gardler and R. Wilson. (Aug. 2, 2012). “Contributor Licence Agreements,” [Online]. Available: `http://oss-watch.ac.uk/resources/cla` (visited on 05/11/2020).

[20] W. Scacchi, “Free and Open Source Development Practices in the Game Community,” *IEEE Software*, vol. 21, issue 1, Jan. 2004. DOI: `10.1109/MS.2004.1259221`.

[21] “How Is Video Game Development Different from Software Development in Open Source?” In *Mining Software Repositories*, (Gothenburg, Sweden), May 2018. DOI: `10.1145/3196398.3196418`.

[22] Game accessibility guidelines. “Basic,” [Online]. Available: `http://gameaccessibilityguidelines.com/basic/` (visited on 04/01/2020).

[23] ——, “Intermediate,” [Online]. Available: `http://gameaccessibilityguidelines.com/intermediate/` (visited on 04/01/2020).

[24] ——, “Advanced,” [Online]. Available: `http://gameaccessibilityguidelines.com/advanced/` (visited on 04/01/2020).

[25] The Linux Foundation. “What Is Linux?” [Online]. Available: `https://www.linux.com/what-is-linux/` (visited on 04/01/2020).

[26] P. H. Salus, *The Daemon, the Gnu, and the Penguin.* May 5, 2005, ch. 7. [Online]. Available: `http://www.groklaw.net/article.php?story=20050505095249230` (visited on 04/01/2020).

[27] “What is Wine?” [Online]. Available: `https://www.winehq.org/` (visited on 04/01/2020).

[28] ReactOS Team & Contributors. “Front Page | ReactOS Project,” [Online]. Available: `https://reactos.org/` (visited on 04/03/2020).

[29] “ReactOS,” [Online]. Available: `https://reactos.org/wiki/ReactOS` (visited on 04/03/2020).

[30] J. Linietsky, A. Manzur, and contributors. “Godot Engine - Features,” [Online]. Available: `https://godotengine.org/features` (visited on 04/01/2020).

[31] The Editors of Encyclopaedia Britannica. (Jan. 28, 2020). “Raster graphics,” [Online]. Available: `https://www.britannica.com/technology/raster-graphics` (visited on 04/01/2020).

[32] ——, (Feb. 21, 2019). “Vector graphics,” [Online]. Available: `https://www.britannica.com/technology/vector-graphics` (visited on 04/01/2020).

[33] increpare. “Bfxr. Make sound effects for your games.,” [Online]. Available: `https://www.bfxr.net/` (visited on 04/09/2020).

[34] ttencate. “jfxr,” [Online]. Available: `https://github.com/ttencate/jfxr` (visited on 04/09/2020).

[35] (Jan. 11, 2017). “Integrated Development Environment (IDE),” [Online]. Available: `https://www.techopedia.com/definition/26860/integrated-development-environment-ide` (visited on 04/09/2020).

[36] A. Nyström, “Agile Solo – Defining and Evaluating an Agile Software Development Process for a Single Software Developer,” Chalmers University of Technology, Jun. 2011.

[37] B. Martin and B. Hanington, *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Quarto Publishing Group USA, Feb. 1, 2012, ISBN: 9781610581998.

[38] The Interaction Design Foundation. “What is User Centered Design?” [Online]. Available: `https://www.interaction-design.org/literature/topics/user-centered-design` (visited on 03/31/2020).

[39] R. Hunicke, M. LeBlanc, and R. Zubek, “MDA: A Formal Approach to Game Design and Game Research,” Jan. 2004.

[40] P. Lankoski, S. Björk, *et al.*, *Game Research Methods*. ETC Press, 2015, pp. 25–26, ISBN: 9781312884731.

[41] The Blender Foundation. “About,” [Online]. Available: `https://www.blender.org/about/` (visited on 04/20/2020).

[42] A. Novell. (Feb. 6, 2020). “I Had to Abandon My Game After 4 Years And It Nearly Broke Me,” [Online]. Available: `https://me.ign.com/en/pc/170093/feature/what-it-feels-like-to-abandon-your-game-after-4-years-of-hard-work` (visited on 04/30/2019).

[43] Creative Commons. “Attribution 4.0 International (CC BY 4.0),” [Online]. Available: `https://creativecommons.org/licenses/by/4.0/` (visited on 12/01/2019).

[44] D. Bridges, “Writing a research paper: reflections on a reflective log,” *Educational Action Research*, vol. 7, issue 2, pp. 221–234, 1999. DOI: `10.1080/09650799900200084`.

[45] K. Compton and M. Mateas, “Procedural Level Design for Platform Games,” in *The Second Artificial Intelligence and Interactive Digital Entertainment Conference*, (Marina del Rey, CA, USA, Jun. 20–23, 2006), pp. 109–111, ISBN: 978-1-57735-235-8.

[46] M. Cook, S. Colton, and A. Pease, “Aesthetic Considerations for Automated Platformer Design,” in *The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, (Stanford, CA, USA, Oct. 8–12, 2012), pp. 124–129.

[47] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, “Rhythm-Based Level Generation for 2D Platformers,” in *The 4th International Conference on Foundations of Digital Games*, (Orlando, FL, USA, Apr. 26–30, 2009), pp. 175–182. DOI: `10.1145/1536513.1536548`.

[48] The Free Software Foundation. (Jun. 29, 2007). “GNU General Public License,” [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 02/13/2020).

[49] R. Valentine. (Jul. 19, 2018). “Unity: "Games wouldn’t see the light of day" without asset stores,” [Online]. Available: https://www.gamesindustry.biz/articles/2018-07-19-well-88-percent-of-what-asks-unitys-global-head-of-asset-store (visited on 05/05/2020).

[50] T. Vasseur. (Jan. 25, 2018). “Art Design Deep Dive: Using a 3D pipeline for 2D animation in Dead Cells,” [Online]. Available: https://www.gamasutra.com/view/news/313026/Art_Design_Deep_Dive_Using_a_3D_pipeline_for_2D_animation_in_Dead_Cells.php (visited on 05/04/2020).

[51] C. Ericson. (Jan. 2, 2009). “Optimizing the rendering of a particle system,” [Online]. Available: https://realtimecollisiondetection.net/blog/?p=91 (visited on 05/04/2020).

[52] M. McWhertor. (Nov. 30, 2012). “The Witness includes more than 1,100 footstep sound effects,” [Online]. Available: https://www.polygon.com/2012/11/30/3712542/the-witness-includes-more-than-1100-footstep-sound-effects (visited on 05/04/2020).

[53] J. Uzan-Naulin. (Nov. 28, 2019). “The (Extra) Territorial Scope of the GDPR: The Right to Be Forgotten,” [Online]. Available: https://www.fasken.com/en/knowledge/2019/11/the-extra-territorial-scope-of-the-gdpr/ (visited on 03/28/2020).

[54] C. Stransky. (Apr. 14, 2020). “Six questions to answer before implementing a telemetry feature,” [Online]. Available: https://dev.to/meeshkan/six-questions-to-answer-before-implementing-a-telemetry-feature-1c5i (visited on 05/04/2020).

[55] The Editors of Encyclopaedia Britannica. (Jan. 24, 2020). “Digital rights management,” [Online]. Available: https://www.britannica.com/topic/digital-rights-management (visited on 03/28/2020).

[56] J. Cox. (Sep. 20, 2019). “’Everyone Should Have a Moral Code’ Says Developer Who Deleted Code Sold to ICE,” [Online]. Available: https://www.vice.com/en_us/article/mbm3xn/chef-sugar-author-deletes-code-sold-to-ice-immigration-customs-enforcement (visited on 02/13/2020).

A Resources

A.1 Study material

The following lists contain references to study material which was of use during this project (or was of limited use in this case, but may be useful for others).

The book *Game Feel: A Game Designer's Guide to Virtual Sensation*[1], by Steve Swink, semi-formally discusses how to make a game enjoyable; or, as a free alternative, the article *Game Feel: The Secret Ingredient*[2], on which the book is based.

The following video sources discuss gameplay design and/or level design:

- *Boss Up - Boss Battle Design Fundamentals and Retrospective*[3], by Itay Keren.
- *Empowering the Player - Level Design in N++*[4], by Mare Sheppard and Raigan Burns.
- *Game Feel - Measuring the Influence of Acceleration and Deceleration*[5], by Gustav Dahl.
- *Level Design in a Day: Decisions That Matter - Meaningful Choice in Game and Level Design*[6], by Matthias Worch.
- *Level Design Workshop - Designing Celeste*[7], by Matt Thorson.
- *The Marriage of Level Design and Controls*[8], by Tommy Refenes.
- *Ten Principles for Good Level Design*[9], by Dan Taylor.
- The *LevelHead* series on the YouTube channel *Sunder*[10].
- *How Mega Man 11's Levels Do More With Less*[11], *Super Mario 3D World's 4 Step Level Design*[12], *The Secret of Mario's Jump (and other Versatile Verbs)*[13], *Why Does Celeste Feel So Good to Play?*[14] and various other video essays by Mark Brown[15].

[1] http://game-feel.com/
[2] https://gamasutra.com/view/feature/130734/game_feel_the_secret_ingredient.php
[3] https://gdcvault.com/play/1024921/Boss-Up-Boss-Battle-Design
[4] https://gdcvault.com/play/1023282/Empowering-the-Player-Level-Design
[5] https://youtube.com/watch?v=S-EmAitPYg8
[6] https://gdcvault.com/play/1020570/Level-Design-in-a-Day
[7] https://gdcvault.com/play/1024307/Level-Design-Workshop-Designing-Celeste
[8] https://youtube.com/watch?v=O4TmH8WG7_M
[9] https://gdcvault.com/play/1017803/Ten-Principles-for-Good-Level
[10] https://youtube.com/channel/UCYwIcyCwXZ--FR0iZ593uBA
[11] https://youtube.com/watch?v=nYxHMZX6lN8
[12] https://youtube.com/watch?v=dBmIkEvEBtA
[13] https://youtube.com/watch?v=7daTGyVZ60I
[14] https://youtube.com/watch?v=yorTG9at90g
[15] https://youtube.com/user/McBacon1337

The following video sources demonstrate how to add polish to a game:

- *The art of screenshake*[1], by Jan Willem Nijman.
- *Juice it or lose it*[2], by Martin Jonasson and Petri Purho.
- *Game Feel: Why Your Death Animation Sucks*[3], by Nicolae Berbece.
- *Oh My! That Sound Made the Game Feel Better!*[4], by Joonas Turner.

The following video sources discuss animation:

- *Animation Bootcamp: An Indie Approach to Procedural Animation*[5], by David Rosen.
- *Animation Bootcamp: 'Rainworld' Animation Process*[6], by Joar Jakobsson and James Therrien.
- *IK Rig - Procedural Pose Animation*[7], by Alexander Bereznyak.

The following video sources discuss the use of mathematics in games:

- *Math for Game Programmers: Building A Better Jump*[8], by Kyle Pittman.
- *Math for Game Programmers: Fast and Funky 1D Nonlinear Transformations*[9] and *Math for Game Programmers: Juicing Your Cameras With Math*[10], by Brian "Squirrel" Eiserloh.

Finally, the following video sources discuss miscellaneous subjects:

- *Automated Testing and Instant Replays in Retro City Rampage*[11], by Brian Provinciano.
- *Crafting A Tiny Open World: 'A Short Hike' Postmortem*[12], by Adam Robinson-Yu.
- *How Cameras in Side-Scrollers Work*[13], by Itay Keren.
- *Lessons Learned Making Gunpoint Quickly Without Going Mad*[14], by Tom Francis.
- *Making 'Night in the Woods' Better with Open Source*[15], by Jon Manning.
- *No More Excuses, Your Guide to Accessible Design*[16], by Tara Voelker.
- *No Time, No Budget, No Problem: Finishing The First Tree*[17], by David Wehle.
- *GuiltyGearXrd's Art Style : The X Factor Between 2D and 3D*[18], by Junya Christopher Motomura.

[1] `https://youtube.com/watch?v=AJdEqssNZ-U`
[2] `https://youtube.com/watch?v=Fy0aCDmgnxg`
[3] `https://gdcvault.com/play/1022759/Game-Feel-Why-Your-Death`
[4] `https://gdcvault.com/play/1022808/Oh-My-That-Sound-Made`
[5] `https://gdcvault.com/play/1020583/Animation-Bootcamp-An-Indie-Approach`
[6] `https://gdcvault.com/play/1023475/Animation-Bootcamp-Rainworld-Animation`
[7] `https://gdcvault.com/play/1023279/IK-Rig-Procedural-Pose`
[8] `https://gdcvault.com/play/1023559/Math-for-Game-Programmers-Building`
[9] `https://gdcvault.com/play/1022142/Math-for-Game-Programmers-Fast`
[10] `https://gdcvault.com/play/1023557/Math-for-Game-Programmers-Juicing`
[11] `https://gdcvault.com/play/1021825/Automated-Testing-and-Instant-Replays`
[12] `https://gdcvault.com/play/1026613/Independent-Games-Summit-Crafting-A`
[13] `https://youtube.com/watch?v=pdvCO97jOQk`
[14] `https://youtube.com/watch?v=aXTOUnzNo64`
[15] `https://gdcvault.com/play/1024197/Making-Night-in-the-Woods`
[16] `https://gdcvault.com/play/1022172/No-More-Excuses-Your-Guide`
[17] `https://youtube.com/watch?v=g5f7yixtQPc`
[18] `https://gdcvault.com/play/1022031/GuiltyGearXrd-s-Art-Style-The`

- *This is a Talk About Tutorials, Press "A" to Skip*[1], by Nicolae Berbece.
- *Thriving in Steam Early Access: Turning 20XX's Slow Launch into Success*[2], by Chris King.
- *Throwing Out the Dopamine Shots: Reward Psychology Without the Neurotrash*[3], by Ben Lewis-Evans.

A.2 Free assets

- 3D models: *SketchFab*, *BlendSwap*, *TurboSquid* and *3D Model Haven*.
- Textures: *CC0 Textures*.
- Sound effects: *freesound* and *kenney.nl*.
- Music: *freesound* and *incompetech* (Kevin MacLeod).
- Various: *OpenGameArt* and *itch.io*.

Additionally, various lists of assets and other resources for game development can be found in various Git repositories, such as `magictools`[4] and `awesome-gamedev`[5].

[1]`https://gdcvault.com/play/1023845/This-is-a-Talk-About`
[2]`https://gdcvault.com/play/1025693/Thriving-in-Steam-Early-Access`
[3]`https://gdcvault.com/play/1024181/Throwing-Out-the-Dopamine-Shots`
[4]`https://github.com/ellisonleao/magictools`
[5]`https://github.com/Calinou/awesome-gamedev`

B
Notes from the game analyses

B.1 Mega Man X

- The jumping height depends on how long the *jump* button is held.
- Dashing and dash jumping is very satisfying; it gives a lot of freedom of movement to the player and lets them set their own pace. You don't get the *dash* before playing through *Chill Penguin*'s stage, and the game feels very slow without it, to the point that Chill Penguin becomes the obvious first choice for a stage.
- You don't lose your *dash* ability when entering one of the "robots" (e.g., the ones in Chill Penguin's stage), which shows how integral it is to the gameplay – if *dashing* was disabled, I wouldn't want to enter the robot.
- A small number of stages with many different new enemies in each. Many enemies are unique to each stage (or reused in one of the final stages).
- Secrets increase replayability.
- Large levels with an increased focus on verticality.
- Uses an enemy or obstacle for a while and then moves on to something new to avoid unenjoyable repetition.
- *Dashing* kicks up dust and plays a sound effect.
- *Dashing* time depends on the amount of time the *dash* button is held.
- If *dash jumping*, the *dashing* velocity stays until the player lands.
- The speed when walking, *dashing* and airborne (normally or *dashing*) is constant. You can turn around immediately on the ground and in the air and keep the same speed, even when *dashing* in the air. You can immediately cancel a *dash* on the ground by turning around. (*Dashing* remains in the air, even when turning around.)
- A level usually has some reoccurring concept, like the lights turning on and off in *Spark Mandrill*'s stage.
- The ability to scale any wall by repeated *wall jumps* gives the player the freedom to be reckless without severe punishment.
- Levels are both vertical and horizontal (but mostly horizontal).

B.2 Super Meat Boy

- The player (as *Meat Boy*) covers about 1/34 of the screen horizontally, and 1/20 vertically. (This may vary depending on the resolution used.)
- The jumping height depends on how long the *jump* button is held.

- You stop almost instantly, but it takes a while to turn around while running, so it's faster to let go of left/right and then hold right/left immediately than to go directly from left/right to right/left.
- The running speed is very fast.
- You can scale any wall by wall jumping (like in MMX).
- Moving leaves "meat trails"; running leaves more. Touching any surface usually leaves trails. Trails stay after death (and death leaves more trails).
- Satisfying sound effects when moving (walking/running) and jumping.
- Only four keys used for gameplay: left, right, jump and run.
- Beating a level shows all attempts, which is satisfying for a platformer where later levels are likely to cause many deaths.
- Only platforming; you can't defeat any enemies, unlike in NSMB2 or MMX.
- Level design is heavily focused on fast movement and *wall jumping*.
- Getting A+ on a level (at least the ones I played) never felt too difficult.
- The *really* frustrating parts of a level were always limited to the bandages (entirely optional collectibles that unlock new characters), at least in the first few worlds.
- You can skip a few levels in every world if you find them too difficult.
- The *Dark World* is a great way to reuse the basic existing level designs but make them more difficult.
- Gradually introduces new mechanics and reuses old mechanics. Many obstacles are reused in similar or different situations in other levels. (Cost-effective for independent developers.)
- Always starts at the same point in a level when respawning, to preserve the flow through the level (unlike *VVVVVV*, for example).
- Levels are both vertical and horizontal (but mostly horizontal).

B.3 New Super Mario Bros. 2

- Mario is reasonably small: he covers about 0.5cm out of 8.5cm of the screen horizontally (5%), and about 0.5cm out of 5cm vertically (10%) in his "short form". (This varies slightly with different 3DS models.)
- The jumping height depends on how long the *jump* button is held. This introduces some more flexibility but is not ideal for people with certain motor problems (e.g., those who can't release their finger quickly enough to perform each of the different levels of jump).
- There does not appear to be any analog movement; this is good for accessibility.
- The player has two main states of movement: walking and running. The latter requires that the *run* button is held.
- The movement system only requires the four directional buttons (although usually only the left and right buttons are used), a *jump* button and a *run* button. Unfortunately, the game cannot be played effectively with one hand on the 3DS due to the inability to remap buttons, but it could have been (e.g., by using the left shoulder button to jump and allow toggling running rather than having to hold the *run* button).
- When running, the player's running speed accelerates for approximately one

second. (This also applies to the walking speed, but the difference in speed and the time to maximum speed is very short.) Running at maximum speed causes a clear change in animation.

- Moving on the ground causes dust (or something similar) to appear, which serves as a visual enhancement for movement.
- The controls are typically very responsive; on a few occasions, I felt as though the game did not register my press of the *jump* button, but this may, at least partially, be an issue with hardware. Turning left and right is very quick on the ground.
- Changing direction while moving (on the ground or in the air) is possible but takes a little while depending on the player's velocity. On the ground, if you want to change direction, it can take approximately two seconds at top running speed (depending on the terrain); it's faster and more reliable to jump and change direction in the air, which encourages jumping. The fastest way to stop and change direction is to hit a wall, either on the ground (which allows you to turn around immediately) or in the air (which allows you to wall jump), but this is not reliable.
- *Wall sliding* and *wall jumping* makes the game very easy on most levels if you move slowly enough; the real challenge comes from trying to beat a level as quickly as possible and finding all collectibles.
- Running is quick and levels are generally designed to let you run through them without stopping if you're good enough. However, there are some moments that force you to stop and turn back or wait, like the beginning of level 1-2.
- Level design is generally quite basic compared to many other Mario games, but the ability to speedrun, the three collectible coins in every level and the movement system make it enjoyable nonetheless.
- Camera: aside from at the beginning and end of an area (an entire level or a part of a level which is separate from the rest of the level), the camera moves as the player does and is always horizontally centered on the player, without appearing to lag behind or in front. Vertically, the camera changes, depending on the height of the area, when the player gains or loses a certain amount of elevation; I did not notice any obvious patterns for how the camera changed aside from making sure that it didn't show anything outside of the limits of the area and that Mario was always clearly visible with some room in every direction.
- Every level (at least in the first three worlds, possibly excluding the hidden levels) introduces something new (e.g., an enemy) but doesn't necessarily focus on it.
- Levels are both vertical and horizontal (but mostly horizontal).