You say jump, I say how high?

A study of jumping in 2D platform games

Written by Martin Fasterholdt

Supervisors: Christoffer Holmgård & Martin Pichlmair

IT University of Copenhagen, Denmark

December 2015

Abstract

This thesis will explore the design of jumping in 2D platform games. Designing movement is usually not a formalised process, but is instead based on intuition, testing, and lots of iterative development. This thesis presents tools and knowledge which can support game designers during this process.

An approach for measuring and analysing movement in existing games will be presented. The first step of the approach is to simulate input to a game otherwise running normally. To accomplish this a program was created which injects a DLL file and hooks into the DirectX callbacks of a game. This way the program can simulate a series of XBox controller input. This gives precise control of how the character in the game moves and jumps. The resulting movement is recorded using screen recording software. To measure this movement a measurement program was created. The recording is loaded into this program, the movement measured, and the data exported as a text file. The text file is imported into Microsoft Excel where the movement can be visualised and further analysed. Based on the data a model of the character's movement can be created. The model shows one potential way to implement the movement of the game and provides detailed information about the properties of the jump. A framework describing jumping was defined to structure the models. Three games are used as case studies throughout the project: *Super Meat Boy* (Team Meat, 2010), *Limbo* (Playdead, 2011) and *Super Mario Bros. 3* (Nintendo, 1988). They will be used to demonstrate the approach just described. The game feel of the jump in each case study will be analysed and discussed based on measurements of the movement in the game.

The framework can be used to describe jumping and be used as an overview of aspects to consider and provides a range of options which can be used as starting points when designing a jump. The case studies provide detailed information about the implementation of the games. Instead of starting from scratch a game designer can

consider the solutions from the case studies. The case studies also suggest how a change to a jump might influence the feel of the jump. The tools developed, the approach defined, and the framework can be used by game designers to measure, model and analyse movement in platform games. A general approach to this is valuable as it allows game designers to conduct their own case studies with games particularly relevant for their production. A prototyping tool was created for this project. This tool is a basic 2D platform game where the character's movement can be adjusted with a wide range of parameters. The tool can visualise a character's jump while parameters are being adjusted. The basic movement of the characters from the three case studies were replicated inside this tool. The prototyping tool can be used by game designers to explore and experiment with the different implementation.

Acknowledgements

Thanks to my supervisors Christoffer Holmgård and Martin Pichlmair for their guidance and feedback throughout the project. Thanks to Mads Johansen Lassen for his help during the development of the program simulating input. Thanks to Tim Garbos for convincing me to make a program for measuring movement instead of using photoshop. Thanks to Justin Stenning for his example project and his articles. Thanks to Jonathan Aldrich for sharing his research about the physics in *Mario* games. Thanks to the developers of EasyHook and SharpDX for providing free software.

Contents

Abstract **2**

Acknowledgements **3**

Introduction **6**

Context **8**

Game Feel **15**

Real-Time Control 16

Simulated Space 18

Polish 19

Case Studies **20**

Measuring Movement **23**

Simulating Input 23

Recording 30

Measurement Tool 31

Analysing Movement **35**

Organising Data 35

Considerations 38

Analysis **41**

Prototyping Tool 48

Movement Framework **53**

Input 56

Ground Movement 65

Jump Takeoff 71

Air Control 78

Jump Release 86

Details 92

Summarize and Reflect **96**
Super Meat Boy 96
Mario 100
Limbo 108
General Reflection 115
Future work 119
Conclusion **120**
Appendix 1, Limbo Input **124**
Appendix 2, Attacthments **126**
References **127**

Introduction

Curiosity regarding jumping in games inspired this thesis. What are the differences between the ways Mario and Meat Boy jump? The two characters clearly have a different pace, but functionally they appear similar. Are they, perhaps, technically the same just with parameters tweaked differently?

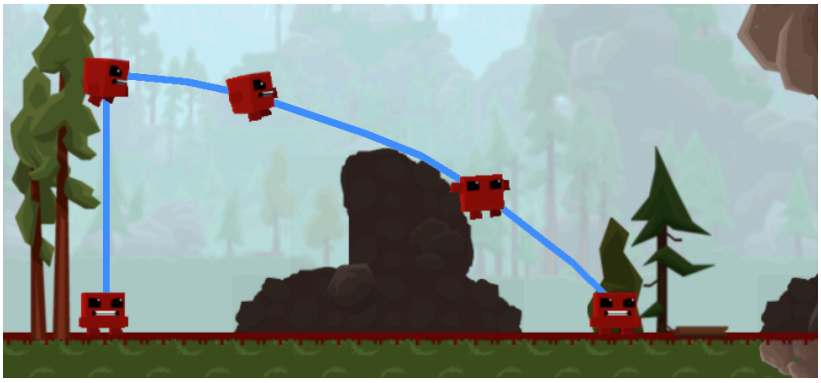


Figure 1: Jump with variable air control in Super Meat Boy (Team Meat, 2010).

Playing *Super Meat Boy* (Team Meat, 2010) while focusing on the details of his movement gave rise to more questions. Figure 1 shows the path Meat Boy takes during a jump. For this particular jump the character is initially standing still. I initiate a jump and Meat Boy jumps upwards. As soon as Meat Boy is off the ground I give continuous input to move right. Surprisingly, the input appears to have no effect and the character moves straight up. Only after reaching the peak of the jump does the input take effect and Meat Boy moves to the right while he descends. This behaviour was unexpected and indicates that how much control the player has in the air varies during a jump. However, this conclusion is based on playing the game and intuitively feeling how the character moves. Precisely estimating position, speed, and acceleration in this manner is difficult. How close to the actual movement of the character are the assumptions? Furthermore, if a close look was required to notice a variable air control what other subtle details might have been missed? To answer these and similar questions requires some way of learning more about movement in games.

Moving a character through a game world is a fundamental interaction at the core of many digital games. Getting this connection between player intention and resulting movement just right can make or break the experience of playing a game. Most aspects of a game have the potential to influence the feel of movement. This project will focus on the relationship between player input and the resulting movement of the game character. This means leaving out visual representation of the game world as well as audio and haptic feedback. Further narrowing the scope, the thesis will look at 2D platform games and exclusively on jumping and running left and right, which will often be referred to as the movement of the character.

The genre of platform games can be defined as video games in which the character jumps from platform to platform. A jump, to push oneself off a surface and into the air, is usually the primary and defining action, performed repeatedly in this type of games. Designing movement for this genre is usually not a formalised process, but is instead based on intuition, testing, and lots of iterative development. The aim of this project is to support game designers during this process. In the following section, the question of how to provide this support will be approached by reviewing related work about jumping.

Context

This chapter will present related work with different perspectives on the topic of jumping in platform games. This will give a broad introduction to the concept of jumping while contextualising the project.

One approach to contextualise jumping is to look at the history of jumping. Tom Butler (2014) provides an overview of jumps through the history of video games. This overview begins with *Donkey Kong* (Nintendo, 1981) which established the platform genre and was one of the earliest games to feature a jumping character. *Super Mario Bros.* (Nintendo, 1985) expands on this jump and is cited as the first game allowing the player to control the height and distance of the jump. Since then, jumping has been used in countless games and been adjusted and remixed. One example is *Braid* (Number None, Inc., 2009) where the player can manipulate the flow of time, while jumping through a series of puzzles. The game pays homage to the *Mario* series with references to enemies and parts of the level design. Jumping continues to be an essential part of today's 2D platform games. Two examples of games released this year are *Ori and the Blind Forest* (Moon Studios, 2015) and *Feist* (Bits & Beasts, 2015). A different historical approach is to look at how parameters of jumping has developed over time. This can for example be seen in the work of Lefky & Gindin (2007), who explore how gravity has developed in *Mario* games over time. According to their research the gravity has consistently gone down. It appears to go towards the gravity we see on earth, it is however still significantly higher and humans would likely not survive under this much gravity.

Figure 2: Donkey Kong (Nintendo, 1981) established the genre of platform games.

Another example is the exploration of how design patterns have been reused and evolved across different *Mario* games (Thompson, 2015). This thesis will not further explore jumping from a historical perspective. However, the approach and tools created for this project can be used to make similar observations, and could as such support this kind of research.

Jason Begy (2010) questions why jumping in general is so common in video games as well as board games. He narrows his focus to the dominating jump, which he defines as a jump used not only as means of locomotion, but to overcome or dominate an enemy or spatial obstacle. He presents examples of this jump in board games across hundred of years, as well as platform games such as *Sonic the HedgeHog* (Sonic Team, 1991), *LittleBigPlanet* (Media Molecule, 2008), and the *Mario* series. Begy argues that the appeal and popularity of this dominating jump comes from the orientational metaphor of GOOD IS UP. Very simplified this implies that the action of moving upwards, above or onto obstacles has a universal appeal. The appeal of the jump is similarly explored by Warren (2014). He suggests that we enjoy jumping in video games because in reality we rarely jump and when we do our abilities are limited. When it comes to jumping the human body is far from ideal compared to animals. In reality humans can jump about half their own height and have very poor control while in the air. Compare that to jumping in the *Mario* series, where the player can jump four or five times the height of the character and remain in control while in the air. Experiencing this superhuman jump capability, according to Warren (2014) is liberating and enjoyable. These two examples offer broad perspectives on the cultural significance of the jump. In contrast, this project will take a practical and more focused approach.

Several authors have written about the feel of jumping in games. For example, Tim Rogers who in his article *In Praise Of Sticky Friction* (Rogers, 2010) argues that Mario owes most of his fame to sticky friction, which in short is Mario's inertia and feeling of weight. Rogers has also written an article entirely about jumping titled *Let's Talk About Jumping* (Rogers, 2009). In this article he talks at length about the significance of jumping

and how it has developed over time, as well as how jumping in different games feel very differently. Swink (2008) also makes a number of case studies where he analyses the jump and game feel in for example *Super Mario Bros.* and *Bionic Commando* (Capcom, 1988). These and similar analyses provide valuable reflections on game design. Part of this project will consist of similar cases studies, but takes a different approach. The analysis presented will similarly discuss the game feel of different games, but arguments and claims will be based on measurements of the movement. This will give information about how each part of the implementation influences game feel. This will also provide generalised knowledge in terms of what to adjust to achieve a certain feel found in a particular game.

A technical way to approach jumping would be to look at how it is implemented. There are many articles providing knowledge on this topic. *The guide to implementing 2D platformers* (Monteiro, 2012) presents a variety of different ways to implement a platform game. The guide begins with the purely tile-based approach seen in *Flashback* (Delphine Software, 1992) and *Prince of Persia* (Brøderbund, 1989) and ends with a vectorial approach based on physics engines seen in games like *Braid* or *Limbo* (Playdead, 2011). Another type of articles are tutorials which go through the implementation process step by step. Tutorials usually also provide the source code free of charge. Yoann Pignole (2013, 2014) has published several of these, some specifically about 2D platformers . One of his tutorials focuses on collision between the character and the environment and presents a solution using raycasting (Pignole, 2013). Another focuses on implementing movement and jumping, which does not feel limp or rigid. He also goes through the implementation explaining how different elements will influence the movement of the character (Pignole, 2014). Mark Venturelli has similarly written a series of tutorials on the topic. In *Game Feel Tips II: Speed, Gravity, Friction* (Venturelli, 2014) he focuses on how the implementation of speed, gravity and friction influence the movement of the character. He expands on this in *Game Feel Tips III: More On Smooth Movement* (Venturelli, 2014), where he also provides playable examples.

Another way to get a head start are the default character controllers often included with game engines. These can easily be adjusted to do basic platformer movement. For example in the game engine *Unity*, developers can drag and drop a character controller into their game and be running and jumping in no time. The Unity asset store also offers a range of 2D character controllers which are either free or relatively cheap.

With all these resources available the technical part of implementing a platformer is well covered. However, in terms of releasing a game the default character controllers or the result from following tutorials are likely not going to be sufficient. Games can be very different and will have different game mechanics, level design, graphics and audio. It therefore stands to reason that movement and jumping will also be designed and adjusted to fit the specific game. This will include adjusting all the parameters of the character. More likely it will include drastically modifying or entirely remaking the implementation as well as continuously adding features specific to the given game. This process is often heavily based on the intuition of the game designer. Knowledge and inspiration might be gathered from existing games by playing them with attention to how they work and feel. However, this will again be based on intuition, and the actual measurements and implementation of the games remain concealed. The aim of this project is to provide information and tools to help the game designer during this process of adjusting movement and examining existing games.

How do we currently get information about a game and its implementation? Developers sometimes share insights about their games in articles or interviews. This can for example be Miyamoto and Tezuka talking about how they designed the first level in the original *Super Mario Bros.*(Nintendo, 1985). They explain how the level was designed to feel enjoyable while gradually teaching the player about the game. They also briefly touch upon how the movement was designed to give Mario a feeling of weight (Eurogamer, 2015). Adam Saltsman (2010) has written an article about how he tuned his game *Canabalt* (Saltsman, 2009). He goes into a lot of details regarding the camera perspective in the game and the run speed of the character.

Similarly David D'Angelo (2015) has written an in-depth article about the movement of the Plague Knight character designed for the expansion to the game *Shovel Knight* (Yacht Club Games, 2014). The article focuses on how the movement for the character was designed to feel wild, maniacal, and explosive while being compatible with the level design of the original game. Another example comes from the online extras to *Indie Game: The Movie* (Pajot & Swirsky 2012). Here the developers of *Super Meat Boy* (Team Meat, 2010) are interviewed and Refenes explains how they noticed players failing to do wall jumps. He dealt with this by adding a delay of about 1/20 of a second when the player navigates away from a wall slide (IndieGame: The Movie, 2012). Developer insights like these examples can provide valuable knowledge for designers. However, they are limited both in amount of details and which areas are covered. As such they might not include a specific detail a designer is looking for. Furthermore, only some developers share insights so there might be no information about a particular game. When available, this kind of information can be great, but a more general approach is needed.

How else might we gather information about movement in a game? One way is to look at the computer memory used by the game while it is running. By inspecting and experimenting with different parts of the memory it is possible to find the exact values used by the game, for example the acceleration of the character when they start to run. This approach is especially ideal for researching older games. They use far less memory than modern games so locating the correct memory values is therefore more approachable. This method has been used by Aldrich (2012) who has gathered the majority of the parameters used in a range of *Mario* games. His research includes *Super Mario Bros.*(Nintendo, 1985), *Super Mario Bros. 3* (Nintendo, 1988), *Super Mario World* (Nintendo, 1990), and *Super Mario Advanced 4* (Nintendo, 2003). A similar approach has been applied to *Sonic the Hedgehog* (Sonic Team, 1991) resulting in a very detailed guide describing the physics of the game (Sonic Retro, 2014).

A similar approach more suitable for modern games is to use a debugger to inspect the game while it is running. Modern games use a lot more memory and the debugger provide search functionality, which makes locating the correct value more approachable. This method has been applied by Michael Olney(2013) who used the OllyDbg debugger as well as an unspecified memory searcher to inspect *Super Meat Boy*. The result is a breakdown of how the basis of the game could be implemented. He shares his implementation, but omits the specific values of the parameters. These methods offer lots of detail and are highly relevant to this project, especially the measurements of *Mario* which will be used throughout this project. However, these methods require a lot of technical knowledge and are extremely time consuming. This type of information is ideal for looking closer at the games. However, a more efficient way to gather information about movement is needed, which does not require technical expertise.

Let us look at a few examples of how this has already been accomplished. Daniels (2013) compared jump duration between 2D and 3D games. His, admittedly not very precise, way to measure this duration was to start a stopwatch at the same time he pressed the jump button, and to stop it when the character's feet touched the ground. He concluded that what he considers "good feeling jumps" have roughly the same duration in 2D and 3D games, lasting somewhere between 0.7 - 0.8 seconds. These kinds of experiments are easy to conduct but imprecise. However, the idea of timing the movement appears promising. A more accurate approach is described by Mick West (2013). His particular method measures the delay between physical input on a controller and visible reaction on the screen. Briefly explained, he uses a video camera to simultaneously record controller being pressed and reaction on a tv screen. The recording can be inspected, and the amount of frames between input and response counted. Knowing that the camera records at 60 frames per second, he can convert frames to seconds which gives him his measurement. This response time is an important factor for how interacting with a game feels. Instead of focusing on this response time, this project will primarily explore what takes place within the game.

If we wish to measure the movement of the game character, a physical video camera is not ideal. Instead, screen recording software can be used to record the screen. This is convenient as the movement of the character can be measured without a physical setup. The previously mentioned study of gravity in *Mario* games used this approach. Again, the frames were used to measure time and the size of Mario was used as a reference point when comparing the gravity from the game with the gravity on earth (Lefky & Gindin, 2007). Recording the screen and measuring the movement seen in the video, is the starting point for the approach developed for this thesis. The next chapter will describe how it was developed and refined.

To summarise, the aim of this thesis is to create a generalised and efficient approach to collecting detailed information about jumping in existing games. Three platform games will be measured using the developed method. A framework of basic movement and jumping in platform games will be defined. This framework will help structure and compare data from different games. It will also establish a terminology for movement which can be used throughout this project and elsewhere. A tool for prototyping jumping will be created which is compatible with the framework and the data measured. This can be used to confirm the measurements by creating simple playable versions of the characters from the games. This tool will also visualise how the characters move, which will be a great help for understanding their movement. The game feel of the three case studies will be discussed base on the measurements taken. With the direction of the project established we can now introduce the concept of *Game Feel* which will be used throughout the project.

Game Feel

This chapter will present the concept of *Game Feel* as described by Steve Swink in the book *Game Feel: A Game Designer's Guide to Virtual Sensation* (Swink, 2008). Game feel can be used to analyse a game or an aspect of a game. Throughout this project game feel will be used to discuss the feel of jumping and moving.

Swink defines game feel as "Real-time control of virtual objects in a simulated space, with interactions emphasized by polish" (Swink, 2008, p. 6). According to Swink a game with "great game feel" conveys a range of experiences to the player. For this project the two most important experiences are "The aesthetic sensation of control" and "Interaction with a unique physical reality within the game". Game feel is a combination of these experiences. They are not mutually exclusive and each experience is always present to some degree. They can change during play with different experiences being in focus at different times. Game feel is analysed by looking at the three components from the definition: *Real-Time Control*, *Spatial Simulation* and *Polish*. These components translate into the experiences and can be used to discuss different aspects of the game. This can be used by a game designer when trying to emphasize a certain feel or locating what is wrong with a certain design. Intentionally avoiding game feel to create the opposite experience of what is described is also a way to use the concept. The three components will now be further described.

Real-Time Control

The focus of *Real-Time Control* is how fast the game responds to input from the player. This is essential for how an interaction feels. An interaction is seen as a cycle. One half of the cycle takes place in the head of the player, as she perceives the current state of the game, interprets it, and reacts by giving input. This takes roughly 240 milliseconds(ms) and covers one reaction from perception starts to input is given. This number is useful as it gives a concrete idea of how fast a player can react. The other half of the cycle takes place in the computer which receives the input, processes it, and produces an output for the player to interpret. For the response to feel instantaneous the computer must respond as fast as the player reacts, which as mentioned takes about 240ms. In fact, for the controls to feel tight, the response time should be 100ms or less. The term responsive will be used to refer to this ideal reaction time. If response time gets as high as 150-240ms it will begin to feel sluggish or floaty and above 240ms the feeling of real-time control is lost. Being responsive establishes the impression of causality. In other words the player connects action with output and feels in control. The numbers presented by Swink (2008) are averages and will vary depending on the psychology of the player and the circumstances under which she plays. With that in mind, the averages can still be used for general comparison.

According to Swink real-time control can be measured as response time. This begins with the reaction time for a single input. The player presses the right arrow key and as a result the game character starts moving right. To have real-time control the game must offer continuous control. This means the game should react to a series of input. For example, if the player wishes to change direction and presses the left key, she will likely expect the character to decelerate and start running left. For both, the single input and the series of input, the time between input and response from the game can be measured. Comparing these measurements with the durations previously mention will determine to what degree the game have real-time control. Which will indicate how responsive the game will feel.

Figure 3: Castlevania (Konami, 1987)

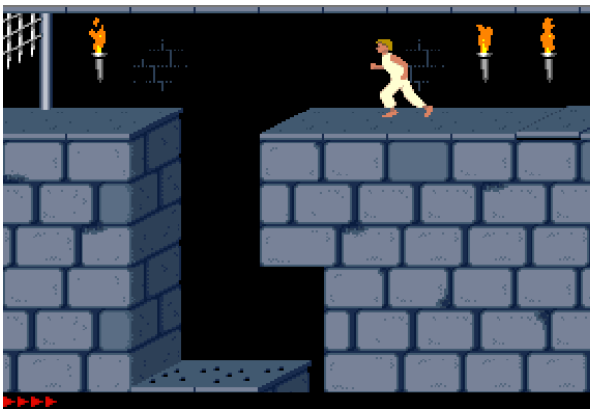


Figure 4: Prince of Persia (Brøderbund, 1989)

Swink provides an example from *Prince of Persia* (Brøderbund, 1989). In this game each animation of the prince moving must be completed before the game reacts to new input. If the player presses right the prince will start running. The response time here is less than 100 ms, fast enough to feel instantaneous. However, the run animation is 900ms long and during this time input is ignored. This breaks the feeling of real-time control and the movement feels sluggish or even unresponsive. Another example measured for this project is movement in *Castlevania* (Konami, 1987). Here, the response time when starting to move, coming to a halt or changing direction is just 33ms. This means the movement has real-time control and will feel responsive.

Simulated Space

This component explores how a game simulates physical interactions within a virtual space and how this is communicated to the player. *Simulated Space* is evaluated by looking at how a game conveys this simulation. The collision between the character and the environment can for example be used to establish the physicality of the game world. When a character is moving gravity, acceleration, and friction can be used to convey the weight of the character. This supports the game as a simulated space. In contrast to this, there are elements that counteract spatial simulation and make the game feel more abstract. For example, representing parts of the game as symbols. Another example is to remove the avatar altogether, as a result interactions with the simulated space will feel more distant. This is, for example, seen in most strategy games. It is worth mentioning, that the realism of the game world is irrelevant in terms of spatial simulation. The game might have no gravity and a variety of fictional creatures, but still communicate and simulate the properties of that world convincingly. It is the amount of abstraction compared to the feeling of presence that is important. The stronger this component is the more convincing the interactions in the game world will feel. This will make the player feel more present and spatially immersed in the game.

To demonstrate the above points, we can compare *Super Mario Bros. to Donkey Kong*. Both games simulate a running and jumping character. In *Donkey Kong* the character moves at a constant rate and every jump is identical. This makes the movement feel stiff and does not convey the feeling of weight or forces applied. However, in *Super Mario Bros.* the character has acceleration, deceleration and variable jump height. The weight of Mario and the impact of the forces when running or jumping are all being communicated to the player. This increases the feel of the game as a simulated space. In the context of this project the concept of simulated space will be applied to different elements of jumping and moving. This will be useful when discussing how each element of a jump influences the feel of moving the character within the game world.

Polish

The third and last component of game feel is *Polish*. This component describes elements which contributes to the feel of the game without being strictly necessary for the game to be played. This could, for example, be run animations, the sound of jumping, or the particle effects spurting behind a sprinting character. It should be possible to remove all polish without altering the functionality of the game. That being said polish can be just as important for game feel as the two other components. The way the game is represented through visuals and audio can be seen as polish, which plays a crucial role in how the game is perceived by the player. To narrow the scope of this project, polish will not be considered. In that respect, the concept of polish represents what is excluded from movement in this project.

Case Studies

Three classic and diverse platform games were chosen as case studies. They will be measured using the approach described in later chapters. The case studies will also be used as examples when describing the movement framework. Lastly, the game feel of the movement in each case study will be analysed and discussed. As mentioned earlier, the games chosen were *Super Mario Bros. 3* (Nintendo, 1988), *Limbo* (Playdead, 2011), and *Super Meat Boy* (Team Meat, 2010). This chapter will introduce the games and explain why they were chosen.

Figure 5: Super Meat Boy (Team Meat, 2010)

Super Meat Boy (Team Meat, 2010) is a modern platform game with many similarities to earlier games in the genre. In the game you play Meat Boy who must jump his way through deadly contraptions. The game is fast paced and the levels relatively short. Often one continuous flow of movement without breaks is required to complete a level. Death is always imminent, and touching any danger whatsoever kills the character on the spot. The level instantly restarts, which establishes a flow between the countless attempts typically required to finish the later levels of the game. The game has been praised for very tight and responsive controls. Apart from being way faster the movement appears similar to games in the *Mario* series.

Figure 6: Limbo (Playdead, 2011)

In *Limbo* (Playdead, 2011) you play as a boy searching for his sister. The world is gloomy, melancholic, and full of hidden dangers waiting to tear the boy apart. To progress, the player must solve a range of puzzles, which often involves navigating elaborate traps with great caution. The game offers dynamic movement which adjusts to the surroundings. The boy can grab ledges while jumping and pull himself up. While on the ground he can step or climb onto small plateaus without needing to jump. The movement in general is slow and does not allow drastic changes. The game was praised for its audio design, style, and atmosphere. The pace of the characters movement supports the mood of the game and does not focus on speed or agility. The movement in *Limbo* is used as a contrasting example compared to the other case studies.

As previously mentioned the *Mario* series established the platform genre and continues to be popular to this day. *Donkey Kong* established jumping in video games, but had a very constrained jump. *Super Mario Bros.* further refined the jump and allowed the player to control the height and distance of the jump. *Super Mario Bros. 3* (Nintendo, 1988) simplified the implementation and removed some of the technical quirks from the previous game and is a common favourite among fans. Unless otherwise noted, *Super Mario Bros. 3* is the implied title when mentioning *Mario* during this thesis.

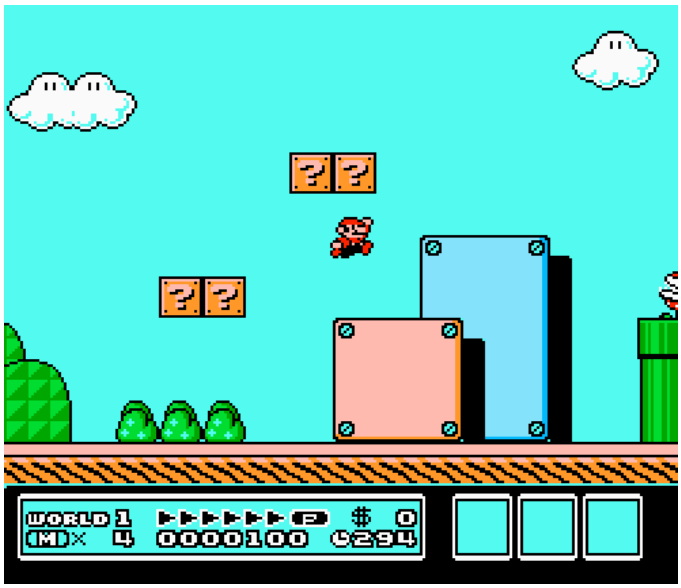


Figure 7: Super Mario Bros. 3 (Nintendo, 1988)

The game was originally released for the Nintendo Entertainment System (NES). This comes with some technical limitations not present for the other case studies. Most importantly, it uses a binary input from the controller and a resolution of just 256 x 240 pixels. This influences the movement of the character as discussed in later chapters. The movement in the game was praised for conveying the weight and speed of character, and the physicality of the game world was unique at the time. As we have seen, *Mario* has already been thoroughly researched, measured, and reverse engineered. By inspecting the memory allocated by the game, Aldrich obtained most of the parameters used for the movement in the game. The guide *Mario Bros. 3 Physics* (Aldrich, 2012) describes the physics and movement of the game. A sample measurement of *Mario* was done confirming the accuracy of the guide. Analysis of *Mario* throughout this project will be based on this guide and the included parameters.

With the case studies in place and the basic concepts introduced we are ready to develop an approach for measuring movement.

Measuring Movement

This chapter will describe an approach for measuring jumping in existing games. The measurements will be the foundation for the analysis, modelling, and reflections done throughout the project. This process will be separated into three sections, looking at simulating input, recording the result, and measuring the movement.

Simulating Input

To accurately control movement in the game, precise control of input is needed. Manually controlling the game is not ideal, since human input is neither precise nor particularly well timed. Using a program to simulate input is a better solution. The program should be capable of simulating a keyboard as well as a controller. It should have an interface for setting up a sequence of timed input and needs to output the current state as well as a timer. It should also be generally applicable, working with as many games as possible without adjusting the code of the program. No existing software was found which was capable of this. Therefore, a new program was developed for this project with the help of a number of existing technologies. For the rest of the project this program will be referred to as the *Input Simulator*. The program is a C# Form Application and was developed using Microsoft Visual Studio, which provides a visual editor for quickly designing a GUI.

The approach developed is based on EasyHook, a framework for injecting DLLs and hooking into API calls. In layman's terms this mean you can write your own custom code and EasyHook can place it within a game already running. Hooking means the code placed in this way can intercept function calls between the game and the operating system. This allows certain parts of the game to be modified or controlled. For this project two functions were intercepted this way. They both originate from the DirectX API.

The first function is from XInput, the API for the Microsoft XBox controller. The specific function we are interested in is XInputGetState which is responsible for getting the state of the controller to the game. The game asks the operating system what the current state of the controller is, our code intercepts this call and instead of the actual controller state we can now send back any controller state we like. This allows us to simulate any controller we wish.

The second function is the update function called every time the game is drawn on the screen. Instead of preventing this call from ever reaching the operating system, we add a few extra elements to be drawn and pass it on like normally. This allows us to draw directly on top of the game, which will be used to write the current state of the controller as well as a timer. This function also works as the main update loop for the Input Simulator.

Injecting a DLL and intercepting these two functions was the main challenge when developing the Input Simulator. Simulating a keyboard turned out to be much easier than simulating a controller. It does not require injection, instead a keyboard event can simply be broadcastet from the Input Simulator and will be picked up by the game automatically. When input simulation and overlay was in place, functionality was added to the program and the GUI expanded.

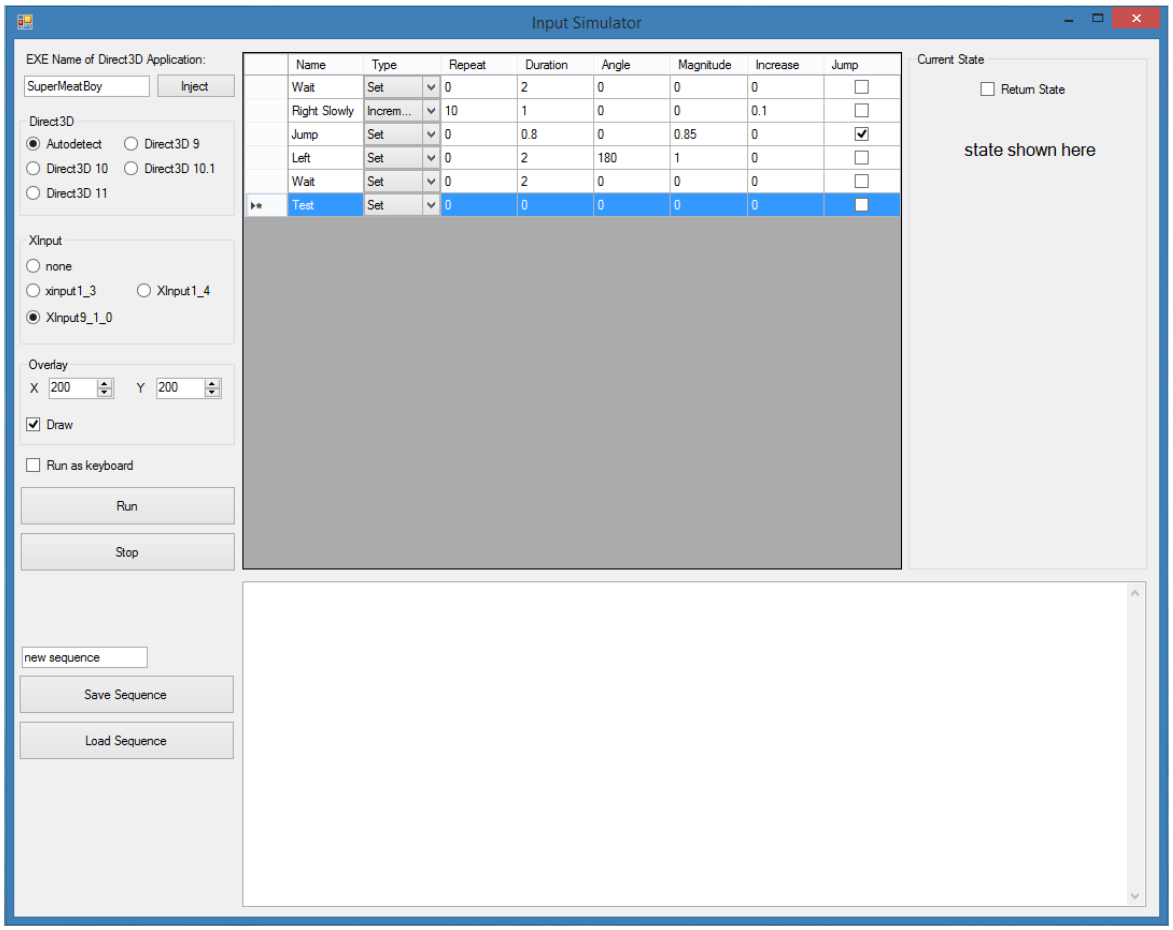


Figure 8: The Input Simulator

Figure 9: While input is being simulated an overlay is drawn on top of the game. This overlay shows the currently simulated state as well as a timer.

Figure 8 shows the Input Simulator. While explaining how to use the software each element of the GUI will be described, starting at the top of the leftmost column. Here the user writes the .exe filename of the game they wish to inject into. If in doubt, the name can be found in windows task manager. The inject button initiates the injection process and will only work if the game is already running. Further down in the column the user can choose the Direct3D version used by the game, this can be autodetected or set explicitly. Similarly, the XInput version of the game can be set. The overlay section controls the orange text drawn on top of the game. X and y control the positioning of the overlay and can be adjusted both before and after injection. The overlay can also be disabled entirely. If "Run as keyboard" is enabled, the program broadcasts the input sequence as keyboard events instead of using injection. The "Run" button starts the simulation of the current input sequence. The user must inject before running the simulation. The "Stop" button stops any input sequence currently being simulated. The buttons labeled "Save Sequence" and "Load Sequence" can be used to save or load a sequence of input. The sequence is saved as a text file in the data folder of the application.

The rightmost column has a section labeled "Current State". Some games do not allow an overlay to be drawn. When this is the case the currently simulated state can be sent back to the Input Simulator and showed here instead. When using keyboard events, the simulated state is also shown here. At the bottom is a white square which is used as a console where the program can write information to the user. If the injection fails, the user is notified here.

	Name	Type	Repeat	Duration	Angle	Magnitude	Increase	Jump
	Wait	Set	0	2	0	0	0	☐
	Right Slowly	Increm...	10	1	0	0	0.1	☐
	Jump	Set	0	0.8	0	0.85	0	☑
	Left	Set	0	2	180	1	0	☐
	Wait	Set	0	2	0	0	0	☐
▶*	Test	Set	0	0	0	0	0	☐

Figure 10: A sequence of input, ready to be simulated by the Input Simulator

In the centre of the application the user can set up a sequence of input to be simulated. Figure 10 shows an example of an input sequence. The rows are always executed sequentially, starting from the top proceeding downwards. Each row represents a configuration of a controller to be simulated. The rows can be moved to change the execution order. If anything is written in the bottom row an additional row is added underneath it, which is how to expand the sequence.

Describing each column will give an overview of how to set up a sequence of inputs. In the *Name* column the user can name each row. The name is displayed in the overlay when the row is being simulated, but does not influence the state of the controller. The name helps the user keep track of how far into the sequence a simulation is. The *Type* column has two settings which influence the entire row. *Set* means the controller state is set as an absolute value. The alternative is *Increment* which cumulatively adds the values of the row to the current controller state. The *Repeat* column is the amount of times the entire row should be repeated. Combined with *Increment* this can be useful for increasing a value slowly over many steps. The *Duration* column controls how long a given row is simulated before the program proceeds to the next row. The *Angle* column is the direction of the controller thumbstick. This can be any angle ranging from 0 to 360. In this case 0 is right, 90 is up, 180 is left, and so forth. If the keyboard mode is active the angle will be converted to the closest arrow key input instead. The *Magnitude* column is how far the thumbstick is being pressed. 0 is a centred stick and 1 is the stick being fully pressed. The *Increase* column is only used when *Type* is set to *Increment* and as such

replaces *Magnitude* when increasing input. The *Jump* column controls if the jump button is pressed while simulating a given row. For the controller the A button is used and for the keyboard the spacebar.

The sequence seen in figure 10 will do the following:

- Wait for 2 seconds without any input.
- Slowly push the stick to the right across 10 seconds. Pushing it 0.1 further every second.
- Hold jump for 0.8 seconds with the the stick pushed 0.85 to the right.
- Release jump and change direction holding left for 2 seconds.
- Conclude by waiting 2 seconds again without any input.

Figure 11 shows a close-up of the overlay drawn by the Input Simulator. The first line shows how many frames per second (fps) the game is running at. It also shows callbacks per seconds (cps), which indicates how many times per second the game is calling the XInputGetState function. In other words, how often the game would normally ask the physical controller for an updated state. Next, the x and y components of the thumbstick are shown, followed by the magnitude of the input. "Jump Released" or "Jump Pressed" indicates the state of the jump input. The line reading "Left 0.566" in the example shows which part of the sequence is currently being simulated. "Left" is the name defined by the user, and 0.566 indicates how many seconds the row has been simulated for. "TotalTime" shows the time in seconds for the entire input sequence. This timer will be essential later when measuring the movement.

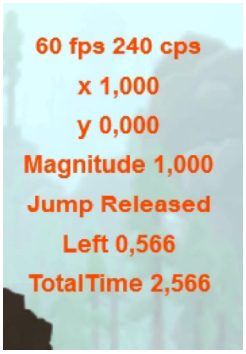


Figure 11: Overlay drawn by the Input Simulator

With this in place the Input Simulator is ready to be used. A variety of resources played a crucial role in the development of the program. Two articles by Justin Stenning were particularly helpful when developing this software. One article introduces the concept of hooking and how to use it to capture screenshots (Stenning, 2010). A second article expands on the first and describes how to use hooking to draw an overlay on a game

(Stenning, 2011). Stenning also shares an implementation example, which injects into games, takes screenshots and draws an overlay (Stenning, 2012). This implementation was used as the foundation for the Input Simulator, which saved a significant amount of time. Mads Johansen Lassen has experience with hooking into XInput and his help during this project and the source code he provided from his implementation was vital for the development of the Input Simulator. Two open-source projects were also particularly useful, SharpDx provided the required XInput DLLs and EasyHook provides fundamental hooking functionality.

The pros and cons of the program will now be discussed. A requirement for the program was to be as generally applicable as possible. The program can indeed be used with different games without requiring any changes in the code. Sequences of input can be saved, loaded and executed for different games making the sequences reuseable. That being said the tool does have certain limitations. The program only runs on Windows and only injects into games also running on Windows. Porting the program to other platforms would likely require significant work and potentially a whole new approach. The Input Simulator only works with games using DirectX 9, but extending the software to newer versions of DirectX should be a straightforward process. The tool currently supports three versions of XInput. This can easily be extended and only requires a few lines of code and including the given XInput DLL in the project. Using XInput means the software always pretends to be an Microsoft XBox controller. It could potentially be extended to support other controllers as well as the DirectInput API used by older games. Simulating keyboard events works as a good fallback if the controller should fail. However, not being able to simulate analogue input is not ideal.

The Input Simulator offers very precise control. The position of the stick can be defined with at least three decimals of precision, and the state of the controller can be changed reliably at least 60 times each seconds. This is both more precise and faster than a human player is expected to play. This precision is ideal for making measurement, but in terms of realistically imitating player input the program does a poor job. Human players

would, for example, rarely give perfectly aligned input, which will be used repeatedly throughout this project. Changing stick direction happens instantly with the Input Simulator. However, for a human moving the stick as fast as possible will still take enough time for the game to receive input along the way. Defining a sequence of inputs to imitate the imperfections of human players would not be possible with this program. A different approach would likely be required to simulate this kind of input. Lastly, any delay caused by the communication between the operating system and the physical controller is not included in the simulation. Being virtual and inside the game, the simulator will reply as fast as the software allows. This again results in a faster response compared to using a physical controller.

Recording

With the Input Simulator up and running the next step is to record the resulting movement as well as the overlay on the game. This was done with screen recording software. The higher the framerate and resolution of the recording, the more accurate the resulting measurements will be. Recording was done with different software in search of a suitable setup. Initially, Fraps and QuickTime were used, however both had significant impact on the performance of the game and the resulting recordings had irregular or low frame rates, often rattling around below 30. To get better recordings, a computer with a Nvidia graphics card was used. This card supports GeForce ShadowPlay, a screen recording software with minimum impact on performance. This is accomplished by using a GPU accelerated video encoder. This means most of the recording process is handled by the graphics card, allowing the CPU to run unhindered. Using this software resulted in recordings with a resolution of 1920x1080 and a steady 60 frames per second. This was the best solution found and was used for all measurements recorded.

Measurement Tool

With the movement recorded measuring can begin. To analyse the movement in the video we need to measure the position of the character over time. If the video recording was flawless we would not need to worry about time. The interval between two frames could be assumed to always be one frame, equivalent to 1/60 of a second. However, even with the optimised video recording there are minor fluctuations. To compensate for this we need to note down the time for every frame that we measure. The timer in the overlay provides us with this information as it shows the time in every frame of the recording.

The first approach was to load the video into Adobe Photoshop. From here the position of the character was measured at each frame using the ruler tool. Each position and the corresponding time was manually written into a Microsoft Excel spreadsheet. This was slow and labour-intensive. Going back and forth between the two programs was not optimal and manually writing positions and times was ineffective.

A program was created to optimise this specific task. OpenFrameworks is an open source C++ toolkit, it has great support for handling video recordings as well as rendering GUI, which is just what we need. The *Measurement Tool* was created in Xcode using openFrameworks and C++.

Figure 12: The measurement tool

Figure 13: Interface for the measurement tool

Figure 14: Overlay drawn by the Input Simulator

Figure 15: Cross hairs used to mark the position of the character and the reference point

Figure 13 shows the interface of the program. In the top field the user types the name of the video recording they wish to measure. Pressing enter loads the video. The video must be placed in the data folder of the application. After the file has been loaded the user can navigate the video with the keyboard. The D key is used to move one frame forwards, and the A key to move one frame backwards. Similarly the E key moves the video one second forwards and the Q key one second backwards. First the user must pick a start time. This is done by navigating to the desired place in the video and inputting a time underneath the filename. The start time should be the value seen in the overlay. The start time allows the user to begin measuring when it is relevant and any movement before the start time is ignored.

Pressing enter confirms the start time and the three bottom rows in figure 13 appear. They show the time and measurements of the current frame. A green crosshair can now be placed by clicking with the mouse. The crosshair should be positioned on the character as seen in figure 15. The user can step through the video one frame at a time continuously placing the crosshair on the character. If needed, the shift key can be used to zoom. The program automatically increases the timer by 1/60 of a second every time the user moves to the next frame. If this does not match the timer in the overlay the user can correct it by using R, T, Y to increase the first, second or third decimal of the timer, or similarly F, G, H to decrease the timer.

If the camera in the game is not static, just measuring the position of the character is not sufficient. If the camera, for example, pans to follow the character, he will remain in the same position relative to the screen. In our measurements this will look as if he is not moving at all. To compensate for any camera movement a reference point is needed. This is visualised as a white crosshair which must also be placed by the user. This is done by pointing with the mouse and clicking the 3 key. The reference point should be placed consistently on the same part of the environment. In figure 15 the reference point is placed on the wall just where the blood ends. When position, reference point, and timer are in order for all frames the user can click export. The program now considers the two

crosshairs as well as the timer. The position of the character in relation to the reference point is calculated. The measurements are exported as a text file and organised to be easily imported into Microsoft Excel.

This tool made the measuring process much faster. Swapping between programs was no longer required. A mouse click to place the crosshair replaced the tedious process of dragging the ruler tool, reading the value, and manually typing it in. Automatically setting the timer and only adjusting it when needed was also much faster than typing it for every frame. Exporting the measurements as a text file also proved surprisingly helpful as demonstrated in the next chapter.

Analysing Movement

Organising Data

Measurements so far have been in pixels. Pixels are easy to work with, however, at this point in the process they are no longer ideal. The pixel measurements are influenced both by the resolution of the game as well as the resolution of the recording. Recordings from different locations in the same game could have different camera perspectives. One recording might be zoomed in, while another shows an overview, this will impact our measurements. These variations must be handled in order to get consistent measurements which can be compared and analysed together.

The chosen solution was to normalize the measurements using the height of the character. For every recording the height of the character when standing still is measured in pixels. By dividing a pixel measurement with this pixel height of the character, the measurement can be represented as a number of character heights. For the rest of the project this length will be referred to as units. 1 unit is equivalent to the height of the character and a length of 3.5 units would correspond to three and a half character heights. The unit, or u for short, will also be used for speed and acceleration. A speed of 8 u/s means the character is moving five character heights per second, while an acceleration of 5 u/s^2 means the speed of the character is increasing by 5 u/s per second. Looking at the measurements in relation to character height avoids the problems mentioned regarding the pixel measurements. With the measurements converted to units we can combine any recordings of the game, without the influence of camera zoom or the resolution of the recording.

Figure 16: Super Meat Boy and Limbo

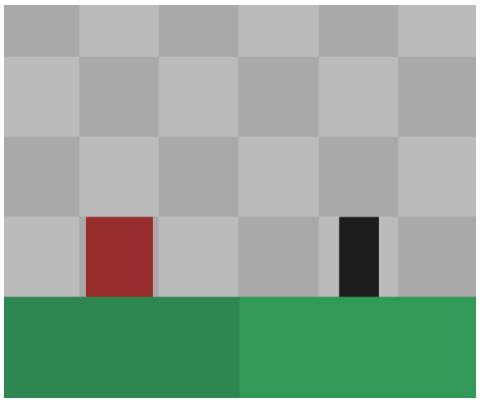


Figure 17: Super Meat Boy and Limbo abstracted

Describing movement using units is also useful when comparing different games. Since the measurements are normalized as character height we can directly compare the movement of two games. However, this comparison assumes that the characters have the same height. Looking at figure 16 this assumption appears to shrink the boy from *Limbo* or alternatively enlarge Meat Boy. What size we perceive the character to be is conveyed by the simulated space of the game and could be a topic for further research. This perception of scale is relevant and should be taken into consideration when comparing games. However, for this project the characters will mainly be seen as abstracted cubes. This will conceal their actual scale as seen in figure 17.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Time	Time Zeroed	Distance Px	Distance Unit	Height Px	Height Unit	Speed Horizontal	Speed Horizontal Fixed	Speed Vertical	Speed Vertical Fixed		Fps
2	0.996	0.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		60
3	1.013	0.017	0.00	0.00	0.00	0.00	0.00	0.34	0.00	0.68		
4	1.030	0.034	0.00	0.00	4.00	0.11	0.00	0.68	6.72	1.36		Resolution
5	1.046	0.050	0.00	0.00	15.00	0.43	0.00	1.00	19.64	2.00		1920 x 1080
6	1.063	0.067	0.00	0.00	27.00	0.77	0.00	1.34	20.17	2.68		
7	1.080	0.084	2.00	0.06	39.00	1.11	3.36	1.68	20.17	3.36		Unit
8	1.097	0.101	4.00	0.11	49.00	1.40	3.36	2.02	16.81	4.04		35
9	1.113	0.117	6.00	0.17	59.00	1.69	3.57	2.34	17.86	4.68		
10	1.130	0.134	8.00	0.23	69.00	1.97	3.36	2.68	16.81	5.36		Acceleration H
11	1.146	0.150	12.00	0.34	79.00	2.26	7.14	3.00	17.86	6.00		20
12	1.163	0.167	14.00	0.40	88.00	2.51	3.36	3.34	15.13	6.68		
13	1.180	0.184	18.00	0.51	97.00	2.77	6.72	3.68	15.13	7.36		Acceleration V
14	1.197	0.201	22.00	0.63	106.00	3.03	6.72	4.02	15.13	8.04		40
15	1.213	0.217	26.00	0.74	114.00	3.26	7.14	4.34	14.29	8.68		
16	1.229	0.233	30.00	0.86	121.00	3.46	7.14	4.66	12.50	9.32		
17	1.246	0.250	35.00	1.00	129.00	3.69	8.40	5.00	13.45	10.00		
18	1.263	0.267	40.00	1.14	135.00	3.86	8.40	5.34	10.08	10.68		
19	1.280	0.284	46.00	1.31	143.00	4.09	10.08	5.68	13.45	11.36		
20	1.296	0.300	52.00	1.49	149.00	4.26	10.71	6.00	10.71	12.00		
21	1.313	0.317	58.00	1.66	155.00	4.43	10.08	6.34	10.08	12.68		
22	1.329	0.333	64.00	1.83	159.00	4.54	10.71	6.66	7.14	13.32		
23	1.346	0.350	72.00	2.06	165.00	4.71	13.45	7.00	10.08	14.00		
24	1.363	0.367	79.00	2.26	169.00	4.83	11.76	7.34	6.72	14.68		
25	1.380	0.384	87.00	2.49	172.00	4.91	13.45	7.68	5.04	15.36		
26	1.396	0.400	95.00	2.71	176.00	5.03	14.29	8.00	7.14	16.00		
27	1.413	0.417	103.00	2.94	180.00	5.14	13.45	8.34	6.72	16.68		

Figure 18: Data imported into Microsoft Excel

Figure 18 shows the data after being imported into Microsoft Excel. The Measurement Tool has organised and extrapolated the measurements. This is accomplished by using the formulas supported by Excel. A formula calculates a result based on values from other cells in the spreadsheet. Each row in the spreadsheet contains information of one frame of the video.

The column labelled *Time* directly corresponds to the timer adjusted during the measurements. Depending on how the start time was set and how the input sequence is set, this timer will start a few seconds into the simulation. To make things easier, the column *Time Zeroed* was added. It has the same intervals as *Time* but begins at zero. The column labelled *Distance Px* is the horizontal distance in pixels between the character and the reference point. *Distance Unit* is the same distance, but converted from pixels to units. Similarly, *Height Px* contains the vertical distance in pixels between the character and the reference point. *Height Unit* is the same measurement but again converted to units. The amount of pixels that goes into one unit is specified to the left in the spreadsheet under the text *Unit.* This value is adjusted to fit the height of the character in the specific recording.

The measurements represent position over time so calculating the the speed of the character is straightforward. The speed of the character is calculated in the columns *Speed Horizontal* and *Speed Vertical*. The speed values are calculated as units per seconds. The last two columns *Speed Horizontal Fixed* and *Speed Vertical Fixed* will be used later when working on the model of the movement. They currently show the speed of the character given a constant horizontal and vertical acceleration as defined by *Acceleration H* and *Acceleration V* to the left in the spreadsheet.

Considerations

The accuracy of the measurements depends on the synchronization between Input Simulator, game and recording. Ideally they should all run steadily at 60fps and in consistent order. In practice this worked well, but was not perfect. Sometimes the character moved visually while the timer in the Input Simulator remained the same. Alternatively, the timer in the Input Simulator sometimes changed without any movement. This kind of behaviour only lasted one frame and in general the synchronization was good. A common recording would have between 0-5 desynchronized frames. This was taken into account when analysing the data.

A limitation of the Measurement Tool is having to visually evaluate the position of the character. The crosshair must be placed on the same part of the character for every frame. For Meat Boy a corner of his body was used and for the boy in *Limbo* the left eye. What is measured is therefore not strictly the movement of the character, but rather the visualisation of movement.

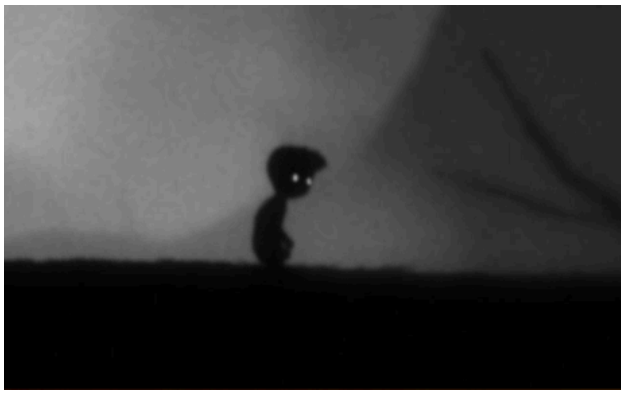


Figure 19: Landing animation in Limbo

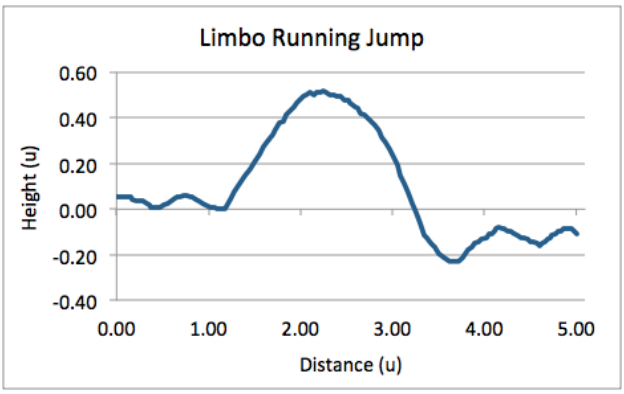


Figure 20: Position when landing in Limbo

Figure 19 shows the landing animation in *Limbo*. During the landing animation the head is lower than when the boy is running normally. Since we measure the position of his eye this will look as if the boy goes through the ground as seen in figure 20. Ideally the character's movement without animations would be measured.

However animation will always, to some degree, influence our measurements. Therefore the animations should be carefully considered when analysing the movement.

Figure 21: Meat Boy changing sprite, the vertical blue line is added as a reference point.

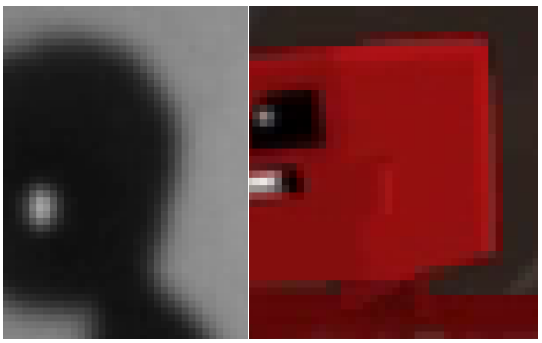


Figure 22: Dithering in Limbo and Super Meat Boy

Another possible source of imprecision is characters changing sprites. Sprites are small, two dimensional images used to draw the character to the screen. When changing direction Meat Boy swaps sprites as seen in figure 21. When measured this will look as rapid movement and for one frame the horizontal speed will spike. Since this only last a single frame it can normally just be ignored when looking at the data.

Another complication is dithering of the character as seen in figure 22. Dithering occurs when the computer approximates from one colour to another. The dithering between the character and the background changes from frame to frame. Dithering makes it difficult to precisely establish the position of the character. How the crosshair is placed can therefore vary with one or two pixels. This also applies to the reference point and measurements where both the character and the camera move simultaneously will be slightly less precise.

The size of one pixel is also a limitation. Using a high resolution recording helps, but the minimal unit measurable is still one pixel. Zooming in on the recording will just cause dithering and will not give more accurate measurements. Games have a maximum resolution. If this is lower than the resolution of the recording it will limit the precision of the measurements. *Mario*, for example, has a maximum resolution of just 256 x 240 pixels. This means that the smallest horizontal distance we can measure is 1/256 of the width of the game. This drastically limits precision. Similarly, one frame of the recording is the minimum time interval. With 60 frames per second this is not a major problem, but will still be responsible for some noise in the data. This imprecision will be larger the faster the character is moving, since we cannot measure what goes on between the frames.

The imprecisions mentioned here should all be taken into considerations when analysing the data. Often one imprecise frame can just be ignored or removed from the data without affecting the overall result.

Analysis

This chapter will look at how the measurements were analysed. The goal is to make a model of how the game might have been implemented. Whether this model matches how the game is actually implemented is not a concern. What is important is that they are functionally equivalent. The better the model, the more accurately we can replicate the movement of the character, which will provide valuable information for analysing the game further.

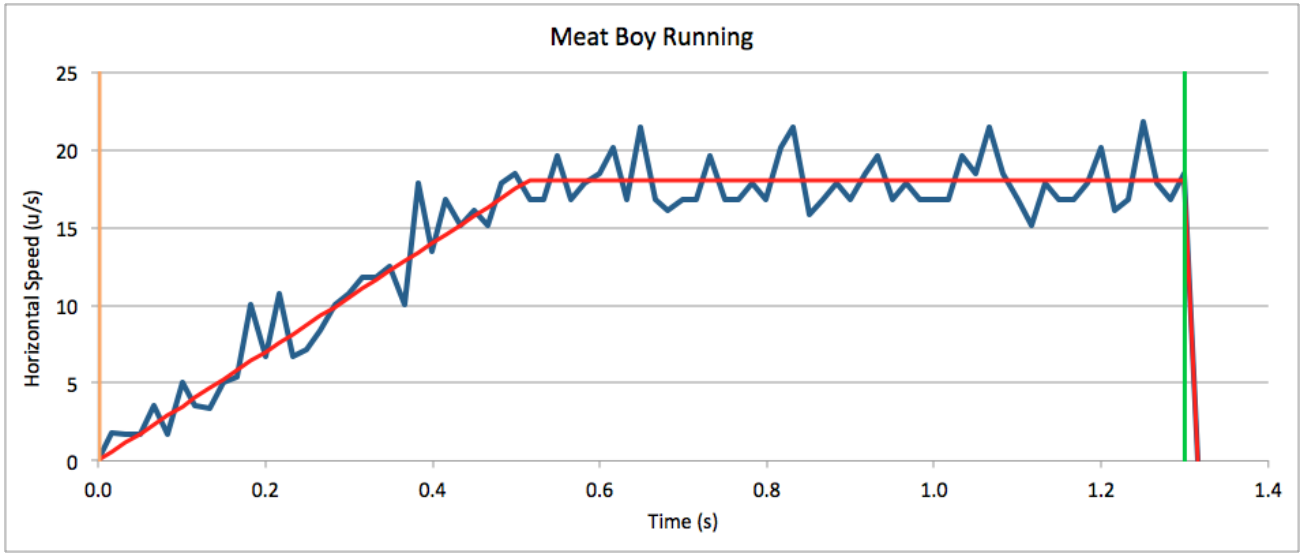


Figure 23: The horizontal speed of Meat Boy running right

Figure 23 shows the movement of Meat Boy visualised as curves. More specifically it shows his horizontal speed over time. The blue curve shows the measured movement, with each point on the curve representing one frame of data. The vertical orange lines shows when input started and the green line when it ended. Looking at how horizontal speed changes over time says a lot about the ground movement of the character. The initial slope is the character accelerating as right input is given. The flat centre part is constant speed, the maximum ground speed of the character. The sudden drop at the end is the character decelerating after input is released.

A model of the character's speed is created based on these assumptions. The red curve shows the resulting model. This model assumes a constant acceleration when input is given and assumes speed is limited by a maximum value. When input is released an instant deceleration is assumed. The model contains specific values representing the movement of the character. The character accelerates at a rate of 35 u/s^2 has a maximum speed of 18 u/s and decelerates instantly, from one frame to the next.

The Input Simulator can mimic analogue thumbstick input. Increasing this incrementally and measuring when the character reacts can be used to measure the minimum input required to move the character. Input smaller than this is ignored by the game. With this information we can now compare horizontal movement with minimum and maximum input. Quite surprisingly, the amount of input had no influence on Meat Boy's run speed. However, in *Limbo* the input amount does influence the movement. In *Limbo* this turned out be a nonlinear relationship and a range of different inputs were measured to establish the relationship.

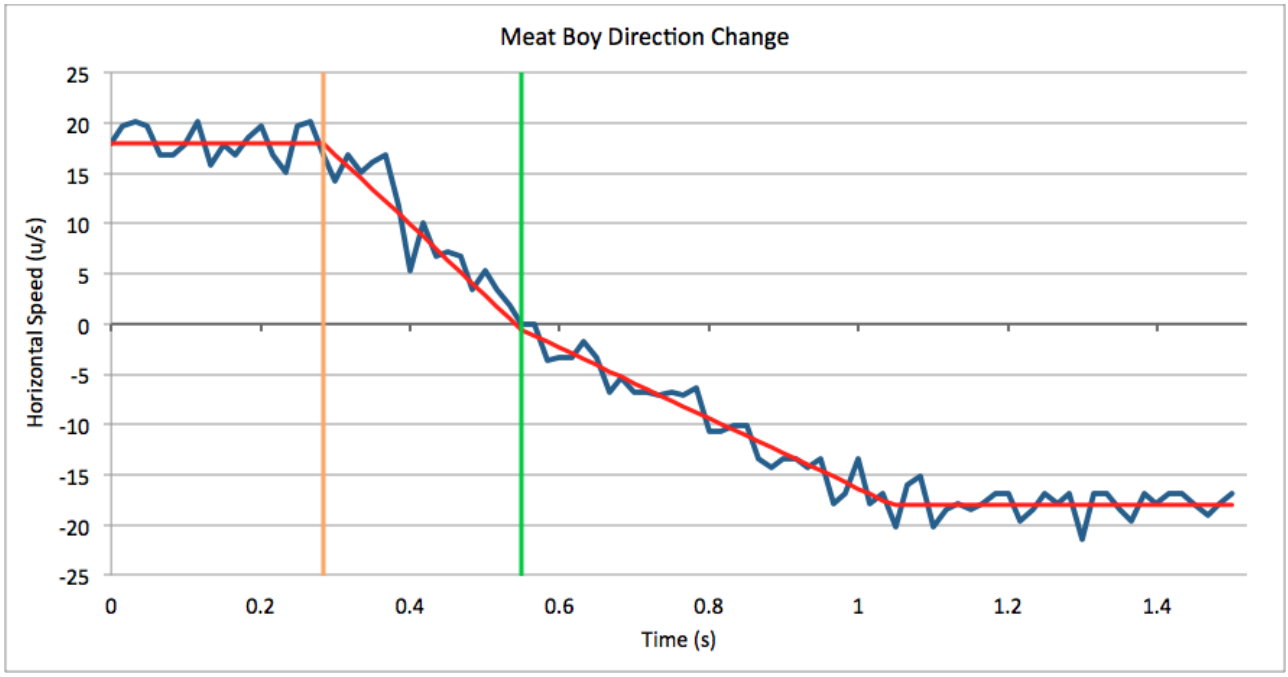


Figure 24: Meat Boy changing direction

Figure 24 shows another example of measurements and resulting model, this time Meat Boy changing direction. The orange line indicates when input was changed from right to left. Notice how acceleration changes during the turn. The resulting model assumes that Meat Boy has a different acceleration of 70 u/s^2 when input is opposite to his current speed. This is the case in the between the orange and the green line. After the green line the regular acceleration is used.

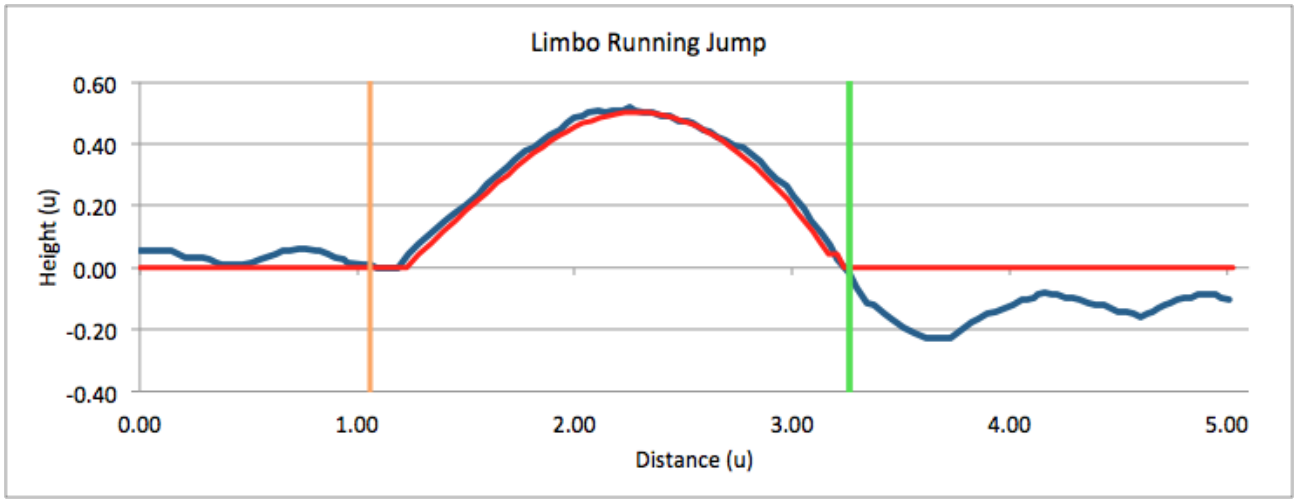


Figure 25: Position over time during a running jump in Limbo

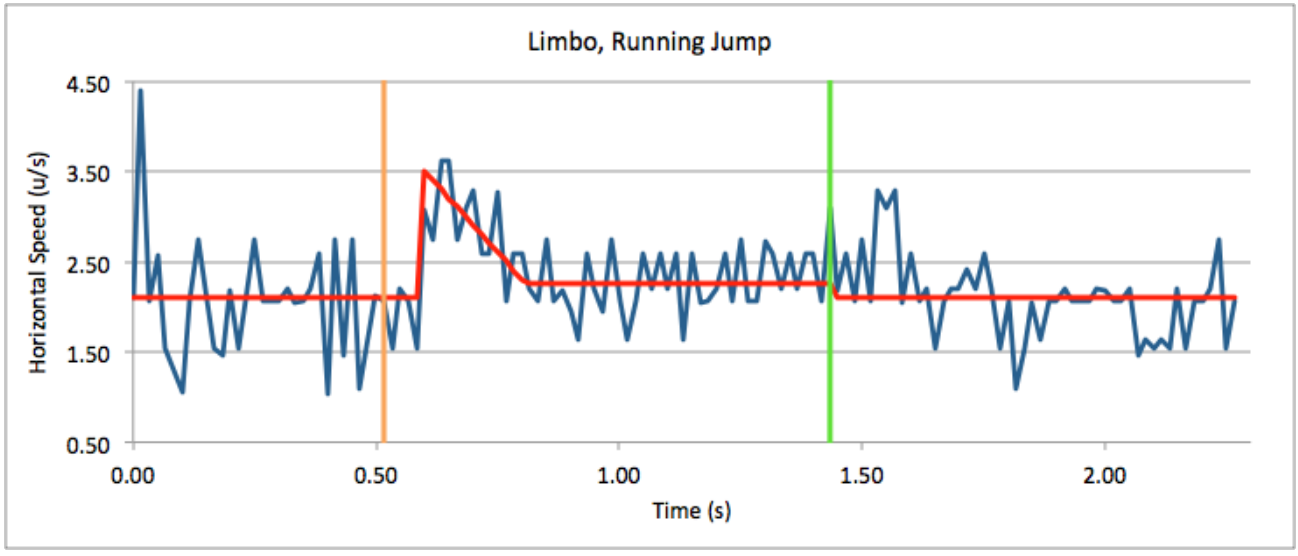


Figure 26: Horizontal speed over time during a running jump in Limbo

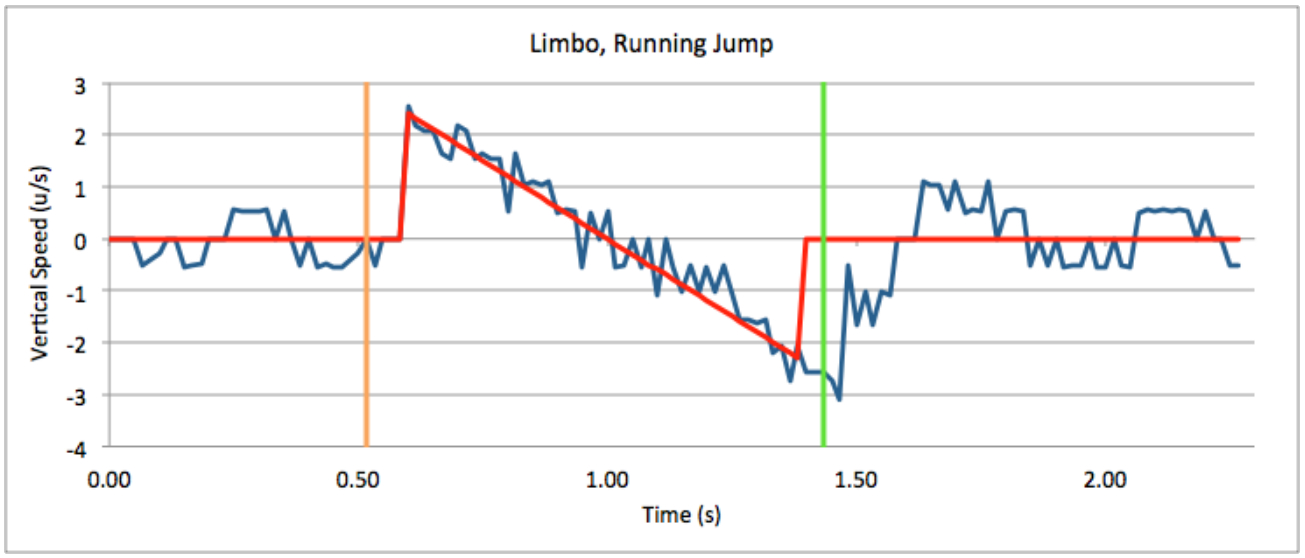


Figure 27: Vertical speed over time during a running jump in Limbo

During analysis it is often helpful to look both at the position and the speed of the character. The model is still based on the speed of the character, but the positions resulting from the model can be compared to the measured positions. As different aspects of the character's movement are analysed, the models for the character can be adjusted or extended to create the best overall fit. Often new measurements would contradict previously made assumptions, which in turn could be adjusted. Figure 25, 26, and 27 show the jump in *Limbo* which turned out to be complex and required several different measurements. The orange vertical line indicates jump input being pressed and the green line indicates when the character hits the ground. The first thing to notice is a four frame delay between input and take-off. This delay was seen consistently in *Limbo* and should be included in the model. The position appears to go underground, this is caused by the landing animation of the character. This is the case because we are using the characters eye to measure position and during the land animation the eye will be further down than when the character is running. The result of the landing animation can also be seen for horizontal speed which has a spike right after landing. The landing needs to be taken into consideration and should not be included in the model. The resulting model presents a range of parameters. A) takeoff velocity with separate horizontal and vertical components. B) A higher maximum horizontal speed when going

through the air than on the ground. C) Horizontal deceleration and minimum horizontal speed while in the air. The red lines in figure 25 and 26 show the speed of the model. The red line in figure 27 shows the position resulting from that model.

Two methods will now be applied to evaluate the precision of the models. The measured positions and the positions predicted by the model will be compared. Position is ideal for comparison as it derives from horizontal and vertical speed. Using position also avoids the fluctuating speed commonly seen in the measurements.

The first method of comparison is Root-mean-square deviation (RMSD). This method is frequently used to compare prediction models with observed values. RMSD is a formula which calculates the magnitude of deviation between two data sets and summarises that deviation into a single value. This method is good for accuracy and will be applied to horizontal and vertical positions separately. The second method is looking at the Fréchet distance between the two curves created by the positions. This method is often explained as a man walking his dog. The man walks along one curve and the dog walks along the other. They walk continuously from the start of the curve to the end of the curve and are connected by a leash. Both can change speed and even stop, but can never backtrack. The Fréchet distance between the curves is the length of the shortest leash sufficient to make such a walk. This method takes the ordering and location of the points into account. The leash explanation makes the resulting value more tangible compared to RMSD.

During this project both methods were implemented from scratch as modules in Excel. They were programmed using the built-in editor and the visual basic programming language. These modules can be exported as .bas files and imported into other spreadsheets. After importing the modules RMSD and Fréchet distance can easily be calculated by selecting the required data. Late in the project an error was found in the implementation of the Fréchet distance, which allowed the man and the dog to walk backwards, which is not allowed in the canonical definition of the Fréchet distance.

Since the curves in this project are relatively simple, this is expected to only cause minor errors. Both methods are scale dependent, which in this context means faster movement is expected to create larger deviations. Therefore different games must be analysed separately.

The methods were applied to measurements and models made for *Super Meat Boy* and *Limbo.* Two examples were chosen to get a feeling of the general accuracy. First the character moving forward and second a forward jump.

	RMSD Distance	**RMSD Height**	**Fréchet distance**
***Super Meat Boy*, Jump**	0.109	0.199	0.239
***Super Meat Boy*, Running**	0.095	-	0.160

Figure 28: RMSD and Fréchet distance for two examples from Super Meat Boy

The results from *Super Meat Boy* can be seen in Figure 28. Jumping is less accurate than running. This is expected as jumping is similar to running, but with added complexity. The largest deviation is the Fréchet distance of the jump curve with a value of 0.239 units. This is equivalent to 23.9% of the character's height or 8.365 pixels, the distance is visualised as a black line in figure 29. With the high speed of Meat Boy taken into consideration, this indicates a highly precise model.

Figure 29: Fréchet distance between model and measurement

	RMSD Distance	**RMSD Height**	**Fréchet distance**
***Limbo*, Jump**	0.025	0.034	0.053
***Limbo,* Running**	0.039	-	0.018

Figure 30: RMSD and Fréchet distance for two examples from Limbo

Figure 30 shows the results from *Limbo*. When visually inspecting the curves of the models *Limbo* appear less precise than *Super Meat Boy*. However, since the movement is much slower the deviations will likewise be smaller. This highlights why the deviations for two different games should not be compared. The largest deviation is again the Fréchet distance of the jump curve, this time with a value of 0.053 units. This is equivalent to 5.3% of the character's height or 3.42 pixels, visualised again as a black line in figure 31. Taking into consideration that the movement in *Limbo* is much slower, this still confirms the accuracy of the model.

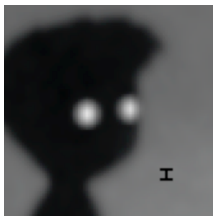


Figure 31: Fréchet distance between model and measurement

From now on illustrations will show the model of the movement and omit the actual measurements. The original data will be attached to the project and include the measurements exported from the Measurement Tool as well as spreadsheets presenting data as well as the models created.

Prototyping Tool

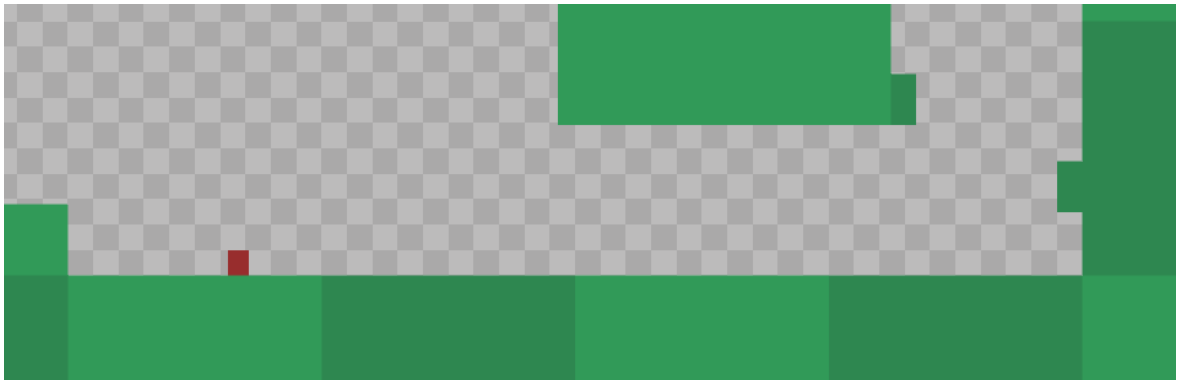


Figure 32: Prototyping tool

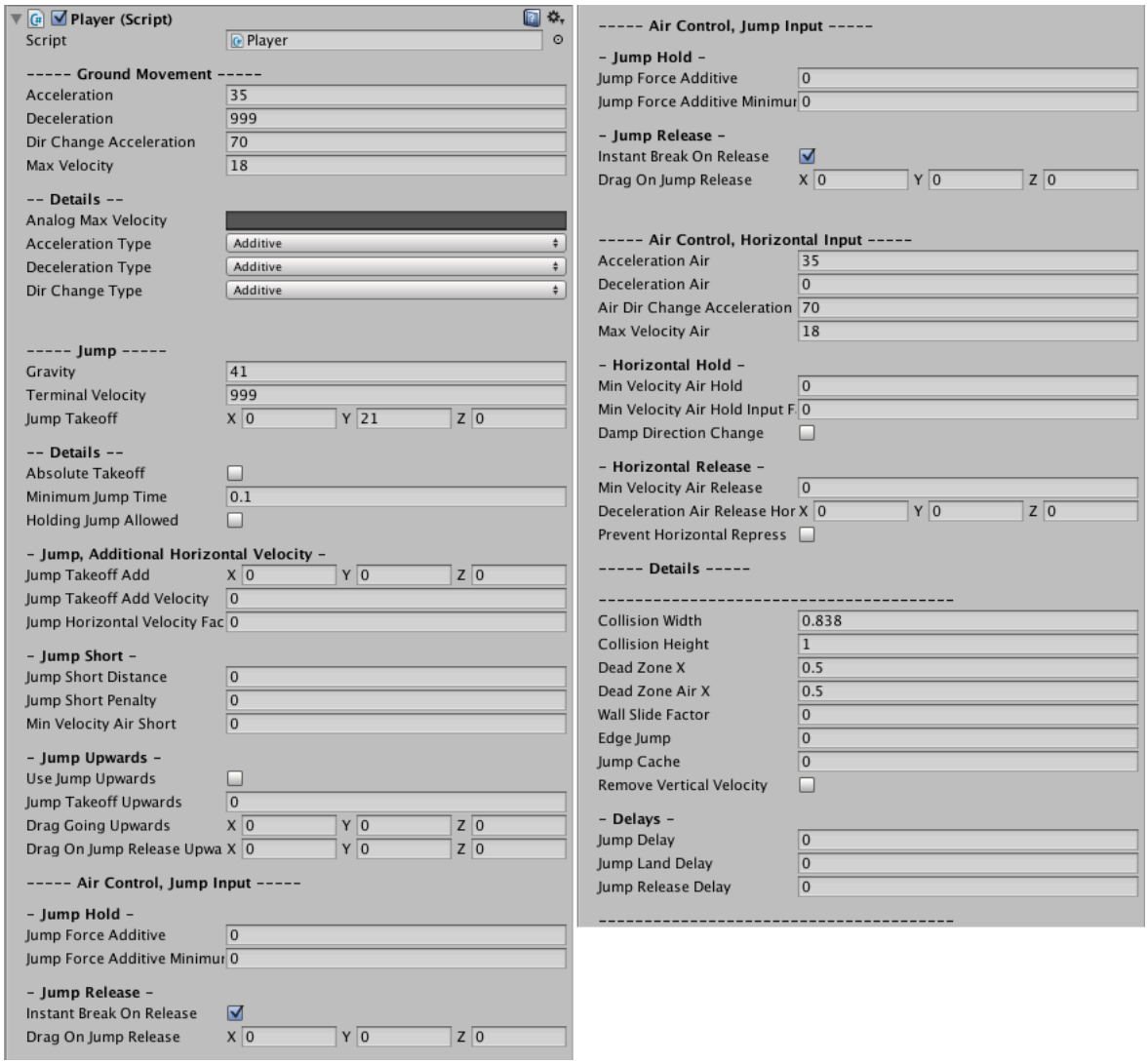


Figure 33: Character parameters in the prototyping tool

A prototyping tool was created while analysing the movement from the case studies. The tool is a simple implementation of a 2D platform game created in the Unity game engine. It includes all the parameters and details explored during this project. As such the movement of both *Mario*, *Super Meat Boy* and *Limbo* can be replicated within the tool. Having a playable version of the models was very helpful when analysing the data and modelling the movement.

Figure 34 shows an example of *Mario* within the prototyping tool. A small part of the first level from *Mario* was also replicated. By playing the prototype version of *Mario,* the movement modelled can be compared to the that of the original game. In *Mario* a jump while standing still can only just make it onto the question blocks, here seen as the first orange platform. The precision of the model can now be checked by performing a similar movement and comparing the result to the game. The jump can also be adjusted within the prototype tool and the model adjusted accordingly. In this way the prototype and the model work together.

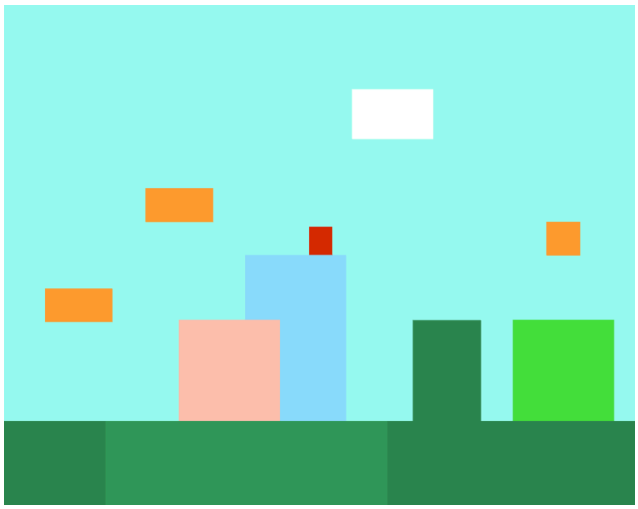


Figure 34: Mario level within the prototype tool

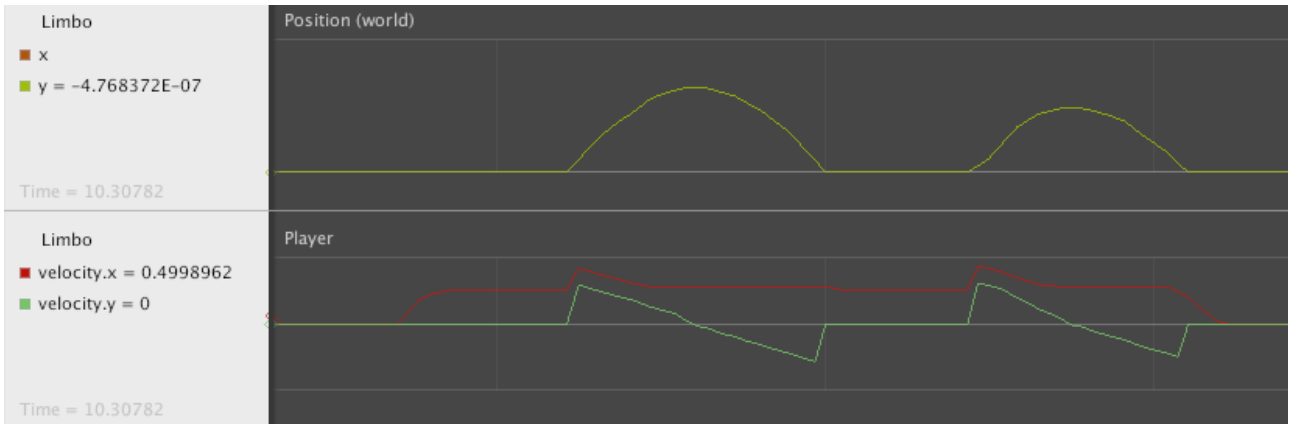


Figure 35: Monitor component

Visualising movement also became an essential feature of the prototyping tool. Looking at how the position of a character changes over time shows a lot about their jump. The movement before, during, and after a jump can be visualised as a curve. From now on such a curve will be referred to as a jump curve. Both size, shape and symmetry of the jump curve turned out to be relevant. Before creating the prototyping tool jump curves where drawn inside Excel. However this was time consuming and not very flexible. Inside the prototype tool jump curves were initially visualised using the Monitor component developed by Peter Bruun (2014) as seen in figure 35. This required almost no work to setup and both position and velocity could be visualised in this way. A weakness of this method was that it only worked while playing and not while editing.

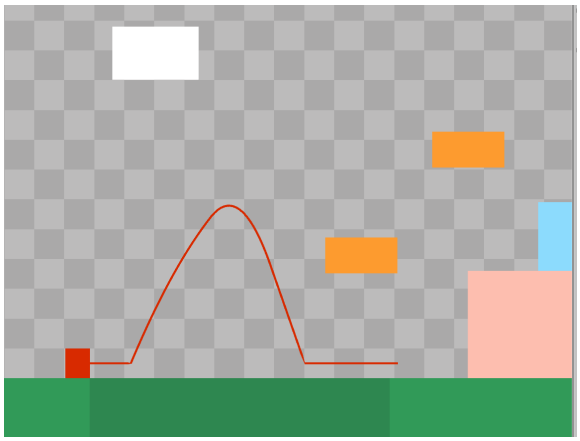


Figure 36: Predicted jump curve

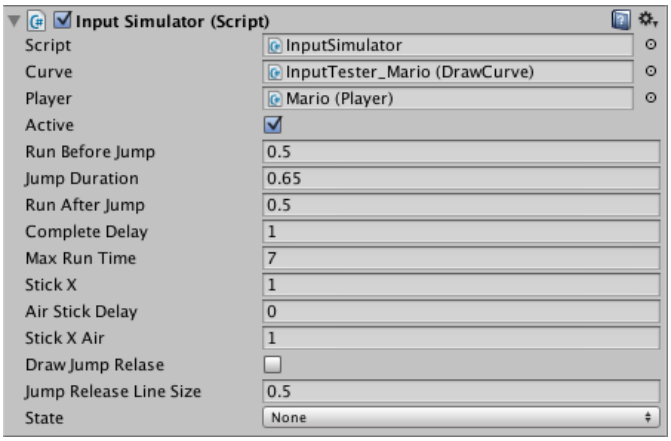


Figure 37: Parameters of predicted jump curve

Further into the project an improved method was developed. With this method, the jump curve could be drawn within the game world as seen in figure 36. The red square represents the character and the red curve shows the jump curve. Each tile in the background is 1x1 unit in size and helps illustrate how large the movement is. This way of drawing the jump curve works while editing and simulates what will happen when the character jumps. In fact, the same code used when playing the prototype, is used when drawing this curve. This secures a very accurate simulation. Any functionality added to the movement will also automatically work both when playing and visualising. The simulation performs a series of input and allows the user to adjust various parameters as seen in figure 37.

This includes how long the character should run before jumping, how long the jump key should be pressed, and how much input to give at various times. Adjusting when jump is released allows the user to simulate smaller jumps which are typically the result of releasing the jump before reaching the peak of the jump. A predefined jump like this is ideal when adjusting parameters and the environment.

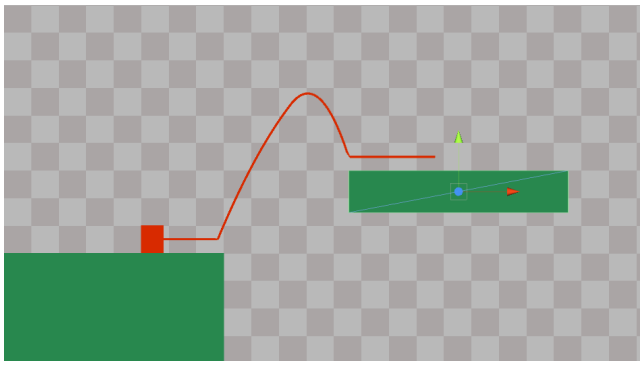


Figure 38: Jump curve adapting to the environment while a platform is being placed.

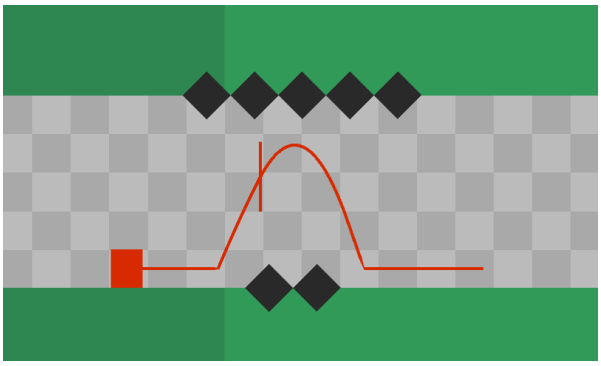


Figure 39: Adjusting gravity to avoid spikes

The curve drawn by the simulation updates continuously and reacts in real-time to all parameters of the character as well as any adjustments to the environment. Providing the user with instant feedback. This method is very adaptable and the character can even be copied to show different scenarios at the same time. Figure 38 shows an example of a platform being placed just within reach of the characters longest possible jump. Figure 39 shows gravity and other parameters being adjusted to prevent the character from hitting the spikes. The red vertical lines shows the jump input being released. This will likely be a challenging jump and to make it through the player must run towards the spikes and, with the right timing, briefly press the jump input.

The prototyping tool can be used by game designers to experiment with various jumps before implementing their own. Various parameters can be adjusted and details added to the jump. A designer can see how a certain adjustment will make the jump feel. The visualised jump curve, provides instant feedback which supports fast and iterative development. The prototyping tool will also be used repeatedly throughout this project to visualise various jumps.

An approach for measuring and analysing movement has been defined and the required tools have been developed. The movement of *Super Meat Boy* and *Limbo* were measured, analysed and modelled using this approach. With this knowledge, creating a framework describing the basic jump is next on the list.

Movement Framework

This chapter will define a framework which can be used to describe jumping and basic movement. The framework is based on the analysis of movement in *Super Meat Boy*, *Limbo* and *Mario*. These three games will be the primary examples when presenting the framework. When needed, properties from other games were measured and included as examples to contrast the three case studies.

The framework divides movement into areas, such as *Ground Movement* and *Air Control*. For each area, a number of contrasting examples will be presented and the area further divided into properties. Each section will conclude with a chart showing the parameters from the games presented. The lack of a given property can often be described in few words, but might have a large impact on the movement in the game, for example the lack of *Air Friction* in *Super Meat Boy* and *Mario*. This will be seen more clearly later, when analysing the movement as a whole. Describing a property of a game to be high or low should be considered in relation to the games measured and does not necessarily extend to all games.

This framework is limited to basic movement. That excludes a range of special jumps such as wall jumps, double jumps, hovering jumps, backflips and so forth. There is a great variety of special moves which could be included in further studies. For this project, only the basic jump is considered. Moving on sloped terrain is common, but is similarly not included. Perhaps the most common feature left out from this framework is the ability to sprint by holding an additional input. Sprinting is possible in both *Super Meat Boy*, *Mario,* and many other platform games, and is often an essential part of the game. However, in the context of this project it would make measurement and analysis significantly more complex. Often, sprinting is the same as regular movement, just at a faster pace, but different games handle the transition between running and sprinting quite differently. With further work, sprinting would be an obvious aspect to include in the framework.

This framework is in no way exhaustive. It was designed to include the most common characteristics worth considering when designing a jump in a platform game. The properties selected represent what was needed to encompass the observations made for this project, and with further work additional properties should be added.
For each parameter of the framework it is possible to add further details. This could for example be variation over time or new factors influencing the scale of a given parameter.
With considerations and limitations established, we are now ready to present the framework.

Overview of the Movement Framework

Input
- Overshooting
- Deadzone
- Blowback

Ground Movement
- Maximum Ground Speed
- Acceleration
- Deceleration
- Turn Acceleration

Jump Takeoff
- Gravity
- Terminal Velocity
- Takeoff Velocity
- Horizontal Takeoff Velocity

Air Control
- Air Acceleration
- Air Friction
- Air Turn Acceleration
- Maximum Air Speed
- Releasing Horizontal Input

Jump Release
- Minimum Jump Duration
- Additive Jump Force
- Release Drag

Details
- Jump Cache
- Edge Jump
- Upwards Jump

Input

This chapter will focus on the analogue controller thumbstick, common to modern controllers and frequently used to control 2D platform games. The controller thumbstick will often be referred to simply as the stick. The input values one would expect from looking at the physical controller, differs in several ways from input actually received. This chapter will focus on these inaccuracies between assumptions and reality. The chapter was inspired by a series of blog posts by Hesselgren (2012), and seeks to confirm and extend his research.

A small tool was created in the Unity game engine to measure stick input. The tool was also used to create the illustrations used throughout this chapter. Measurements are taken by manually moving the stick around for several minutes, while continuously marking each input as a black pixel. The aim is to cover the whole input space evenly, as well as testing the outer maximum. The examples presented through this chapter were created using the XBox One controller as seen in figure 40. Towards the end of the chapter measurements from different controllers will be compared.

First the controller will be introduced in a little more detail. The thumbstick input from the controller is separated into a horizontal and a vertical axis, also referred to as *Horizontal Input* and *Vertical Input*. The value of each axis is represented as a value between -1 and 1. When the stick is untouched both horizontal and vertical input is zero. The horizontal input decrease when the stick is pushed left and increase when pushed right.

Figure 40: XBox One controller

Similarly the vertical input decrease when the stick is pushed down and increase when pushed up. The values of the two axes are combined into components of a vector. Speaking about the length of a given stick input refers to the length of this vector. In other words, how far the stick is moved away from its default central position. This will also sometimes be referred to as the size of the input or the amount of input. With this introduction to the controller we are right to get into the details.

Overshooting

As a vector, the input can be visualised as a point on a 2D surface, which is easy to mentally map to the physical controller. In figure 41, we see the raw input from the controller. As the physical restriction on the stick is circular, one might expect the input to also be a circle, as seen that is not entirely true. The input looks somewhat circular, but there is more going on.

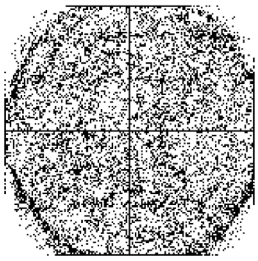


Figure 41: Raw stick input

Furthermore, if we rotate a fully pushed stick, we might expect to see a consistent input length of 1. This, again, would correlate to the physical construction of the controller, but turns out not to be the case. In figure 42, all input with a length greater than one is marked red. The input we expected is indeed included, but there is also a lot of input that overshoots the expected length. This overshooting input explains why we do not get the expected input.

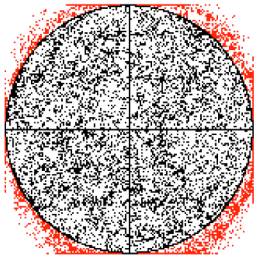


Figure 42: Overshooting

Ignoring this effect can cause problems. For example, when using the length of the input to determine the run speed of a character. The maximum possible speed should occur when the input length is one, but when the input overshoots, the character is suddenly running faster than expected. Overshooting will likely differ between controllers. This will result in the character running at different speeds depending on which controller is used.

The solution to overshooting is to clamp the input vector in the game before considering the input. In other words, scaling down any input vector with a length greater than one. The result can be seen in figure 43.

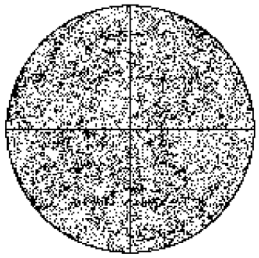


Figure 43: Clamped input

Deadzone

A *Deadzone* is used to ignore input under a given size. When the player is not pushing the stick, we expect both horizontal and vertical input to be zero. However, in practice this is not always the case. Controllers sometime give a consistent non-zero input without being touched, since the stick is leaning slightly away from the center position. This will often get worse and more likely the more the controller has been used, but can also happen with a new unused controller. Similarly, a player with a finger on the thumbstick and the intention of holding it completely still might give small input variations. In both cases, the deadzone will make sure the input is ignored.

Another use of a deadzone is to ignore small, but deliberate movements of the stick and only accept input of a sufficient size. The amount of input required to move the character will influence how the interaction feels. Which deadzone value is ideal depends on the game in question, so when designing, the deadzone is worth carefully considering. We will now describe and discuss different ways to implement a deadzone.

Square Deadzone

One way of implementing a deadzone is to clamp each input axis separately, creating a *Square Deadzone* as seen in figure 44. Here, horizontal and vertical input between -0.3 and 0.3 are clamped to zero. Input where one of the axes has been clamped, is marked in blue. The white square in the middle shows input which is entirely ignored. This means that diagonal input must be slightly larger before being accepted, compared to pushing the stick horizontally or vertically.

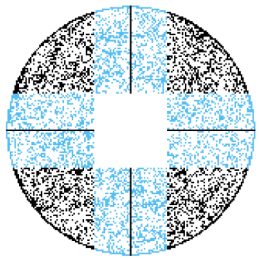


Figure 44: Each input axis

Another effect of this implementation is a tendency towards pure horizontal or vertical input. For example, given a large horizontal input and a small vertical input, the vertical part is clamped, leaving a purely horizontal input. The black horizontal and vertical lines visualise this tendency. When playing, this can feel like the controller snaps a little to the horizontal and vertical axes.

Circular Deadzone

Probably the most common way to implement a deadzone is using a *Circular Deadzone*. This solution considers the length of the input vector and discards input below a given threshold. A threshold of 0.3 is used in the example seen in figure 45. As such, input in any direction is treated the same.

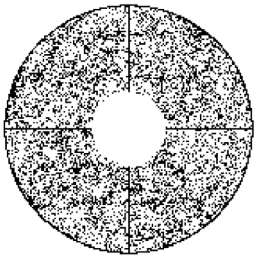


Figure 45: Circular deadzone

Custom Deadzones

Some games require the player to do precise horizontal sweeps when, for example, aiming in a first person shooter. In that case, the *Bowtie Deadzone* described by Sutphin (2013) could be considered. An example of such a deadzone can be seen in figure 46. It includes a circular deadzone, but on top of that, the vertical input is clamped according to a threshold. The size of this threshold goes up as horizontal input increases. This adjust the input towards a horizontal line, helping the player do the sweep motion, while still allowing small diagonal inputs.

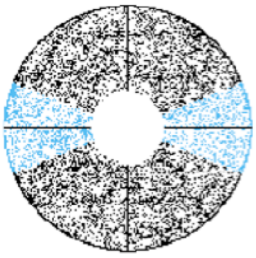


Figure 46: Bowtie deadzone

Figure 47 shows a *Directional Deadzone*. This implementation considers the angle of the input and adjusts it to the nearest of 8 directions. This mimics the technical limitations of a classic arcade controller. A similar deadzone is described by Hesselgren (2013) as *8 Way Analogue*.

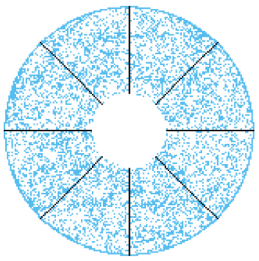


Figure 47: Directional deadzone

For further details regarding the technical implementation of the deadzones, as well as how to treat the input after the initial filtering see Hesselgren (2013) and Sutphin (2013).

Figure 48 presents deadzones measured for existing games. They all appear to use the circular deadzone. *Limbo* is the only game measure where movement speed is influenced by input size. Therefore, a lower deadzone makes sense to get the largest input span possible. The input in *Limbo* also goes through a complicated process as described in Appendix 1.

Limbo	*0.358*
Super Meat Boy	*0.5*
Braid	*0.502*
Spelunky	*0.64*

Figure 48: Deadzones from existing games

The other three games have similar deadzones with *Spelunky* (Mossmouth, 2013) being slightly larger. Based on these measurements, the common threshold appears to be around 0.5. *Mario* is not included, as it uses the NES controller which does not support analogue input.

Blowback

A phenomenon, which we have chosen to call *Blowback,* was found during this project. When a thumbstick is released a spring pulls it back to its default centre position. Sometimes the stick will go fast enough to bounce briefly past the centre before settling, resulting in a blowback. This will be most noticeable if the thumbstick is fully pushed and then suddenly released. When this happens, the stick can give some unwanted input for one or two frames. Depending on the size of the deadzone, as well as how movement is implemented, this blowback can cause problems.

Consider, for example, the following very common case. A player is running right and decides to stop by removing the finger from the controller. The stick is now pulled back by the spring and potentially causes blowback. If the input resulting from the blowback is not caught by the deadzone, and the game is implemented to be as responsive as possible, the character will start going left. Since this input is very brief, no significant movement will take place. However, it might be enough to visually turn the character the other way. Depending on the graphics of the game, this could visually be a big change and

potentially result in a turn animation. Blowback does not happen consistently and depends on the type of controller and how it is used by the player. However, having the character randomly turn around is probably not desirable.

Speaking from real-world experience, changing from the XBox 360 controller to the XBox One controller during a game production aroused suspicion. After the change, the character would sometimes do unwanted turns after the thumbstick was released. This lead to further exploration under the assumption that the new controller had a larger blowback. Measuring the effect showed that this was indeed the case and the blowback turned out to be much larger than expected.

Blowback was measured in four directions: up, down, left, and right. In general, the controllers seem to slightly favour the purely horizontal and vertical axes. This can be seen as black lines in the visualisation of the input. This is likely caused by the way the springs in the controllers are constructed. Figure 49-51 show the data collected for the different controllers.

Controller: XBox One

Largest overshoot:	1.264	**Largest input at rest:**	0, 0

Blowback

Horizontal Min:	-0.279	**Horizontal Max:**	0.777
Vertical Min:	-0.634	**Vertical Max :**	0.438

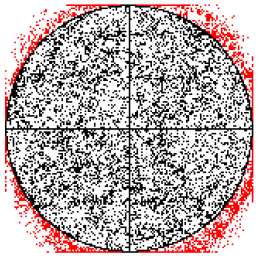
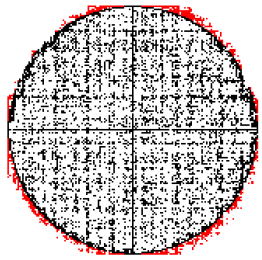


Figure 49: Measurements from the Xbox One controller

Controller: PlayStation 4

Largest overshoot: 1.100 **Largest input at rest** : 0, 0



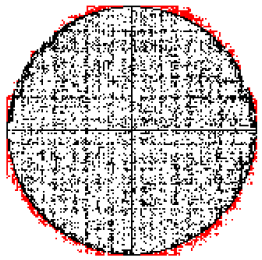
Blowback

Horizontal Min:	-0.458	**Horizontal Max:**	0.264
Vertical Min:	-0.264	**Vertical Max :**	0.429

Figure 50: Measurements from the PlayStation 4 controller

Controller: Xbox 360

Largest overshoot: 1.277 **Largest input at rest** : 0, -0.053



Blowback

Horizontal Min:	-0.190	**Horizontal Max:**	0.198
Vertical Min:	-0.134	**Vertical Max :**	0.000

Figure 51: Measurements from the Xbox 360 controller

Comparing blowback on the XBox 360 controller with the XBox One controller confirms the previously mentioned assumption. The highest blowback from XBox One is 0.777, much higher than 0.198 from the XBox 360. A similarly high blowback of 0.458 was measured on the PlayStation 4 controller. This data shows worst case scenarios and does not indicate how often blowback occurs in practice. The wear of a controller might also influence the blowback, and further testing with comparison of new and old controllers could show how a controller changes over time in this regard.

Based on this limited data, blowback appears to be worth considering and testing for when creating a game. Using an Xbox One controller, the blowback reaction of a range of games were briefly tested. The games tested were *Super Meat Boy*, *Limbo*, *Spelunky*, and *Braid.* The test consisted of running left and releasing the thumbstick. This was repeated several times. If blowback is not handled, the character is expected to occasionally flip and face right. This turned out to be the case for all games measured. This could indicate that the problem is new and caused by recent controllers. Alternatively, the developer might not be aware of the problem, or might have chosen to ignore it.

It could also be that the problem is challenging to solve. Using a deadzone as high as 0.7 to prevent blowback is not a viable solution, as this would drastically limit the input area available. One potential solution could be to only look at consistent input and to ignore input only sent for one or two frames. This makes the game less responsive, which would not normally be acceptable. Most potential ways of fixing this problem would depend on how the movement in a given game is implemented. Further research on blowback is required for more specific solutions to the problem. With a better understanding of input we are ready to dive into movement.

Ground Movement

Figure 52: Meat Boy running right

This chapter covers the movement of the character when on the ground. It is divided into the following properties: *Maximum Ground Speed*, *Acceleration, Deceleration,* and *Turn Acceleration*.

Maximum Ground Speed

This represents the fastest possible horizontal speed of a character while on the ground. When a horizontal input is given in *Castlevania* the character moves forward at his *Maximum Ground Speed* as seen in figure 53. As long as input is given Simon moves at a constant speed, when input is released he immediately stops. As such he instantly accelerates and decelerates.

The vertical orange line indicates when input was pressed, and the green line when it was released. This reveals a subtle delay of 2 frames between input and movement. This might feel slightly like acceleration, but is likely a technical quirk rather than a conscious design decision.

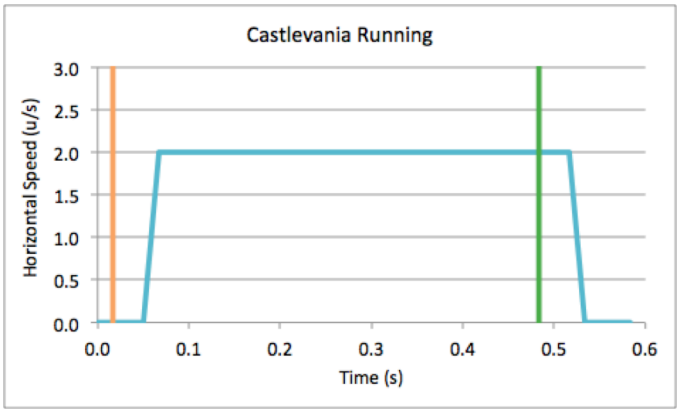


Figure 53: Horizontal speed in Castlevania

Acceleration

When horizontal input is given *Acceleration* is the rate at which the horizontal speed of a character increases.

Such an acceleration is used in *Super Meat Boy*. Figure 54 shows the horizontal speed of the character when running right. Meat Boy accelerates and gradually reaches his maximum ground speed. If the horizontal input is released, he stops immediately from one frame to the next. This sudden stop also occurs when landing after a jump, and results in landings without any skidding.

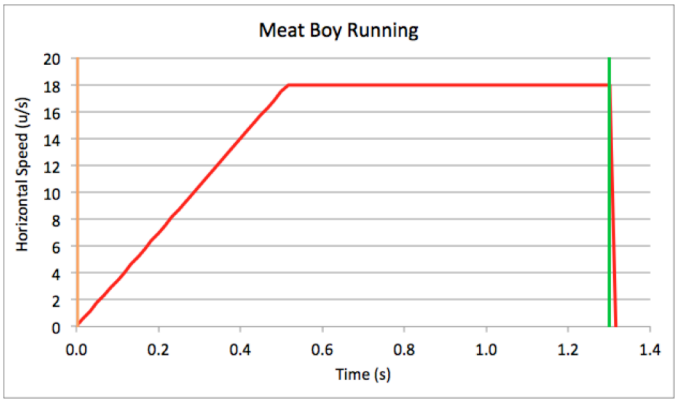


Figure 54: Horizontal speed in Super Meat Boy

Deceleration

When no input is given *Deceleration* is the rate at which a character comes to a halt.

Figure 55 shows the horizontal speed of Mario when running right. At the start, Mario can be seen accelerating towards his maximum ground speed. When input is released, his deceleration gradually brings him to a halt. For Mario, acceleration and deceleration happens to be the same rate, but this is not always the case.

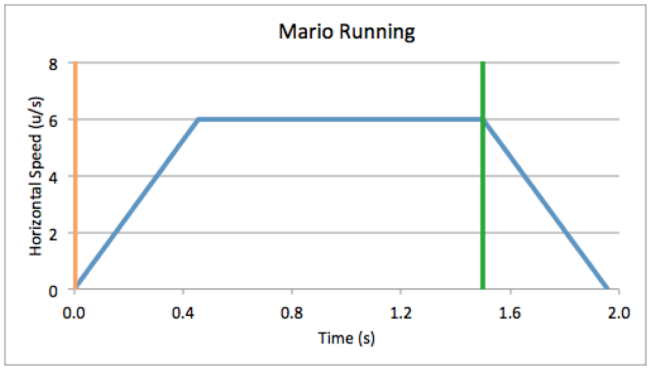


Figure 55: Horizontal speed in Mario

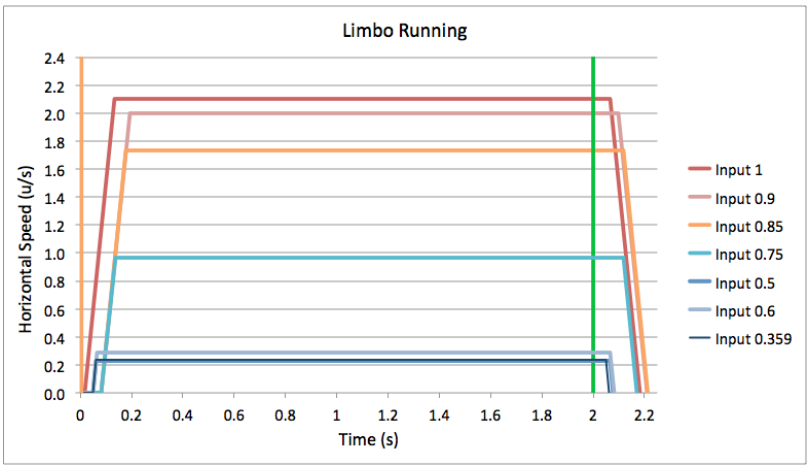


Figure 56: Horizontal speed in Limbo

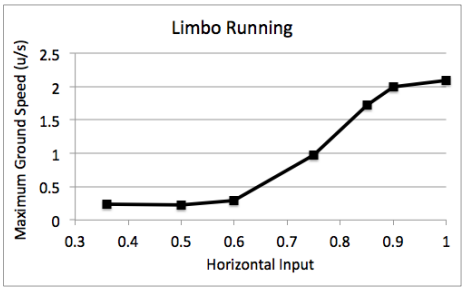


Figure 57: Relationship between input and speed in Limbo

The horizontal input is treated as binary in *Super Meat Boy*, *Mario*, and *Castlevania*. *Limbo,* on the other hand, has an analogue maximum ground speed. A higher horizontal input leads to a higher run speed as seen in figure 56. This allows the player to only slightly push the stick and as a result walk forward slowly. Pushing the stick in this manner will be referred to as *Feathering* the stick. The relationship between input and speed is not linear, but follows a curve as seen in figure 57. The input curve transitions between a lower plateau and an upper plateau. The plateaus support a feeling of consistency in the input. If there was no lower plateau the minimum speed would only be attainable near the deadzone, however this could easily slip inside the deadzone losing input all together. Furthermore, horizontal stick input is rarely perfectly aligned, but instead angled slightly. The result is that a fully pushed stick might have an input size slightly below 1. Here the upper plateau creates consistency as input within the range of the plateau will result in the maximum speed. For a more detailed breakdown of this curve see Appendix 1.

Looking again at figure 56, we also see that *Limbo* treats acceleration and deceleration differently. The transitions all take less than 0.2 seconds. Furthermore, they appear only partially related to the current horizontal input. In this way, they act more like animation transitions with a predefined duration and less like an acceleration.

The maximum ground speed in *Limbo* is much lower than *Mario* and *Super Meat Boy*, so accelerating and decelerating quickly does not feel abrupt. The vertical orange line shows horizontal input being pressed, and the green line shows input being released. Like with *Castlevania* we see a brief delay between input and reaction. When input is pressed there is a delay between 3-5 frames, and when released a delay between 3-7 frames.

Edge Aware Deceleration

If a character does not stop instantly, but decelerates to a halt after input has been released, there is a risk of sliding into danger or off a platform. This might be exactly what the player tried to avoid when releasing the input. To prevent this, some games check if danger or edges are nearby and apply a stronger deceleration or stop the character instantly. This *Edge Aware Deceleration* is for example used in *Toto Temple Deluxe* (Juicy Beast, 2015), where the character instantly stops instead of sliding off ledges or into danger. This is described in further detail by the developers in their blogpost about the movement in the game (Alexandre, 2014).

Turn Acceleration

This type of acceleration is used if the player change direction of the stick input while the character is already moving. If the direction of the input is opposite to the direction of the character's current movement, a *Turn Acceleration* can be applied instead of the regular acceleration. Turn acceleration could be seen as acceleration and deceleration combined. Turn acceleration controls how far the character will skid while decelerating. For the measured games, this acceleration is applied until the the speed of the character reaches zero. From here, the normal acceleration takes over and accelerates the character in the new direction. Increased acceleration when turning is a well known detail, for example described by Pignole (2014) as a Reactive Percentage.

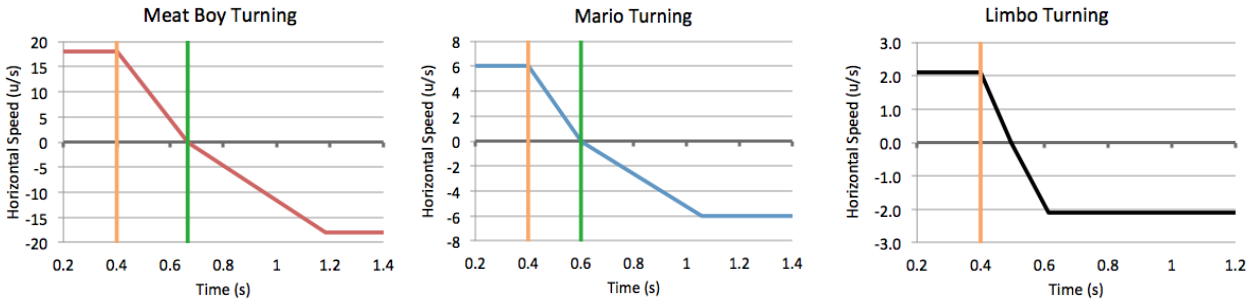


Figure 58: Horizontal speed when turning and changing direction in Super Meat Boy, Mario and Limbo

Figure 58 shows the turn acceleration in the three case studies. The orange vertical lines indicate when input was changed from right to left. The green lines indicate when the character's speed reaches zero. From here on, normal acceleration is applied instead of turn acceleration. *Mario* is an example where we see a turn acceleration larger than acceleration and deceleration combined. The result is that Mario turns faster and skids less. This results in a slightly more responsive reaction when turning compared to just decelerating when input is released. A different approach is seen in *Braid,* where turn acceleration is only slightly larger than deceleration. This gives the turns a more floaty feeling than in *Mario.* On the other hand, deceleration in *Braid* is higher than in *Mario*. Mario slowly slides to a halt across 0.457 seconds whereas Tim from *Braid* quickly stops in just 0.17 seconds. Treating deceleration and turn acceleration as separate properties allows direction change and deceleration to be adjusted independently.

The instant brake and the turn acceleration measured for Meat Boy appear to be in conflict. The character comes to an instant standstill when input is released for as little as one frame. However, opposite input activates turn acceleration, and the character skids for 2.16 units, decelerating across 0.27 seconds. This implies that releasing input for one frame before changing direction will prevent skidding and thus lead to a faster turn. If the player is going towards a ledge and wishes to avoid falling off, it would therefore be a better choice to release input than to give an opposite input. It is doubtful that many players will notice or be able to exploit this subtlety, but in *Super Meat Boy*, common input can still result in both fast turns and skidding turns.

This has different implications when playing with a keyboard rather than an analogue controller stick. If the player uses just one finger, the time between releasing a key and pressing the next is enough to trigger the instant halt. However, most players will use multiple fingers and here the skidding turn becomes relevant. When transitioning between directions in this fashion, the player will often briefly be holding the keys corresponding to both directions. This will avoid the brake condition and result in a skidding turn. If the player does not press the keys simultaneously, but have at least one frame between inputs, she will of course get the fast turn as described when playing with one finger. If both horizontal direction keys are held down continuously, the game gives priority to the left input, regardless of which direction was pressed first.

On the other hand, using an analogue controller thumbstick will always result in a fast turn. Moving the stick between left and right will leave at least one frame with the stick in the deadzone, enough to trigger the instant brake. Interestingly, the Input Simulator can still perform skidding turns with the thumbstick by instantly changing directions between frames. As we shall later see, the turn acceleration is still relevant for a human player, but requires that the character is in the air when initiating the turn.

The following chart summarises the parameters of ground movement from the games mentioned throughout the chapter.

	Meat Boy	*Mario*	*Limbo*	*Castlevania*	*Braid*
Max Ground Speed u/s	18	6	0.226 - 2.1	2	3.55
Acceleration u/s^2	35	13.125	-	-	10
Acceleration Duration s	0.516	0.457	0.06 - 0.2	0.016	0.35
Deceleration u/s^2	-	13.125	-	-	23
Deceleration Duration s	0.016	0.457	0.06 - 0.2	0.016	0.17
Turn Acceleration u/s^2	70	30	-	-	26
Skid Distance u	2.157	0.458	0.118	-	0.158
Turn Duration s	0.782	0.657	0.23	0.016	0.47

Figure 59: Parameters and derived measurements for ground movement

Jump Takeoff

This chapter describes the jump takeoff and will present the properties *Gravity, Terminal Velocity* and *Takeoff Velocity.* For now, we will assume that the player holds jump input and horizontal input throughout the entire jump. The results of releasing input during the jump will be explored in later chapters.

Gravity

Gravity is the rate at which a character accelerates towards the ground when falling. The fall speeds measured will include both gravity and air resistance.

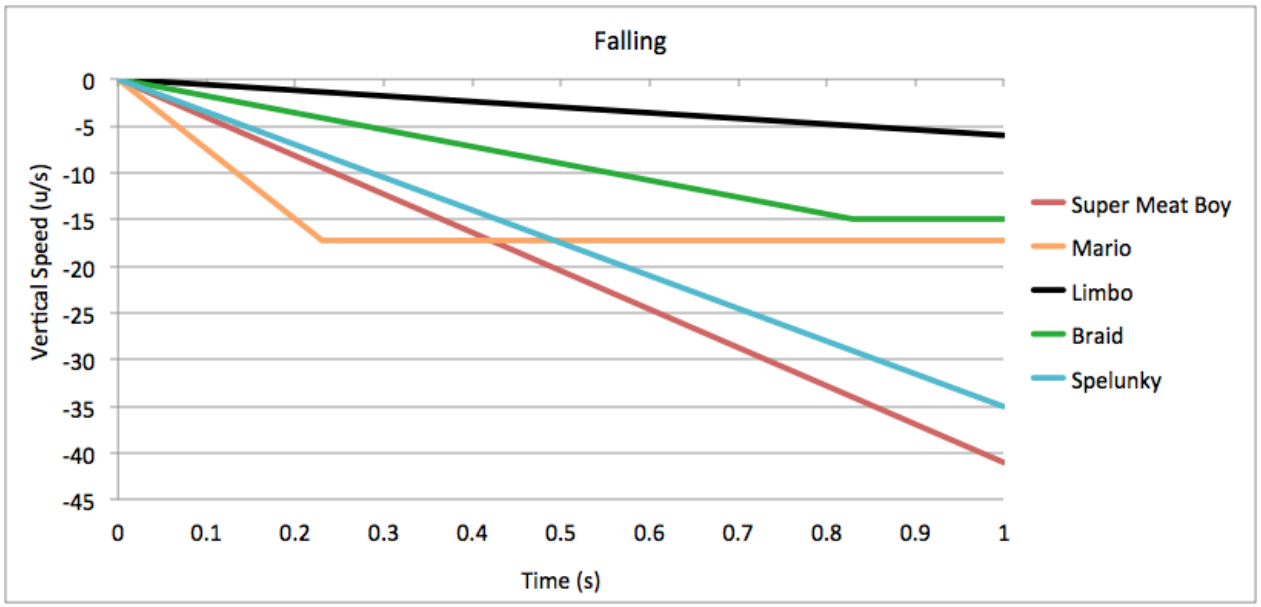


Figure 60: Vertical speed of game characters falling

Figure 60 shows the vertical speed of falling characters. The straight lines show a constant rate of acceleration, this indicates that the games have no air resistance, at least not in the traditional sense. On earth air resistance is dependent on fall speed. In other words, falling faster leads to greater resistance. If this kind of air resistance was present in the games the curves would be bending instead of going straight. Therefore, it seems fair to disregard air resistance and just consider the measurements as gravity.

Comparing the games, we see they have quite different values for gravity. This means the characters will fall at different speeds. Gravity is essential and influences most other aspect of a jump.

Terminal Velocity

Mario does not have an increasing air resistance, but he does have a maximum fall speed. After 0.230 seconds, the vertical speed of Mario stops increasing. This can be seen in figure 60 at the point where the curve changes from a downwards slope to a horizontal line. This maximum fall speed is also called *Terminal Velocity,* and is the highest attainable velocity of a falling object. As mentioned previously this is reached gradually in the physical world as a result of increasing air resistance. It takes a falling human roughly 15 seconds to reach 99% of terminal velocity. Mario reaches this as a cutoff instead of gradually, and it takes him just 0.230 seconds. This is fast enough to occur during most jumps in the game. Reaching terminal velocity during a jump gives a more linear fall, resulting in a slightly longer jump distance. This will be more apparent later when considering the complete jump curves. As seen in figure 60 *Braid* also has a terminal velocity, again implemented as a cutoff. However, only when falling for more than 0.83 seconds does the character reach this terminal velocity. A descend from a common jump in the game last just 0.342 seconds. This means terminal velocity will only influence longer falls and have no influence on the majority of jumps and smaller falls. Measuring some of the longest falls in *Super Meat Boy* and *Limbo*, shows no terminal velocity.

Takeoff Velocity

To make a character jump, a vertical force is added. This force will be referred to as the *Takeoff Velocity*. As we will see, this force can be influenced by various factors, for example horizontal speed or traversed distance.

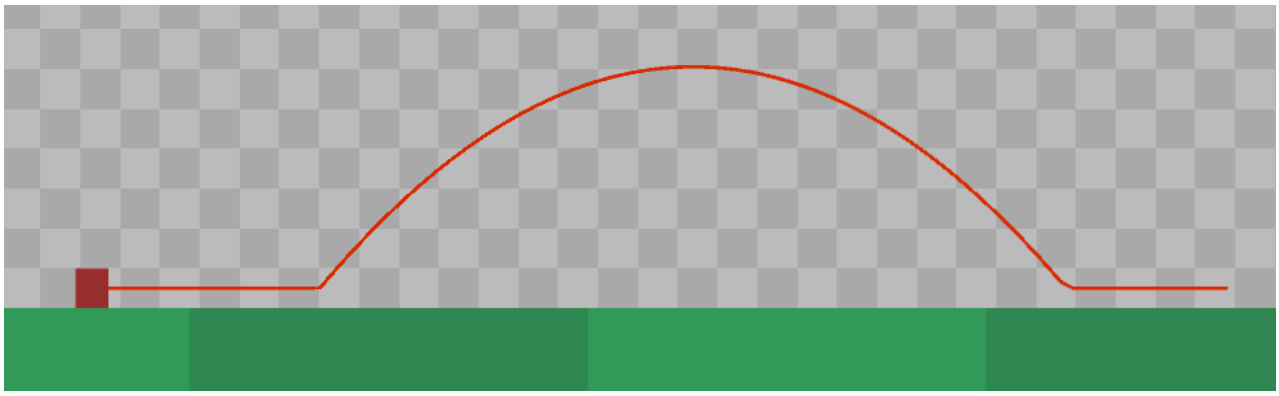


Figure 61: Meat Boy performing a running jump

When Meat Boy jumps he gains a vertical speed of 21 u/s. The rest of the jump is simply gravity working against this initial velocity. As seen in figure 61 Meat Boy simply reaches the peak of his jump and accelerates towards the ground until he lands.

Horizontal Speed (u/s)	Takeoff Velocity (u/s)
Speed > 12	15.75
Speed > 8	14.75
Speed > 4	14.25
Speed < 4	13.75

Figure 62: Takeoff Velocities in Mario

In *Mario*, the takeoff velocity is higher the faster he is running at the moment of takeoff. *Mario* has a total of four different takeoff velocities. The run speed required for each of the velocities can be seen in figure 62. The jump curves resulting from the two smallest velocities are compared in figure 63. The light blue curve shows the smallest possible takeoff velocity. This is also the velocity applied if Mario jumps while standing still. If Mario is running faster than 4 u/s, a higher velocity is used, resulting in the dark blue jump curve. The two largest jump velocities are only obtainable when sprinting.

As the different steps are not explicit in the game, the player is likely to perceive them as fluently transitioning from one to the next, rather than perceiving the actual intervals. The player will probably notice that running faster results in higher and thereby also further jumps. This means that when preparing for long jumps, backing up makes sense. This behaviour can create a moment of tension as the player accelerates towards the edge of a bottomless pit, trying to reach sufficient speed before jumping.

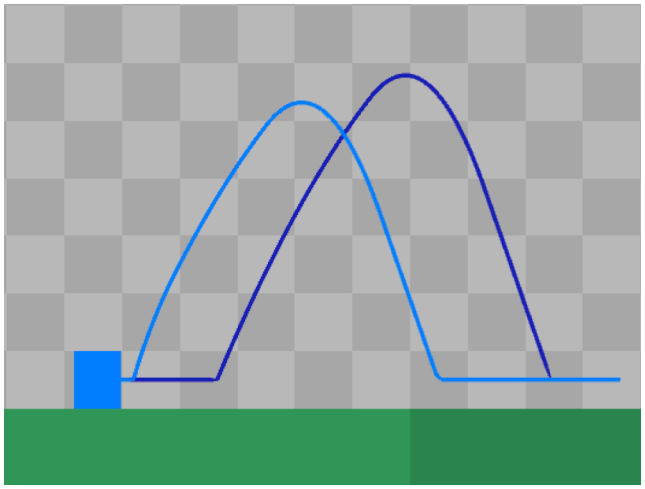


Figure 63: Two different takeoff velocities in Mario

Since Meat Boy is always given the same takeoff velocity regardless of his running speed, this tension of accelerating towards the edge is not present in the same way. In contrast, the consistent takeoff velocity in *Super Meat Boy* allows fast consecutive jumps. The player does not have to prepare each jump, but can rely on the character's ability to do a jump with maximum strength at any time.

Limbo has a more dynamic takeoff velocity. Like in *Mario,* the takeoff velocity is influenced by horizontal speed. Higher horizontal speed results in higher takeoff velocity. This relationship is fluent and not discrete. Figure 64 shows the span of possible jumps. As previously mentioned, the boy in *Limbo* accelerates quickly and reaches maximum ground speed in just 0.06 - 0.2 seconds. Therefore, the effect of horizontal speed is mainly noticed when feathering the stick, resulting in a consistently low run speed.

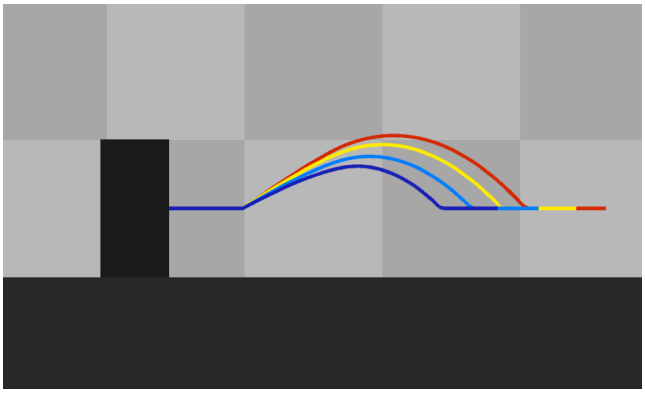


Figure 64: Limbo jumping with different run speed

In *Limbo,* the distance the character has been running since standing still also influences the jump. When jumping, the game checks if the character has run more than 5u. Any jump before this distance is considered a short jump, and 0.1u/s subtracted from the takeoff velocity. Figure 65 shows the difference between a short and a long jump.

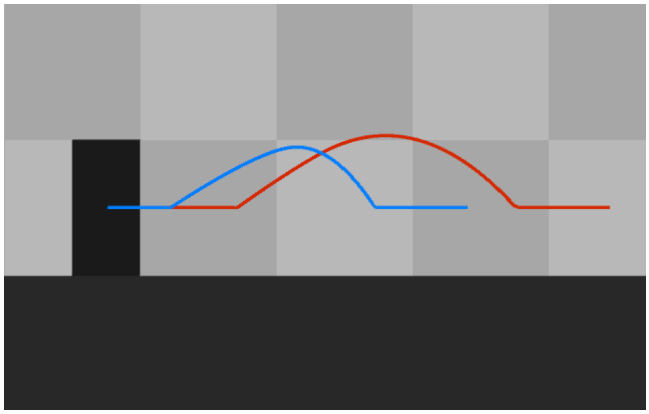


Figure 65: Short and long jumps in Limbo

The distance requirement means that jumping when standing still or after running very briefly will feel weak. Like in *Mario*, this encourages the player to backup and prepare before doing a jump. However, since the distance required to do a long jump in *Limbo* is much shorter than *Mario*, almost any attempt to run before jumping will result in a long jump.

Horizontal Takeoff Velocity

Amongst the measured games *Limbo* is the only game to have a *Horizontal Takeoff Velocity*. This velocity is not influenced by horizontal speed or traversed distance, but is a consistent 3.5 u/s. This means the jumps have a direction instead of just being an upwards force. The horizontal takeoff velocity is actually higher than the vertical velocity. The result is that the character is much better at jumping forwards than upwards.

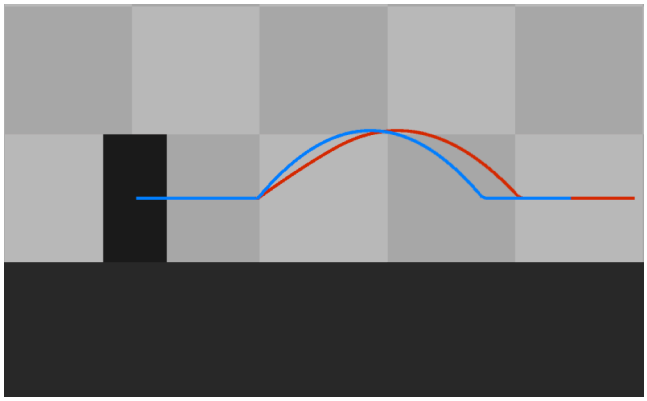


Figure 66: The impact of horizontal takeoff velocity in Limbo.

Figure 66 shows the impact of the horizontal takeoff velocity. The red curve is a regular jump and the blue curve a jump without the horizontal takeoff velocity.
The effect might seem subtle, however when looking at air control in the next chapter the significance of this velocity will become clear.

Figure 67 and figure 68 present an overview of the jumps measured in this chapter and figure 69 shows the parameters.

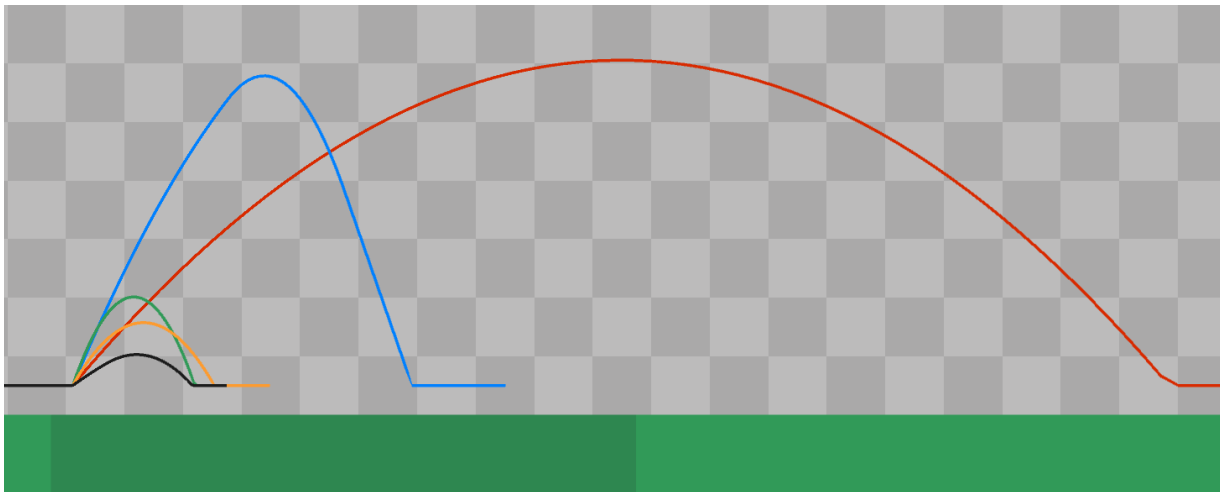


Figure 67: Red: Super Meat Boy, Blue: Mario, Green: Spelunky, Orange: Braid, Black: Limbo

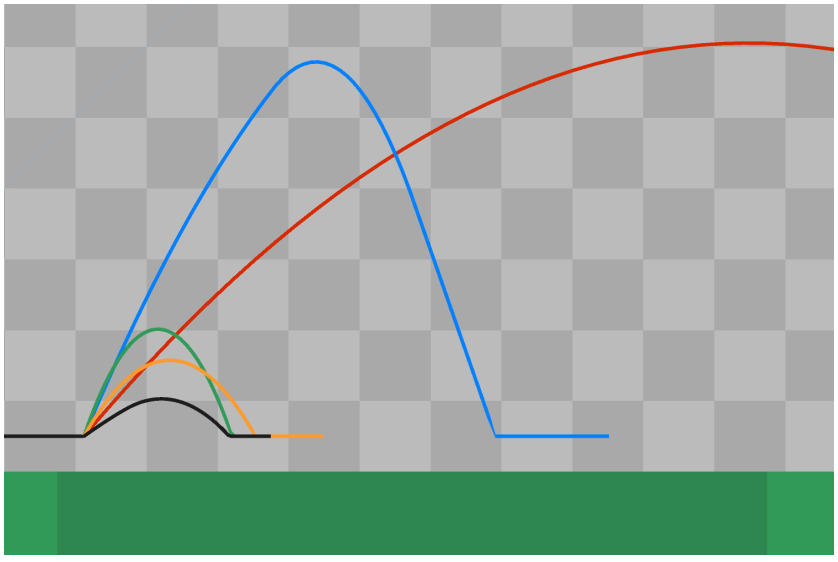


Figure 68: Red: Super Meat Boy, Blue: Mario, Green: Spelunky, Orange: Braid, Black: Limbo

	Meat Boy	*Mario*	*Limbo*	*Spelunky*	*Braid*
Gravity u/s^2	41	75	6	35	18
Terminal Velocity u/s	-	17.25	-	-	14.5
Takeoff Velocity Vertical u/s	21	13.75 - 15.75	1.78 - 2.47	10	6.15
Takeoff Velocity Horizontal u/s	-	-	3.5	-	-
Maximum Jump Height u	5.554	5.287	0.529	1.512	1.074
Maximum Jump Distance u	18.599	5.7	2.037	2.071	2.367
Maximum Jump Duration s	1.05	0.967	0.85	0.6	0.683

Figure 69: Parameters and derived measurements for jump takeoff

Air Control

This chapter will look at how horizontal input influences the movement of the character while in the air. It will have some similarities to the previous chapter on ground movement. The properties presented will be *Air Acceleration*, *Air Friction*, *Air Turn Acceleration, Maximum Air Speed* and *Releasing Horizontal Input*.

Air Acceleration

When horizontal input is given by the player, *Air Acceleration* is the rate at which the horizontal speed of the character increases while in the air.

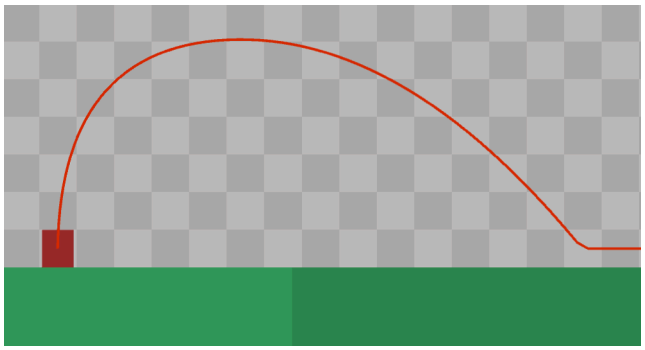


Figure 70: Air acceleration in Super Meat Boy

Figure 70 shows a jump curve from *Super Meat Boy* which illustrates air acceleration. The character initiates a jump without any horizontal speed or horizontal input. Right after takeoff, full horizontal input is given. While in the air, the horizontal speed of the character increases. Both Meat Boy and Mario accelerates at the same rate in the air as on the ground. This is not always the case, for example, in *Sonic the Hedgehog* (Sonic Team, 1991), the air acceleration is twice as high as the acceleration on the ground (Sonic Retro, 2014).

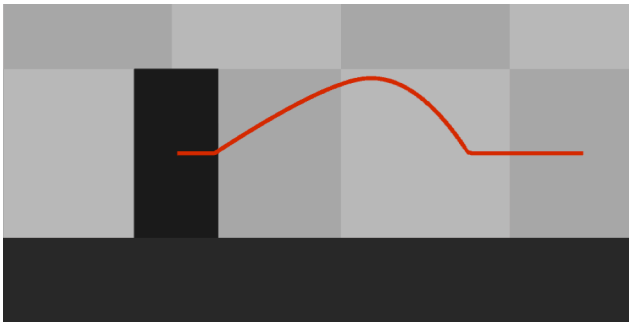


Figure 71: Limbo horizontal takeoff velocity

Games which applies a horizontal takeoff velocity will likely have less, if any, use of air acceleration. This is for example the case with *Limbo.* When a jump begins, the horizontal takeoff velocity causes an immediate speed increase and no further air acceleration is needed.

This highlights the effect of horizontal takeoff velocity, a concept discussed in the previous chapter. Figure 70 shows the jump in *Super Meat Boy*. Here, the air acceleration results in a jump curve gradually being directed forward. The jump from *Limbo* can be seen in figure 71. Notice how the horizontal takeoff velocity directs the jump forward right from the start.

Air Friction

A character's rate of deceleration while in the air is called *Air Friction*. It is a deceleration of the horizontal speed and is applied despite horizontal input.

Neither Meat Boy nor Mario are subject to air friction and will continue forward without decelerating at all. If no horizontal input is given, the speed at takeoff is preserved. Any additional speed gained by acceleration in the air is likewise preserved.

Limbo, on the other hand, applies an air friction. The effect can be seen in figure 72. The red curve is the jump as it exists in the game and the blue curve represents a hypothetical jump without air friction. Without friction the jump is, as expected, significantly longer.

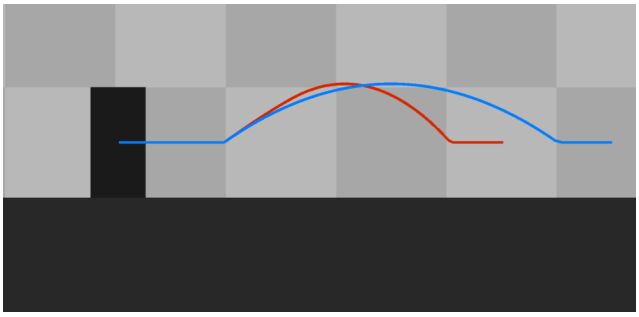


Figure 72: Limbo without air friction

This makes sense if we consider the horizontal takeoff velocity. This velocity is larger than the vertical takeoff velocity and will lead to the long jump seen in blue. Air friction limits this reach by gradually decelerating the character in the air. This gives a different effect than simply lowering the horizontal takeoff velocity. Instead of a consistent speed, this jump will have an initial burst of speed and then gradually decelerate. Having two factors to adjust also gives more freedom when designing the jump. If the horizontal takeoff velocity is increased, air friction can be adjusted accordingly to avoid changing

the length of the jump, resulting in an increased burst of speed and a more aggressive friction.

Landing in *Limbo* without air friction would also not be ideal and would feel abrupt. Right after landing, maximum ground speed would clamp the horizontal speed resulting in a sudden deceleration. However, the deceleration caused by the friction brings the speed down which leads to a smooth transition when landing.

Air Turn Acceleration

This property is similar to turn acceleration, but used when the character is in the air. *Air Turn Acceleration* is the rate at which the character accelerates when given an input opposite to the direction of the current speed. Meat Boy and Mario both use the same air turn acceleration as the turn acceleration they used on the ground. This makes for very consistent controls as being in the air neither benefits nor hinders horizontal movement. The player will likely not notice that acceleration in the air is the same as on the ground. However, it might be more noticeable if it was not the case. Because of the vertical speed during a jump the total speed of the character while in the air can actually be faster than on the ground,

As previously mentioned, Meat Boy has some interesting turn behaviour on the ground caused by deceleration and turn acceleration conflicting. This is not the case in the air, as Meat Boy has no air friction, air turn acceleration can be applied unhindered. However, changing direction before or right after a landing results in different movement and is worth discussing. If the player changes direction right before a landing, air turn acceleration will briefly be applied while in the air. After landing, turn acceleration will take over and the character will skid on the ground before finally accelerating in the opposite direction. If, instead, the player lands and waits one frame before turning, the instant ground deceleration prevents skidding. This will result in a faster more precise turn.Whether players will consciously notice this difference is unclear, but it will influence the feel and control of landing.

Consider a case where the player wishes to land on a small platform, about 1 unit wide. The player starts running, and then jumps. In the air, the player aims the trajectory at the platform, and the character is coming down to land.

In *Super Meat Boy*, the best option is to release horizontal input. Since there is no air friction this will not affect the trajectory. After landing on the platform, the character will stop immediately. Trying to adjust the horizontal speed before landing could result in skidding and potentially falling off the platform.

In contrast, Mario skids 1.226 units when landing from a jump, enough to skid off the platform. The player is required to give opposite input in order to decelerate and stay on the platform. This input can be given both before and right after landing. Here, air turn acceleration and turn acceleration gives the player the ability to control the landing. A different way to control the landing, which is possible in both games, is to overshoot the platform and decelerate in the air while above the platform. This results in a more vertical landing with less horizontal speed. However, such a landing is not always easy and might not be possible, depending on the level design.

Yet another way to deal with landings, which both games could be said to encourage, is to not stay on the platform, but maintain forward momentum. Instead of decelerating, the player would land briefly on the platform and perform a new jump or intentionally run off the platform, continuing forward through the level. More experienced players are expected to master this kind of maneuvers.

Maximum Air Speed

Fastest possible horizontal speed while the character is in the air. Meat Boy and Mario both have the same *Maximum Air Speed* as their maximum ground speed. Having a consistent maximum speed makes implementing the transitioning between air and ground easier, since speed does not need to be adjusted when the character changes state.

If a jump has a horizontal takeoff velocity, the maximum air speed will probably be larger than the maximum ground speed. If this is not the case, horizontal takeoff velocity will often have no effect. If, for example, the character is moving at maximum ground speed when jumping, the horizontal takeoff velocity will have no effect since the speed cannot increase further.

Looking for example at *Limbo,* there is no maximum air speed at all. Without air acceleration there is no need to limit the horizontal speed in the air. Whichever velocity is applied at takeoff is accepted and only decelerated by air friction as usual.

Limbo does, however, have a *Minimum Air Speed*, which limits how low the horizontal speed can be while in the air. Figure 73 shows the effect of minimum air speed. The red curve is the jump as it exists in the game, and the blue curve is a hypothetical jump without minimum air speed. This minimum does not appear to be constant but is instead slightly influenced by the horizontal input. So, even without any air acceleration, different amount of horizontal input can still slightly adjust horizontal speed while in the air.

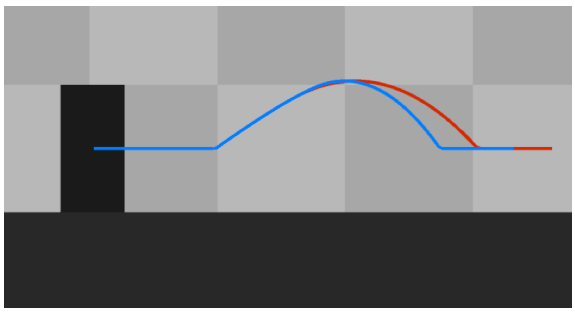


Figure 73: Minimum air speed in Limbo

Releasing Horizontal Input

How does releasing horizontal input influence the character's movement in the air? In both *Super Meat Boy* and *Mario*, releasing horizontal input at any point after the takeoff does not affect horizontal speed.

If horizontal input is released during a jump in *Limbo,* an increased air friction is applied. This significantly changes the jump curve as seen in figure 74. The vertical line indicates when horizontal input was released. Horizontal input in the opposite direction to the current jump is handled as if input was released.

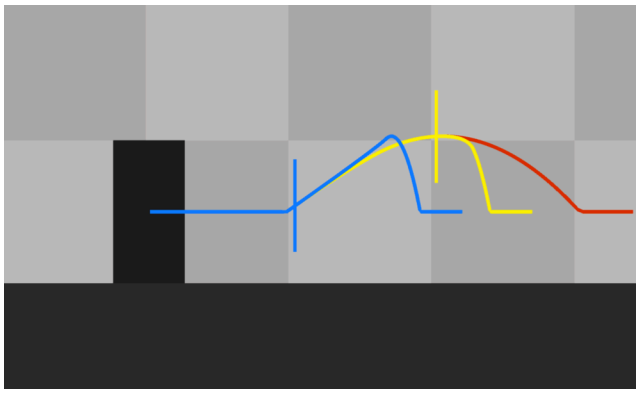


Figure 74: Releasing horizontal input in Limbo

If this increased friction has been applied, a new horizontal input will have no effect before the character lands. This makes for a more smooth jump, as the player cannot go back and forth between different frictions.

To summarise this chapter, figure 75-77 show the span of possible air movement in each of the games. The red curves show jumps with maximum horizontal input. Blue curves show jumps where the horizontal input direction was changed right after takeoff. The yellow curves show jumps where the horizontal input direction was changed halfway through the jump. Depending on when input direction is changed, the character can land at any position between the red at the blue curve.

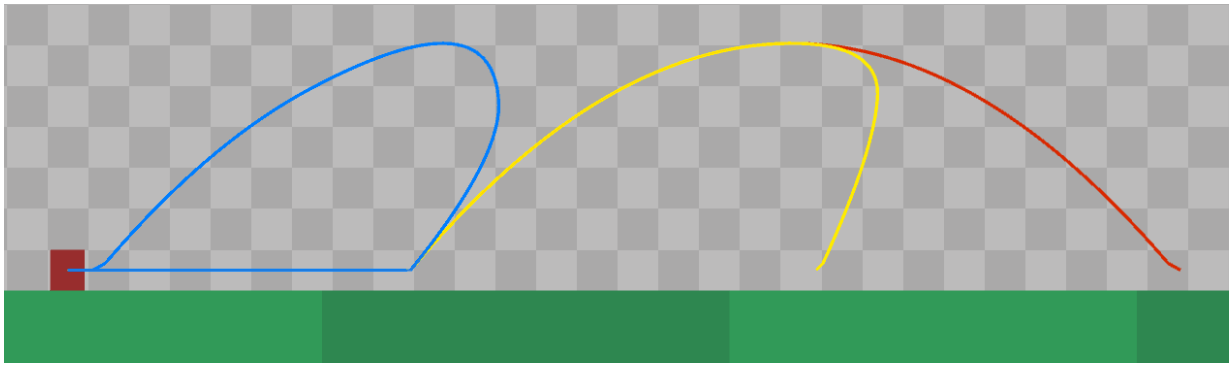


Figure 75: Air control in Super Meat Boy

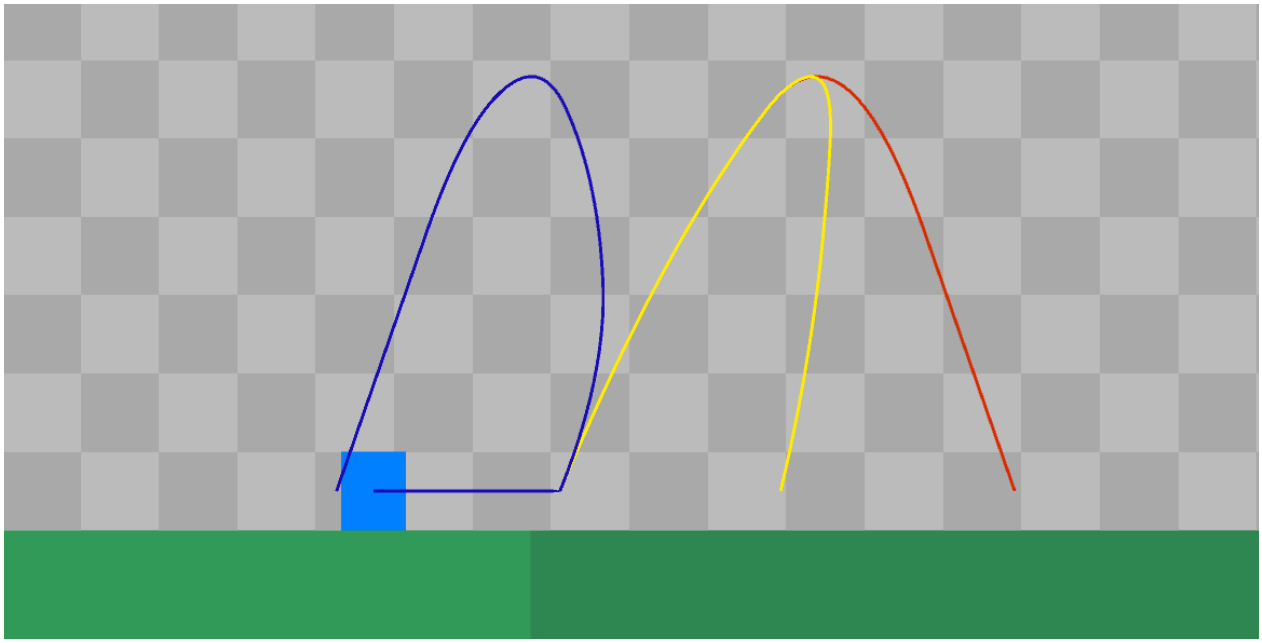


Figure 76: Air control in Mario

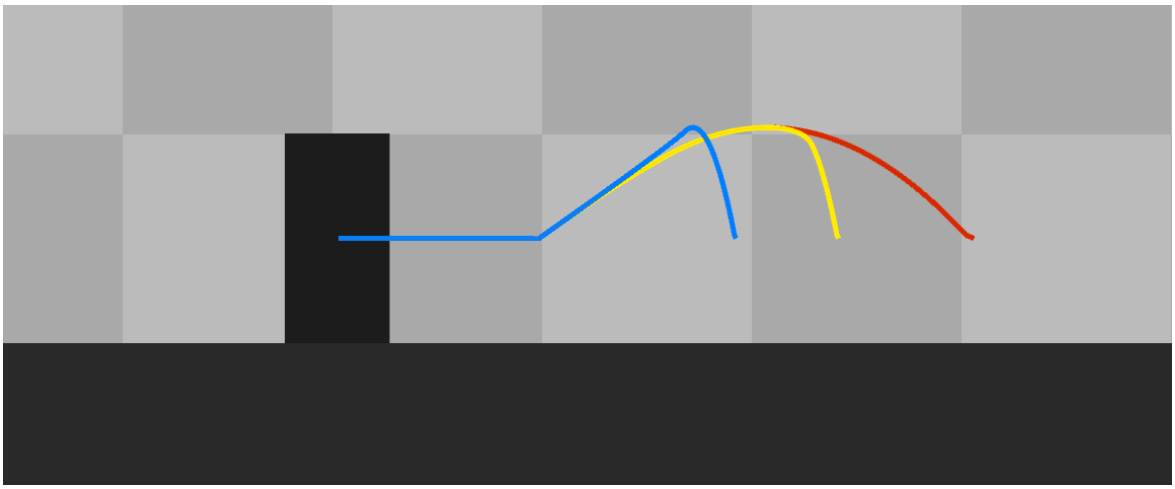


Figure 77: Air control in Limbo

	Meat Boy	*Mario*	*Limbo*
Air Acceleration u/s^2	35	13.125	-
Horizontal Air Friction u/s^2	-	-	6
Air Turn Acceleration u/s^2	70	30	-
Maximum Horizontal Air Speed u/s	18	6	-
Minimum Horizontal Air Speed u/s	-	-	1.5 - 2.284
Jump Input Released			
Horizontal Air Friction u/s^2	-	-	9
Minimum Air Speed u/s	-	-	0.784

Figure 78: Parameters for air control

Jump Release

This chapter will look at how holding or releasing the jump input influence the jump of the character. Allowing the player to perform small and large jumps depending on how long jump input is pressed was common with the games explored for this project. While looking at different ways to implement this a number of properties will be presented including *Minimum Jump Duration*, *Additive Jump Force,* and *Release Drag.*

In some games releasing jump after takeoff has no effect on the movement of the character. This is for example the case in *Braid* and *Feist*. This results in very consistent jumps, but takes some control and flexibility away from the player.

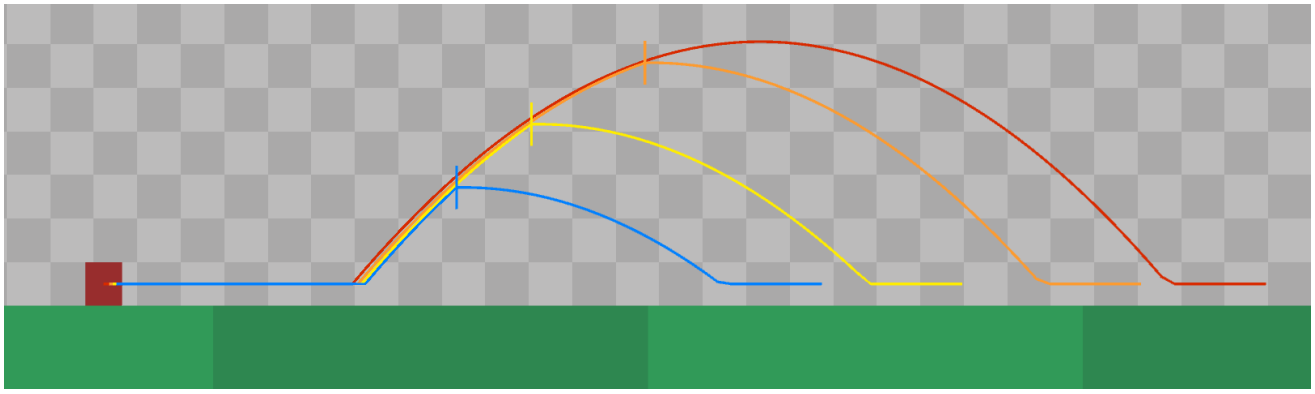


Figure 79: Releasing jump input in Super Meat Boy

Releasing jump input in *Super Meat Boy* sets the character's vertical speed to zero resulting in an instant vertical brake. The horizontal speed is not influenced and the brake only influence positive vertical speed, so releasing jump input while descending from a jump or falling from a platform has no effect. Colliding vertically with the environment results in the same effect as releasing jump input. Figure 79 shows the result of releasing jump at different times. The vertical lines indicate when jump input was released. This brake feels most abrupt when jump is released shortly after being pressed. The character has a higher vertical speed compared to later in the jump and the brake will be most noticeable. Towards the peak of the jump the effect of releasing jump will be barely noticeable.

Because of this instant brake, pressing jump for just one frame would result in Meat Boy moving upwards for just one frame before going falling back down. This movement is so small and brief that it looks and feels more like a twitch than a jump. To avoid this *Super Meat Boy* has a *Minimum Jump Duration*. This represents a brief duration after a jump where releasing jump has no effect. Only after this duration will the release take effect. The result can be seen in figure 80. For both curves jump input was pressed for just one frame. Vertical lines indicates when the game considers the jump input to be released. The dark blue curve shows the one frame jump. Whereas the light blue curve shows the effect of a minimum jump duration at 0.1 seconds, which is the smallest possible jump in *Super Meat Boy*.

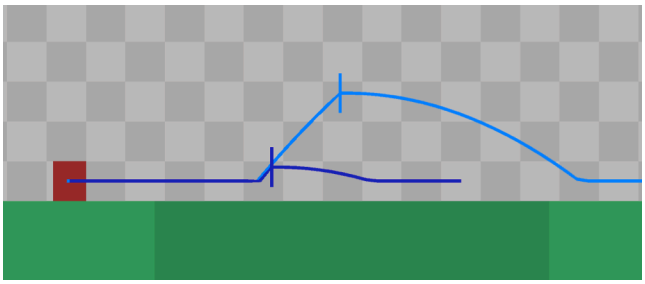


Figure 80: Minimum jump duration in Super Meat Boy

Releasing jump input in *Mario* is closely connected to his *Additive Jump Force*. An upwards vertical force applied during a jump, which partly or completely counteract gravity. Alternatively, this can be seen as a change of gravity.

The effect of the additive jump force in *Mario* can be seen in figure 81. The dark blue curve shows a jump without any additive jump force and the result is caused by takeoff velocity alone. The light blue curve shows the jump with the added force. The upwards force is smaller than gravity, which means the total vertical speed of the character is decreasing during the jump.

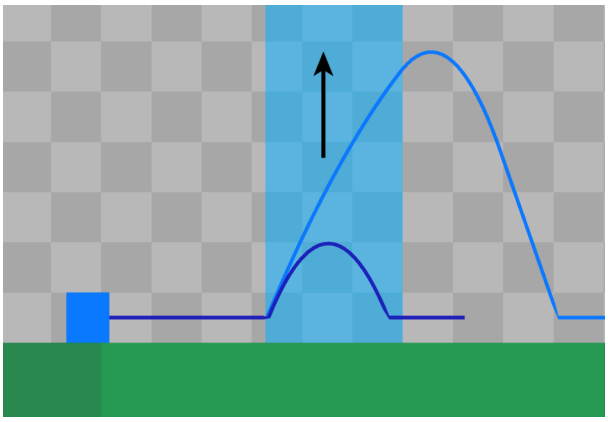


Figure 81: Additive jump force in Mario

In *Mario* the force is added as long as the player is pressing the jump input and has a vertical speed above 8 u/s. The light blue area in figure 81 indicates when the additive jump force is applied.

Other games could have similar or completely different logic for when to apply additive jump force. It could for example be based on a timer instead of speed or the amount of force added could vary during the jump.

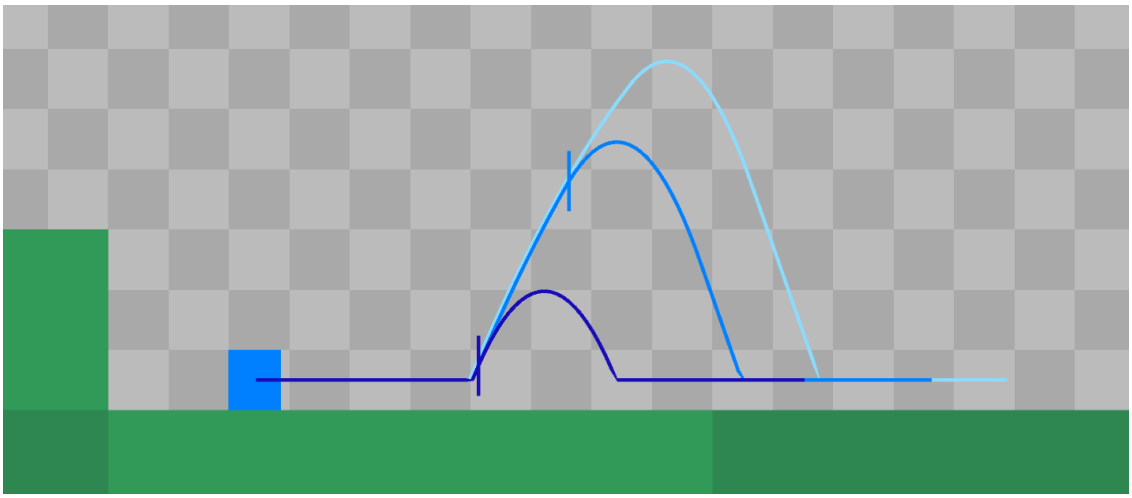


Figure 82: Releasing jump input in Mario

The result of releasing jump input in *Mario* is simply the result of no longer adding additive jump force. The result can be seen in figure 82, where the vertical lines indicate when jump input was released. The light blue curve shows a jump without a release. The blue curve shows a jump with release after 0.25 seconds. Notice that after releasing jump input the character continues to go upwards and has a rounded peak before falling downwards. Very different from the instant brake in *Super Meat Boy*. The dark blue curve shows the smallest possible jump and input was pressed for just one frame. *Mario* has no *Minimum Jump Duration* but also no brake when releasing input. This jump will therefore be the result of just applying *Takeoff Velocity* and letting gravity do the rest. It will not be a twitch, but a small jump with a height of 1.475 units. The jump curve will be similar to the top of the blue curve. With this implementation the size of the takeoff velocity will determine the smallest possible jump.

In *Mario* it is possible to reactivate the additive jump force. This happens if the player release jump input very briefly and press it again while vertical speed is above the threshold. The input must only be briefly released because gravity quickly brings the velocity below the threshold. For example, releasing the jump input after 0.25 seconds leaves only about four frames or 0.067 seconds to repress jump. It is unlikely that the player can react this fast. However, if jump is accidently released for one frame this reactivation could have an effect.

A different implementation is used in *Limbo* where releasing jump input activates a *Release Drag.* This is a downwards force applied when the character is in the air and jump input is not pressed. This force is only applied as long as vertical speed of the character is positive.

Figure 83 shows the result of releasing jump input. The red curve shows a jump without release. The orange curve shows the result of releasing jump input after 0.2 seconds. The yellow curve shows a very brief jump input of only 0.02 seconds, close to one frame. Release drag and gravity gradually stops the jumps and the minimum jump will be rounded and not twitchy. Looking at the yellow curve it seems input has been released before the jump begins. This is because of a short delay between jump input and the actual jump. This delay is 4-5 frames or about 0.08 seconds. Similarly, there is a short delay from jump input is released to release drag is applied again about 0.08 seconds.

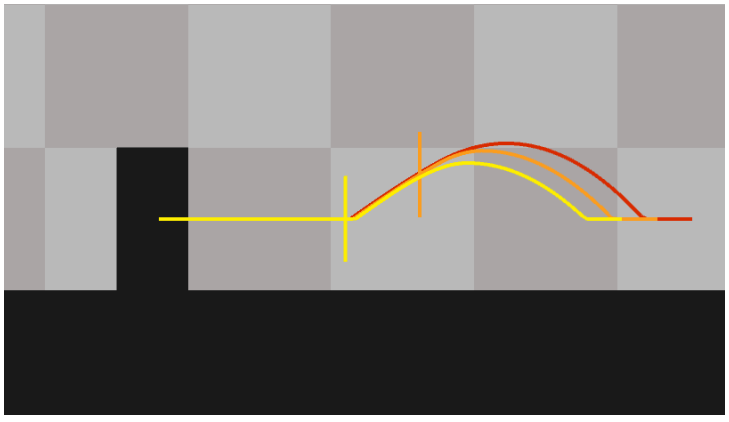


Figure 83: Releasing jump input in Limbo

It is worth mentioning that release drag can be combined with the previously mentioned friction occurring when horizontal input is released. Figure 84 shows the result of releasing both horizontal input and jump input right after takeoff. This means that to achieve the largest possible jump the player will have to hold both jump input and horizontal input during the jump.

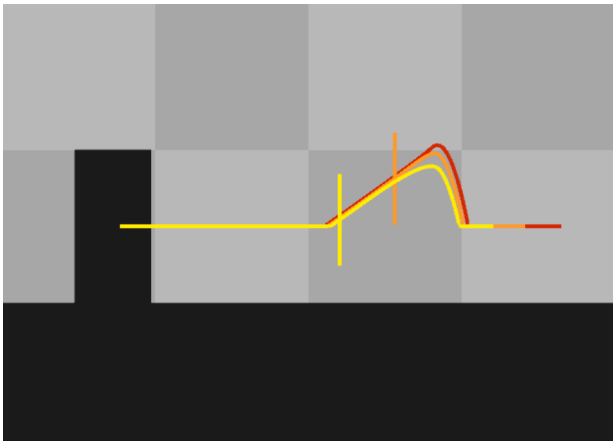


Figure 84: Releasing both horizontal input and jump input in Limbo

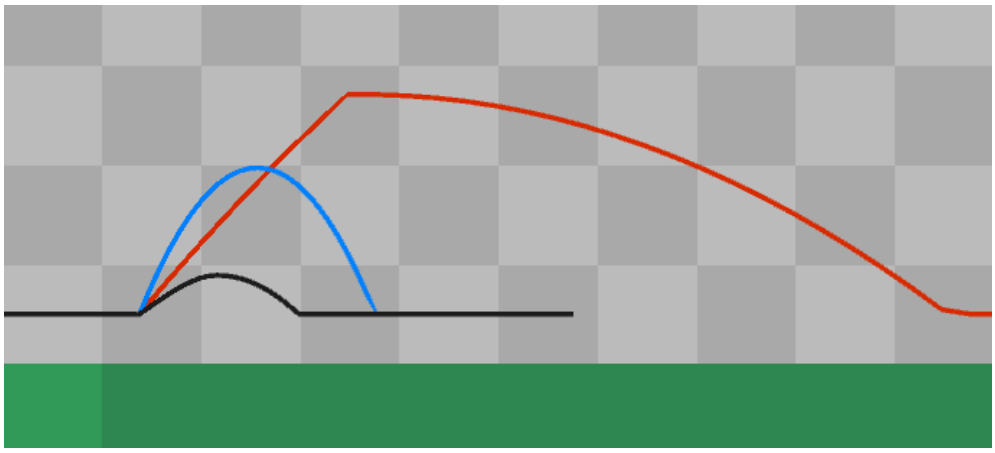


Figure 85: Jump input released after one frame for each of the three case studies.

	Meat Boy	*Mario*	*Limbo*
Instant Break	yes	no	no
Minimum Jump Duration s	0.1	-	-
Additive Jump Force u/s^2	-	60	-
Additive Jump Force, Threshold u/s^2	-	8	-
Release Drag u/s^2	-	-	3.75
Hold Jump Input	no	no	yes
Minimum Jump Height u	2.211	1.475	0.392
Minimum Jump Duration s	0.467	0.4	0.65

Figure 86: Parameters and derived measurements for releasing jump input

Figure 85 presents the minimum jump from the three case studies, which highlights the differences between the implementaitons. Figure 86 is an overview of the parameters presented in this chapter. There are many different ways to handle the release of jump input and this was just three of them. Fineberg (2015), for example, presents a range of simple and practical solutions to jump release. Most of them are similarly to the approaches described here except his last solution. For this solution upwards force is applied at takeoff and during the ascend of the jump. This force is active as long as jump input is pressed, but the amount of force applied is gradually decreased. In a way this solution could be seen as a variation of the additive jump force from *Mario*. This jump was not measured and just mentioned here briefly as an alternative.

Details

This chapter will look at three details found during the project, which did not fit well into the previous chapters. How input is treated in very specific cases is the focus of these details. They might seem subtle at first, but can have a lot of influence on how responsive the game feels.

Jump Cache

Consider the following scenario. The character is falling towards the ground and the player wishes to perform a new jump as soon as possible. The player tries to time a press of the jump input with the character hitting the ground. If this jump input is one frame prior to landing no jump will occur. This is, for example, the case in *Super Meat Boy*. To prevent this, some games have a *Jump Cache*, a duration after pressing jump where the input is still considered valid. If jump input was pressed while in the air and the character lands on the ground within this duration, a jump is performed. A similar detail is described by Pignole (2014) as a tolerance. He describes the jump input being stored and performed later, but in his description the distance to the ground determines the validity of the input instead of a duration.

Jump cache can for example be seen in *Mario*, where jump input is stored for 1-2 frames. This might be intentional or might be caused by the way ground collision is handled. A more obvious example is *Braid*. Here, jump input is stored for as long as 0.23 seconds. Figure 87 shows the earliest time where a jump input is valid. For both *Mario* and *Braid* jump can be released while still in the air and the cache will still take effect when the character lands.

Figure 87: Jump cache in Braid

Checking for a jump cache in *Limbo* revealed a different detail. After performing a jump, the jump input normally has to be released and repressed in order to perform a second jump. However, holding down jump input in *Limbo* results in consecutive jumps.

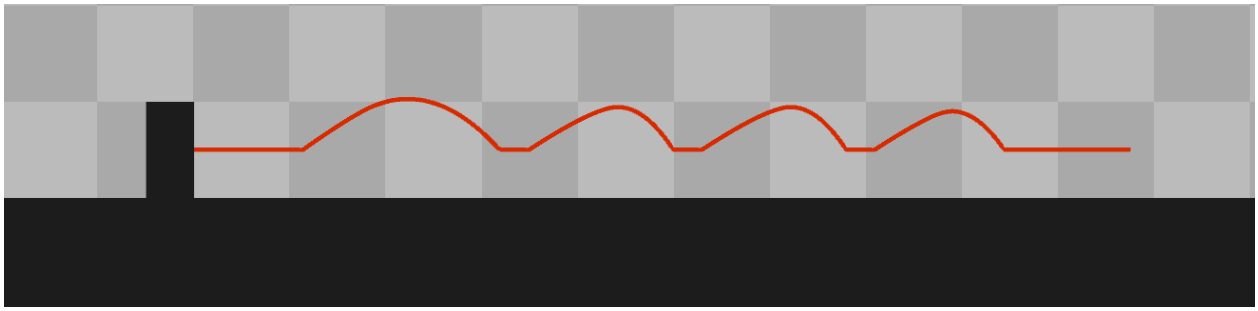


Figure 88: Holding jump input in Limbo

As such holding right input and jump input will result in a forward jump, followed by a new jump every time the character lands. The resulting curve can be seen in figure 88. There is a delay of about 0.05 seconds after landing before a new jump is possible. This movement also result in the previously mentioned short jump, because the distance traversed between jumps is not large enough to allow the long jumps. This can be seen as the first jump is slightly larger than the following. From this we can determine, that the fastest way to move forward in *Limbo* is to jump as much as possible. For each jump the character gains speed from the horizontal takeoff velocity. While in the air the character moves faster than on the ground because of the increased maximum air speed. For this approach to work the player should also make sure to run 0.5 units between jumps to avoid short jumps.

Edge Jump

Running towards an edge and performing a jump as close to that edge as possible can be tricky. If the player jumps just one frame too late no jump will be performed and the character will fall of the platform, potentially to a terrible fate. To avoid this some games allow the character to perform an *Edge Jump* for a short duration after leaving the platform. This detail was mentioned by Venturelli who calls it a Ghost Jump (Venturelli, 2014) as well as Pignole who describes it as a time delay (Pignole, 2014).

Edge jumping is, for example, used in *Limbo* where a jump can be performed up to 0.08 seconds after leaving a ledge. Figure 89 shows the latest frame where jumping is still possible. Because of the delay between input and action, the jump will actually be performed about 4 frames later than this. This means the boy would be slightly further away from the platform when the jump is actually performed.

Figure 89: Because of edge jumping, the character can still jump briefly after running of the platform.

Upwards Jump

Another detail from *Limbo* is the *Upwards Jump*. It occurs when jumping while standing still and not giving any horizontal input. It works like a regular jump, but in the implementation it is handled as a different state. The takeoff velocity of this jump adds no horizontal speed and the vertical speed added is higher than a jump when running at full speed. While the boy is going upwards and has a positive vertical velocity a downwards force is added, there is no similar force added to the regular jump. The release drag of the upwards jump is applied when the boy has a positive vertical velocity and jump input is released. This is the same as the regular jump, however the size of the drag is larger for the upwards jump. Horizontal input has no effect during an upwards jump and as such there is no air control. All this result in a jump with a slightly increased jump height. The upwards jump allows the designer to adjust this case separately from the regular jump.

If horizontal input is given within 4 frames after an upwards jump has started it will transition into a regular jump adding horizontal speed and acting like a regular jump from then onwards. This is expected to handle cases where the player attempts to give horizontal input and jump input simultaneously, but horizontal input is a few frames later than the jump input. In this case the transition allows a forward jump, instead of an upwards jump seemingly ignoring the horizontal input.

	Upwards Jump	**Regular Jump**
Vertical Takeoff Velocity u/s	2.95	2.4
Horizontal Takeoff Velocity u/s	0	3.5
Vertical Air Friction u/s²	1.45	0
Release Drag u/s	5.95	3.75

Figure 90, Comparing the forces of the upwards jump and the regular jump

Summarize and Reflect

This chapter will look at the three case studies more holistically. Each section will start with a brief summarization of the movement in the game, followed by an analysis of the movement in context with the general design of the game. As part of this analysis the affordance of each jump will be discussed. Affordance in this context refers to the types of movement encouraged or emphasized by a jump.

Super Meat Boy

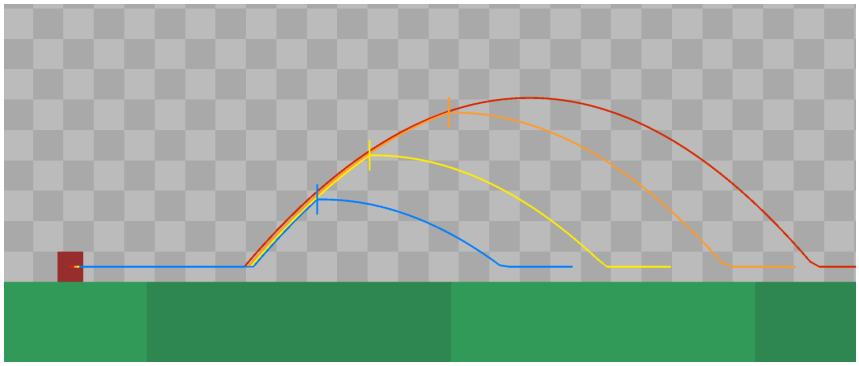


Figure 91: Jumping in Super Meat Boy. The vertical lines indicate when jump input was released. The blue jump curve is the smallest possible jump and the red jump curve the largest.

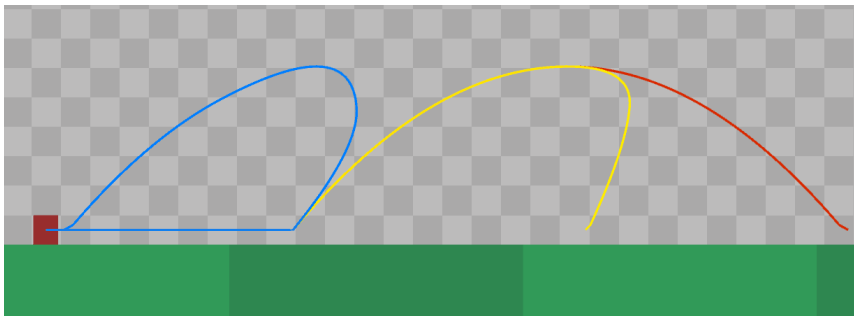


Figure 92: Air control allows Meat Boy to land anywhere between the blue and the red jump curve.

Ground Movement	
Max Ground Speed u/s	18
Acceleration u/s^2	35
Deceleration u/s^2	-
Turn Acceleration u/s^2	70
Jump Takeoff	
Gravity u/s^2	41
Terminal Velocity u/s	-
Takeoff Velocity Vertical u/s	21
Takeoff Velocity Horizontal u/s	-
Air Control	
Air Acceleration u/s^2	35
Horizontal Air Friction u/s^2	-
Air Turn Acceleration u/s^2	70
Maximum Horizontal Air Speed u/s	18
Minimum Horizontal Air Speed u/s	-
Jump Release	
Instant Break	yes
Minimum Jump Duration s	0.1
Additive Jump Force u/s^2	-
Additive Jump Force, Threshold u/s^2	-
Release Drag u/s^2	-
Hold Jump Input	no

Figure 93: Summary of the properties from Super Meat Boy

The horizontal input in *Super Meat Boy* is treated as binary and all input above the deadzone is treated equally. This is the case both on the ground and in the air. On the ground Meat Boy accelerates gradually, and when horizontal input is released he stops instantly. He applies a turn acceleration when changing direction, which sometimes conflict with the instant brake as previously discussed. When jumping Meat Boy applies a vertical takeoff velocity which is independent from run speed and distance traversed. In the air Meat Boy is pulled downwards by gravity and has no terminal velocity.

In the air Meat Boy has the same maximum speed, acceleration, and turn acceleration as on the ground. Releasing horizontal input while in the air prevents further acceleration, but no friction or drag is applied. This prevents the conflict between turn acceleration and instant brake, which was present on the ground. If jump input is released during a jump any positive vertical speed is set to zero, while horizontal speed is unaffected. This gives an instant vertical brake similar to the horizontal brake occurring when running on the ground.

Super Meat Boy is based on the concept of speed. The character moves quickly and each level is designed to encourage speed. Coming in contact with any danger kills the character without mercy. There is barely any load screen, instead the character immediately respawns at the start of the level. This allows quick consecutive attempts which tend to blur together. Many attempts are often required to beat a level, particularly the later levels in the game. The movement of Meat Boy is carefully designed to support this focus on speed.

Super Meat Boy offers simple, consistent, and responsive movement. For example, the symmetry between horizontal movement on the ground and in the air. They are identical apart from the instant brake on the ground. This gives a smooth transition between the two states, which supports a flowing motion even when rapidly changing from ground to air.

The takeoff velocity is consistent across any conditions and does not require preparations. This again supports flowing motions as the player can rely on the jump and perform it without hesitation. The levels in *Super Meat Boy* often encourage this flowing motion as seen in figure 94. Here Meat Boy jumps from a wall, avoids a saw blade, runs to a new wall, and continues upwards. This level was played as one series of actions without taking breaks on the way. In many levels moving continuously is even a requirement to succeed. This can be due to platforms crumbling under the feet of the character or lava gradually rising, keeping players on their toes.

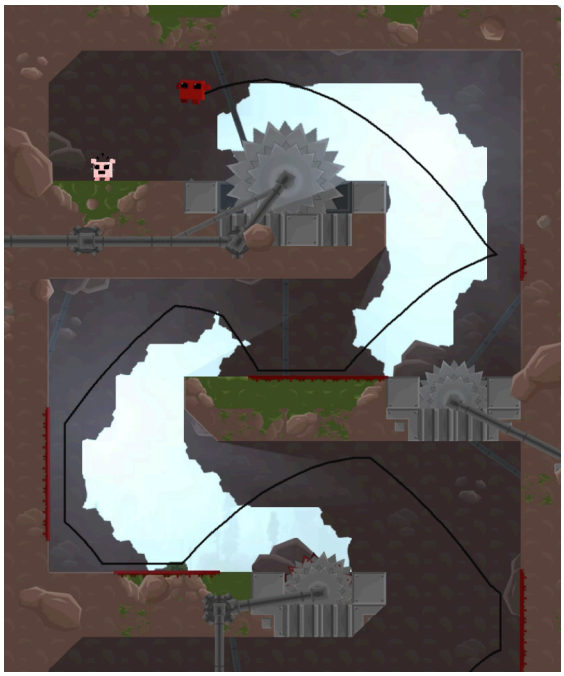


Figure 94: Moving through a level in one continuous flow of movement

The controls in *Super Meat Boy* are very responsive. This can, for example, be seen with the instant brakes, both horizontally and vertically. They allow the player to react quickly and with high precision, which is often needed to complete a level. The instant brake can feel abrupt, especially with high speed, for example, when doing tiny jumps or coming to a halt while running fast. This arguably removes some of the feeling of the game as a physically simulated space. This can, for example, be seen if a player runs at full speed, releases horizontal input for one frame and jumps. In this case the character will stop instantly and jump straight upwards. The movement of Meat Boy will in this case be doing a 90 degree turn. This highlight how simulated space has a lower priority than responsiveness.

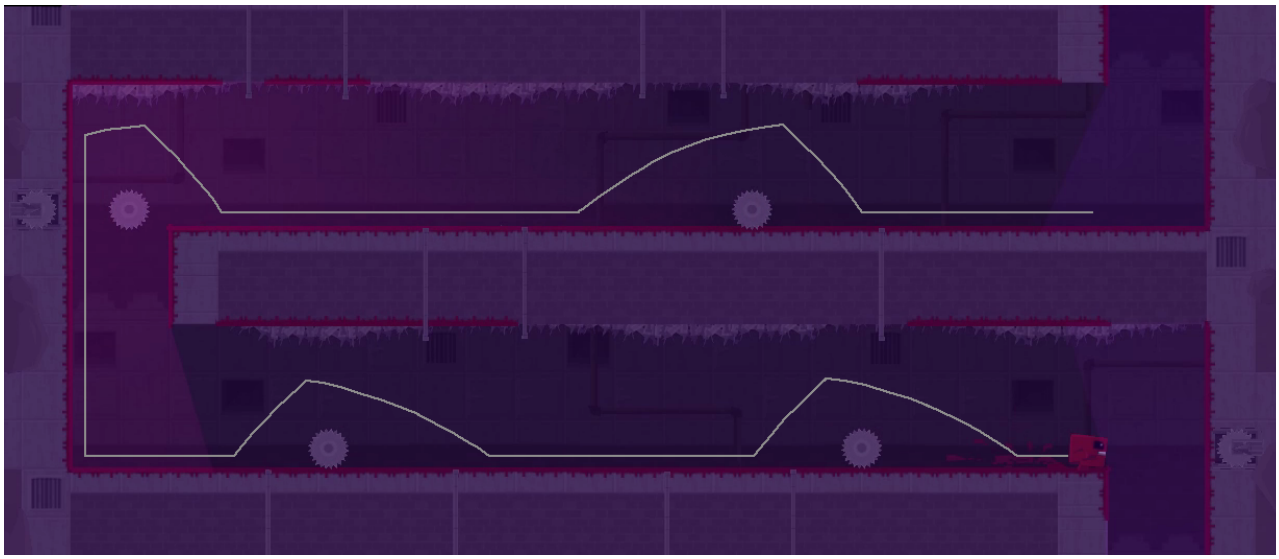


Figure 95: Avoiding saw blades in Super Meat Boy. The saw blades have been edited to show their position when the character dodged them.

Responsive controls combined with fast horizontal movement results in a jump which affords avoidance. This can, for example, be seen in figure 95, where the player must navigate corridors filled with moving saw blades which are fired repeatedly towards Meat Boy. As if this was not enough the ceiling is covered in broken glass. The vertical brake is essential for moving through this level and highlights the avoidance afforded by the jump. The fast horizontal speed also plays a role and gives a larger span of success by helping the character get away from danger as soon as the required height is gained.

The player can move the character with a lot of freedom. Moving fast horizontally and having a high takeoff velocity allows leaps across giant gaps. As such, Meat Boy can move much farther horizontally than vertically. One way to improve vertical movement would be to increase takeoff velocity, however this would also drastically change horizontal movement, which is already more than sufficient. Instead, Meat Boy makes good use of the environment. When hitting a wall Meat Boy slides upwards and can jump from the wall. There is no limit to how many times Meat Boy can do a wall jump. With a little air maneuvering the player can stay in one place vertically by continuously jumping from the wall. This gives the player a lot of freedom to move vertically. Because of this, the

level design is less restricted and the game contains levels with both vertical and horizontal layouts. Moving freely again supports the high speed of the game and facilitates a flowing motion through both horizontal and vertical levels.

Mario

The input in *Mario* is binary, a technical limitation of the hardware it was released on. On the ground, Mario accelerates when running and decelerates when horizontal input is released. The rates are the same, therefore it takes the same amount of time to decelerate as it took to accelerate. When Mario changes direction on the ground, a turn acceleration is applied. This acceleration is 14.3% larger than acceleration and deceleration combined and controls the iconic skid when the character changes direction. While in the air Mario has the same acceleration and turn acceleration as on the ground. In the air there is no friction and the character does not decelerate when horizontal input is released.

When Mario jumps, a vertical takeoff velocity is applied. The size of this velocity depends on how fast he is running. There is a total of four possible takeoff velocities and horizontal speed determines which one is used. Without sprinting only the two smallest velocities are obtainable. While in the air a high gravity is applied. However, it is quickly limited by a terminal velocity. This prevents further downwards acceleration after falling for just 0.230 seconds. This is fast enough to occur during common jumps.

When in the air a consistent upwards force is applied, partly counteracting gravity. For this force to be applied the jump input must be pressed and vertical speed must be above a threshold. This is also how the player can control the height of the jump. As soon as jump input is released the upwards force is no longer applied. Without the force gravity will quickly turn things around and the character starts descending.

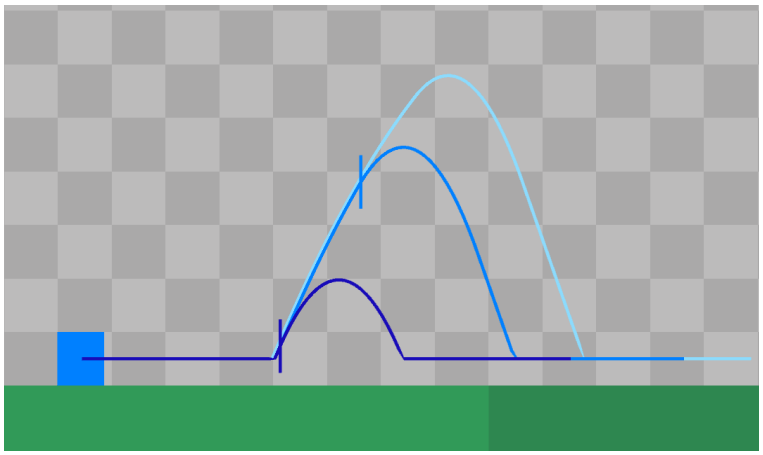


Figure 96: Jumping in Mario. The vertical lines indicate when jump input was released. The dark blue jump curve is the smallest possible jump and the light blue jump curve the highest.

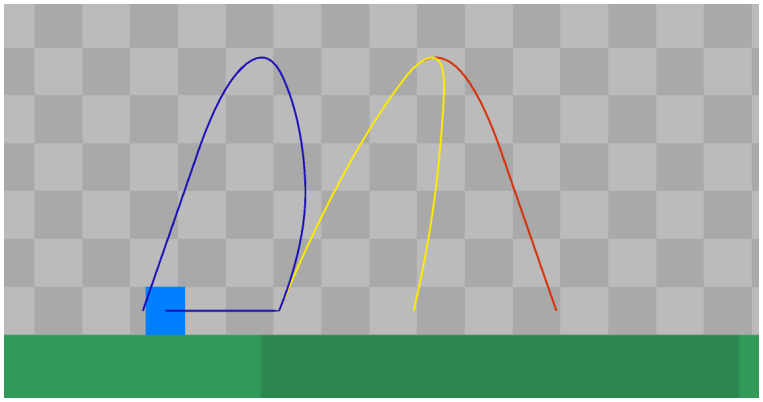


Figure 97: Air control allows Mario to land anywhere between the blue and the red jump curve.

Ground Movement	
Max Ground Speed u/s	6
Acceleration u/s^2	13.125
Deceleration u/s^2	13.125
Turn Acceleration u/s^2	30
Jump Takeoff	
Gravity u/s^2	75
Terminal Velocity u/s	17.25
Takeoff Velocity Vertical u/s	13.75 - 15.75
Takeoff Velocity Horizontal u/s	-
Air Control	
Air Acceleration u/s^2	13.125
Horizontal Air Friction u/s^2	-
Air Turn Acceleration u/s^2	30
Maximum Horizontal Air Speed u/s	6
Minimum Horizontal Air Speed u/s	-
Jump Release	
Instant Break	no
Minimum Jump Duration s	-
Additive Jump Force u/s^2	60
Additive Jump Force, Threshold u/s^2	8
Release Drag u/s^2	-
Hold Jump Input	no

Figure 98: Summary of the properties from Mario

How the game conveys the weight and the speed of the character is often praised. When released the game was among the first platform games to accomplish this feeling of physical simulation. Looking at the data we can see some of the aspects that supports the game as a simulated space. The character, for example, gradually decelerates to a halt. This conveys the weight of the character and simulates how speed is handled within the game world. The turn acceleration in *Mario* lets him skid for 0.183 seconds.

The movement is again gradual and not instant, conveying the weight, speed, and friction of the simulated space. While in the air, the character similarly do not react instantly and needs to accelerate or counteract existing speed.

Having takeoff velocity increase with run speed conveys an important part of the simulated space. It presents a relationship between run speed and the general size of the jump. This will not only convey the simulated space of the game, but also influence how the player controls the character. The player should learn to anticipate and prepare for a jump. This creates tension when running towards an edge as the player attempts to reach enough speed to perform a sufficiently large jump. A similar case occurs when jump input is released. Here the character continues upwards and smoothly rounds off the jump curve. To control this effectively the player should not only rely on reacting quickly. Instead, the current vertical speed should be taken into consideration and jump input released in anticipation of danger. It is unlikely that the player will consciously deliberate any of this. However, the player should develop an intuitive understanding of the simulated space of the game. This focus on controlling the speed of the character and the forces applied direct the attention of the player and emphasize the game as a simulated space.

This much attention on weight and physicality take away some responsiveness. The input is treated immediately, but it takes significant time to accelerate or overcome existing speed, this can make Mario feel heavy or make the controls feel floaty. This is primarily a frustration when very fast or precise movement is required. With time the player is expected to get used to the feeling of weight and adjust input accordingly. Next, a few examples of level design from *Mario* will be presented and how they support physical simulation and responsiveness will be discussed.

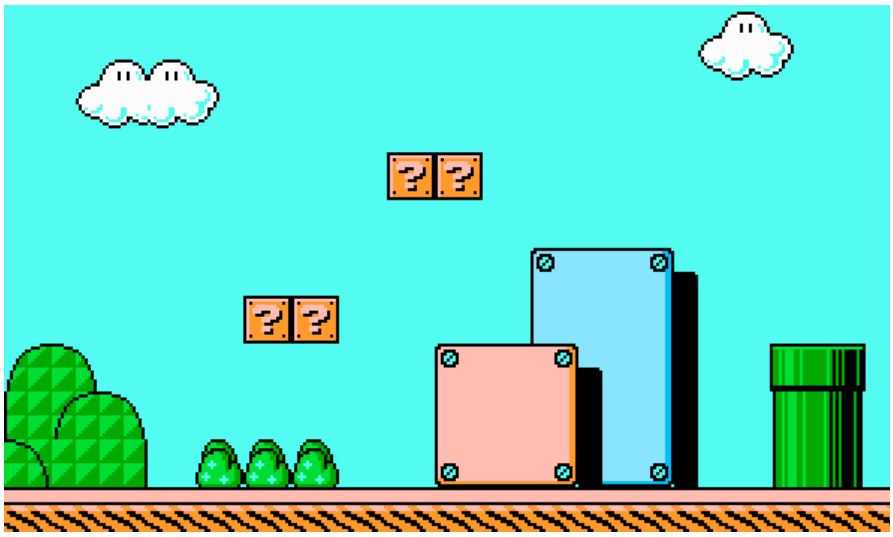


Figure 99: first set of platforms in Mario

Figure 99 shows the first set of platforms in *Mario*. Imagine that the player wishes to jump and land on the first set of question blocks by performing a running jump. Horizontal input is released during the jump or shortly after landing. In both cases, Mario continues through the air and only after landing does horizontal deceleration begin. When decelerating he slides 1.226 units across 0.450 seconds before standing still. The width of one question block is 1.067 units. This means that landing on a single question block with a running jump will always result in Mario sliding off the block. In figure 99 we see that the two first platforms encountered consist of two question blocks. This gives the platforms a total width of 2.133 units, which is enough for Mario to slide to a halt without falling. Similarly, the pipes often used as platforms have the same width as two question blocks. Throughout the game platforms with this width are common. Figure 100 shows two examples where the platform width is used repeatedly.

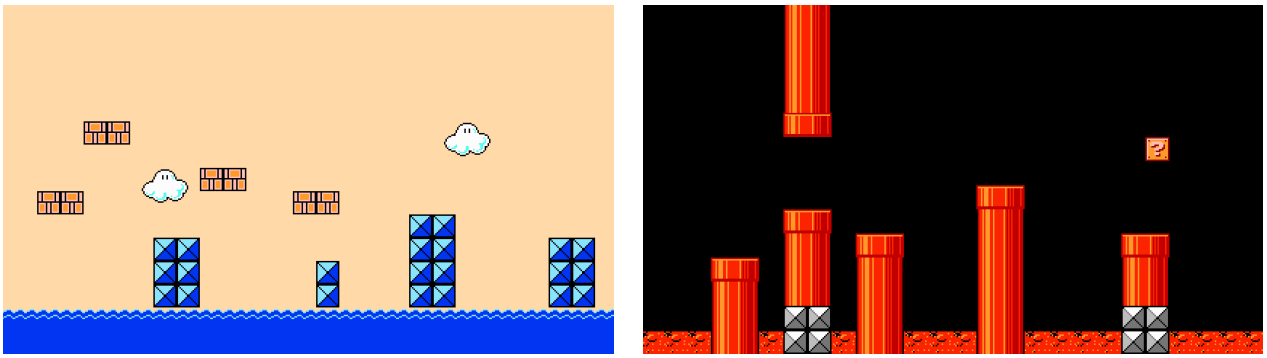


Figure 100: Platforms two tiles wide are large enough to allow a sliding deceleration after a landing

Power-ups can often be found in single question blocks. However, in this case the Power-up is essential, not so much staying on the block. A platform underneath is also guaranteed, since the player has to hit the block from below to reveal the power-up. This mitigates risk and jumping for the power-up and sliding off will likely not have serious consequences. As an alternative to decelerating on platforms, the player can keep horizontal input and do a new jump continuing through the level. The movement in *Mario* could be said to encourage this with gradual acceleration and takeoff velocity dependent on run speed. However, It is not required of the players early in the game.

Only later in the game is the player is expected to have precise control of Mario and be capable of performing very precise landings. For example, changing input direction to counteract the speed of the character, either in the air or right after landing. This requires practice and the first couple of jumps like this do not have harsh consequences if the player fails.

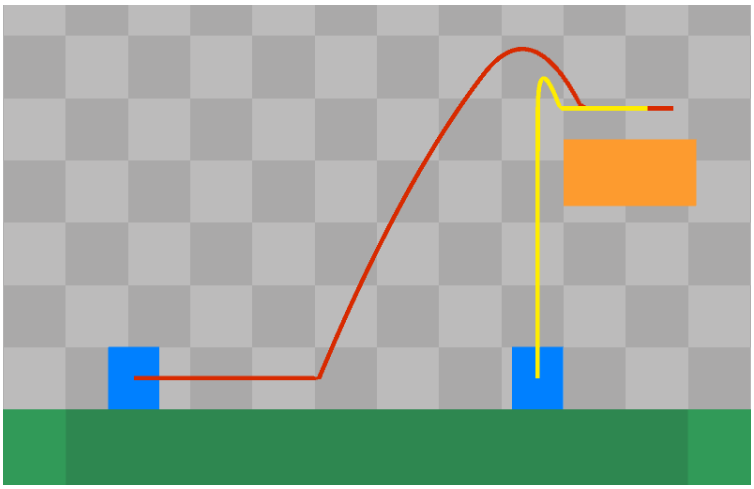


Figure 101: Different ways to reach a platform

A different way to reach a platform is a jump upwards while standing still beside the platform as seen in figure 101. While in the air, horizontal input is given to move the character onto the platform. Again, we see how the level design supports the movement. The jump height when standing still is 4.817 units and the height of the platform is 4.333 unit. This mean the jump is just high enough for the upwards jump to work. If the player were to take a small run up and it was longer than 0.305 seconds, Mario will get a slightly higher takeoff velocity. The result is a jump height of 5.287 units, which is 0.47 units higher than a jump when standing still. This gives just a little room for errors and helps the player reach the platform. When sprinting this effect will be even more noticeable.

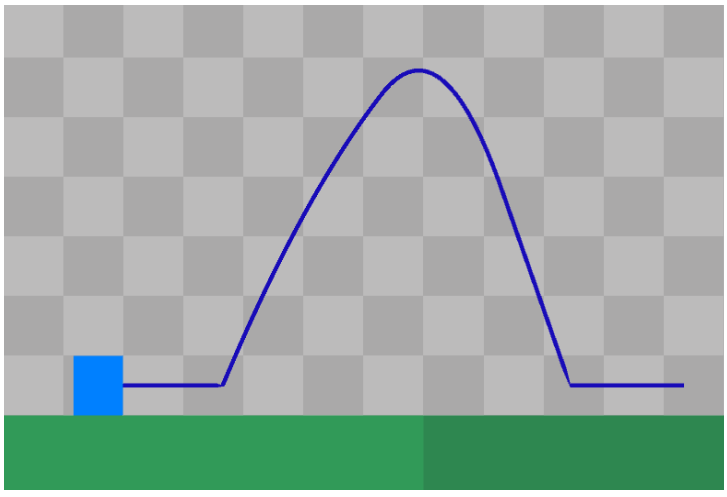


Figure 102: Jump curve from a running jump in Mario

Looking at the shape of the jump curve in figure 102, we can see the effect of different parameters. This shape is different compared to the round and symmetrical jump curve of Meat Boy. This curve has a pointy top and slopes to both sides. The downwards slope is caused by the terminal velocity, which controls both how steep the descent will be and when acceleration will stop. The upwards slope is caused by the additive jump force and the threshold for applying this force. Adjusting these parameters controls steepness of the slope as well as how high the jump will be. Gravity influences most aspects of the jump, but especially the pointy top of the curve. The top is caused by gravity acting alone, without additive jump force or terminal velocity. In this respect, gravity controls how pointy the curve will be. If jump is pressed for just one frame the minimum jump will occur. A tiny jump which resembles the top of a regular jump. Since, jump input is immediately released no additive jump force will be added. Furthermore, the jump is too brief for terminal velocity to take effect. As such the minimum jump is controlled only by takeoff velocity and gravity.

The jump in *Mario* affords stomping, which is used repeatedly in the game to land vertically on enemies. One important factor is the relatively low horizontal speed. This narrows the amount of possible landing positions, making it easier to hit a target. However, the entire shape of the curve is important. The terminal velocity gives a slightly slower descend, allowing the player to adjust the landing. The jump gradually rounds off when jump input is released, which also adds time to correct the jump. The jump curve is also asymmetrical and leans forward ever so slightly. This is because ascend and descend are handled differently and the way the parameters have been adjusted. It takes 0.533 seconds to reach the peak of the jump and 0.417 seconds to go back down. This slightly increased downwards pace supports stomping as well. This stomping motion also affects landing vertically on platforms as previously discussed. If the jump had a different shape the width and height of the platforms would likely have to be adjusted accordingly. As such, both the interaction of stomping enemies as well as the level design, work in correlation with how the jump is implemented and adjusted.

As mentioned previously, the parameters used for *Mario* during this project were not measured, but found in the guide *Super Mario Bros. 3 Physics* (Aldrich, 2012). All parameters in the guide were presented as four digit hexadecimal numbers. From left to right the four numbers represent blocks, pixels, sub-pixels, and sub-sub-pixels. Since the numbers are hexadecimals, one block corresponds to 16 pixels, each pixel corresponds to 16 sub-pixels and so forth. For ease of use, the numbers were converted into units for this project. The fact that pixels are divided further internally is worth considering. It implies that even though *Mario* is only visualised in a resolution of 256 x 240 pixels, his internal position and speed works with smaller intervals. This can sometimes be experienced in the game when Mario decelerates. With a slow speed Mario can stand still for a few frames and then suddenly move an extra pixel. This extra movement is likely the result of sub-pixels and sub-sub-pixels being added to the internal position. At some point, the position is rounded up to the next whole pixel and the character is moved. This detail highlights an interesting dynamic between an internal model with more detail than the game can represent visually.

Limbo

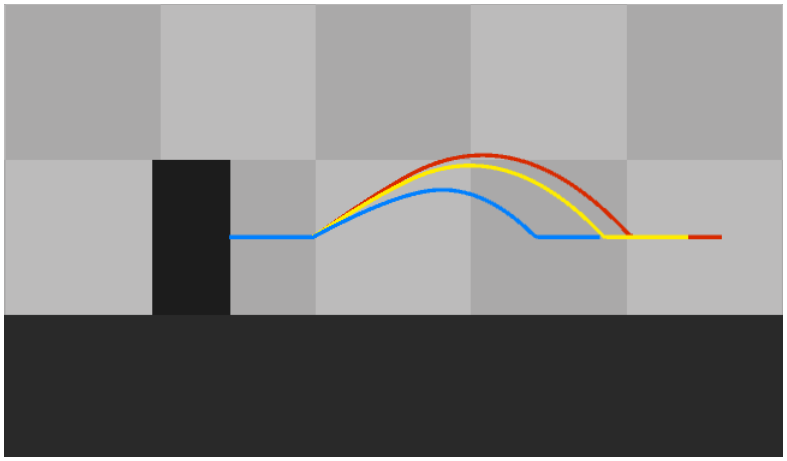


Figure 103: Run speed determines takeoff velocity. The blue jump curve shows the smallest possible jump and the red jump curve the highest.

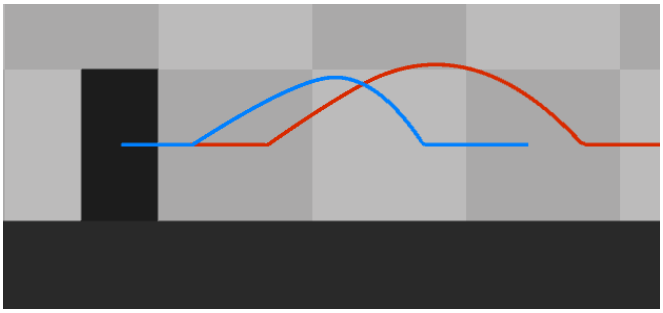


Figure 104: Long and short jump based on traversed distance

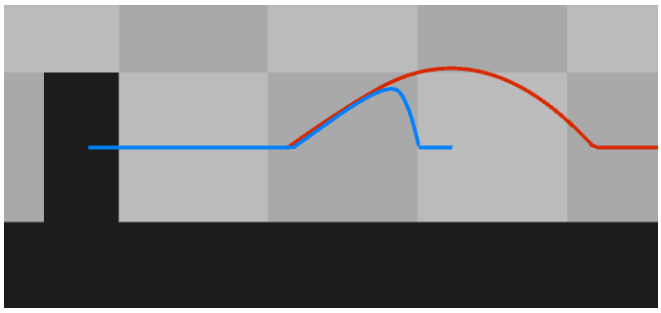


Figure 105: Releasing jump input and horizontal input allows the boy to land anywhere between the blue and the red jump curve.

Ground Movement	
Max Ground Speed u/s	0.226 - 2.1
Acceleration u/s^2	-
Deceleration u/s^2	-
Turn Acceleration u/s^2	-
Jump Takeoff	
Gravity u/s^2	6
Terminal Velocity u/s	-
Takeoff Velocity Vertical u/s	1.78 - 2.47
Takeoff Velocity Horizontal u/s	3.5
Air Control	
Air Acceleration u/s^2	-
Horizontal Air Friction u/s^2	6
Air Turn Acceleration u/s^2	-
Maximum Horizontal Air Speed u/s	-
Minimum Horizontal Air Speed u/s	1.5 - 2.284
Jump Release	
Instant Break	no
Minimum Jump Duration s	-
Additive Jump Force u/s^2	-
Additive Jump Force, Threshold u/s^2	-
Release Drag u/s^2	3.75
Hold Jump Input	yes

Figure 106: Summary of the properties from Limbo

The horizontal input in *Limbo* is treated as analogue both on the ground and in the air. The input also goes through a number of steps before influencing movement as described in Appendix 1. On the ground the Boy has an analogue run speed controlled by the amount of horizontal input. He accelerates, decelerates and turns very fast. The speed of these transitions do not appear to be linked to the amount of horizontal input given. In that respect, they act more like animation transitions and less like physical simulations.

The takeoff velocity in *Limbo* has both a vertical and a horizontal component. The horizontal component is always the same, whereas the vertical component changes based on run speed and traversed distance. Running faster at takeoff results in a higher jump, since acceleration happens very quickly this is most noticeable when feathering the stick. Jumping before having traversed a certain distance results in a short jump, and jumping after the distance results in a long jump.

The boy has a higher horizontal maximum speed in the air than on the ground. Without it the horizontal component of the takeoff velocity would not have any effect. When jumping the horizontal speed immediately accelerates to the maximum air speed. While in the air the boy decelerates, but never below his minimum air speed. The size of the minimum air speed is determined by amount of horizontal input during the jump.

If horizontal input is released during a jump, a horizontal drag is applied. Changing the direction of the horizontal input during a jump gives the same result. This means it is not possible to change direction while in the air. Because of the horizontal takeoff velocity the character starts with the maximum air speed and decelerates. Holding horizontal input does therefore not cause any acceleration in the air, but instead prevents a higher deceleration. As soon as the increased deceleration is active it will be applied for the rest of the jump, and giving a new horizontal input has no effect. If the boy is going upwards in the air and jump input is not pressed a vertical release drag is applied. This allows the

player to adjust the height of the jump. As such horizontal input can be used to apply horizontal drag, whereas jump input can be used to apply vertical drag.

Jumping without horizontal speed or horizontal input results in an upwards jump. This upwards jump has its own takeoff velocity and drag. Early in the upwards jump it is possible to give horizontal input and transition into a regular jump. With the upwards jump the character jumps slightly higher than with the regular jump. *Limbo* has a lot of small details, for example, the edge jump which allows the character to jump after running of a ledge, even though the character is technically in the air. Another example is continuously holding down the jump input, which results in the character jumping repeatedly, this was not the case for other games measured.

The jump in *Limbo* is dynamic and flexible. This comes from the analogue way input is treated as well as the variety of factors that influence takeoff velocity and the deceleration during a jump. This all creates an organic jump, which adapts to the circumstances and the input given. This jump conveys weight and speed of the character and clearly communicates the game world as a simulated space.

Figure 107: A run up, ledge grab, horizontal input, and jump input are all required to jump across this gap.

However, a highly dynamic jump like this can also be challenging for players to control. For example, if the player does not notice the difference between the short and the long jump, she could potentially keep doing the same jump without a run up and fail repeatedly. This is partly solved in the level design by allowing the player to traverse gaps with a short jump at the edge of the platform.

A different challenge comes from the way input is treated during a jump. The longest jump requires the player to keep both jump input and horizontal input pressed during the entirety of the jump. This can be challenging for the players as they might release one of the two inputs without noticing the subtle difference in the jump. The developers even received supports email stating that certain jumps in the game was impossible, likely caused by this need to hold two different types of input simultaneously. Figure 107 shows an example of such a jump. To make it across the gap the player must do a run up and hold horizontal input and jump input pressed during the jump.

Limbo has the most realistic jump compared to the other games measured. More precisely the properties of the jump are somewhat similar to what we observe in the real world. As an experiment, let us imagine that the boy is 1.5 meters tall and try to convert some of his measurements to meters. His run speed would then be 3.15 m/s which is a little too fast for a jog, but slow compared to Usain Bolt, who ran faster than 12 m/s during the 2012 olympics (Fordyce, 2015). The boy's highest jump would be 0.914 meter. A high jump for most people, but not impossible. The guinness world record for highest standing jump is 1.52 meter (Guinness World Records, 2012). The gravity in *Limbo* would be 9 m/s^2 quite close to about 9.8 m/s^2 on earth. In comparison the gravity in *Super Meat Boy* is 6.83 times higher than *Limbo*, while the gravity in *Mario* is even higher. Realisme can also be seen in the controls of the character especially when in the air. Most platform games allow the character to accelerate or change direction while in the air or even curve a jump around obstacles. None of this is possible in the real world or in *Limbo*, where you can only control how much you decelerate.

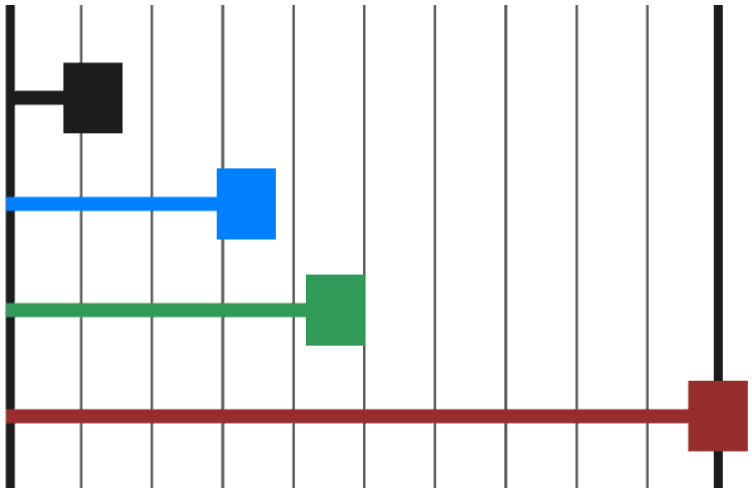


Figure 108: Estimated race between Usain Bolt and the game characters. The game characters are not allowed to use their sprint mode. From top to bottom: Limbo, Mario, Usain, Meat Boy.

The movement and jumping in *Limbo* is slow paced compared to the other games measured. This supports the melancholic mood of the game and give the player time to reflect and think about the puzzles. The pace can also be calming and establish a momentary feeling of security. This will increase the surprise when the character is viscerally torn apart by a hidden trap. As mentioned *Limbo* has a dynamic jump consisting of a combination of physical forces not entirely disproportionate to reality. This strongly establish the physical simulation of the jump and conveys the simulated space of the game world. Responsiveness is also present at least in parts of the jump. Because the run speed is relatively slow the character can change speed rapidly without feeling abrupt. This allows fast acceleration, deceleration and direction change which makes the ground controls feel responsive. There is limited responsiveness in the air, but this is handled in the level design. As such, changing direction or landing on tiny platforms is never required. Allowing the player to perform an edge jump also further supports responsiveness.

The movement in *Limbo* seems like a good balance between simulated space and responsiveness. It fits the game well, but would likely not scale well to the faster pace seen in *Mario* or *Super Meat Boy*. The dynamic jump would likely feel too unreliable at a faster pace, where consistency is usually needed. Having limited air control also feels very impeding when moving faster and many of the levels in, for example, *Super Meat Boy* would not be possible without a lot of air control.

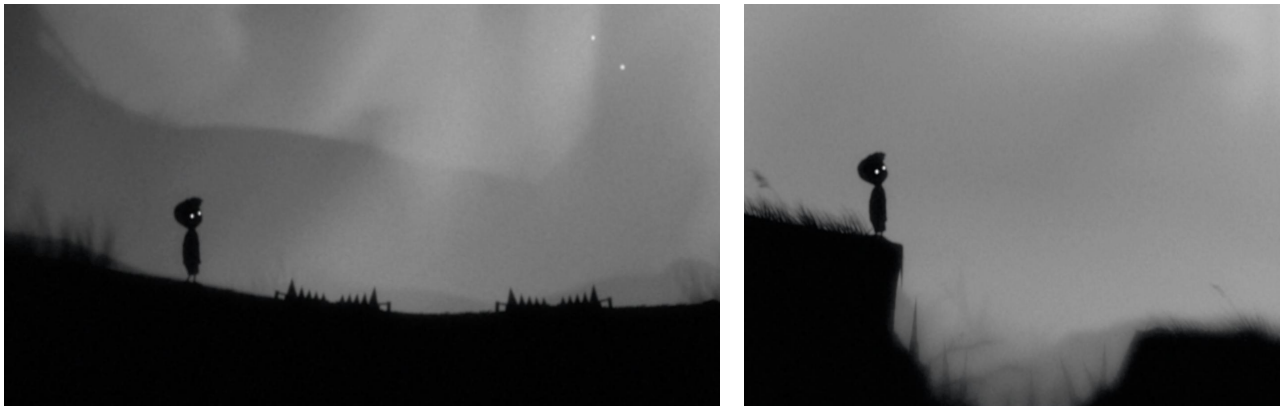


Figure 109: Jumping forward to avoid danger

The jump in *Limbo* affords horizontal jumps across gaps and traps. The takeoff velocity has a high horizontal component, which means the jump is directed forward. The boy is also accelerated forwards, which is most noticeable when moving slowly just before jumping. Despite his limited jump height and run speed, this acceleration allows the boy to do quite long jumps. The influence of this can be seen throughout the level design of the game. Most obstacles are traversed by jumping forwards and across them and it is uncommon to jump upwards to avoid danger. Figure 109 shows two examples of obstacles traversed horizontally.

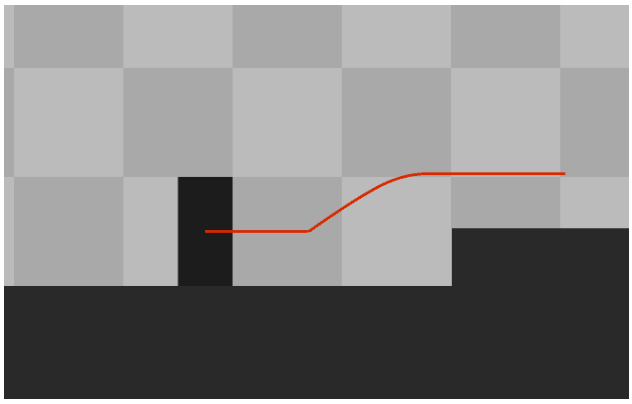


Figure 110: Highest platform reachable with jump alone

Figure 111: Example of platform reachable with jumping and a ledge grab

The boy rarely if ever jumps from beneath a platform and lands on top of it as seen in figure 110. Since jump height is small, this platform is the highest platform reachable with a jump. Because of the many variations in the jump, the player might even struggle to make it onto this platform. Not being capable of doing a jump which looks trivial feels frustrating and very limiting. To avoid this the game allows the boy to grab hold of ledges and climb up. If he is close to making the jump, he will just briefly touch the ledge while stepping onto the platform. If he is further away he can stretch his arms and hang from the platform and climb up afterwards. This can happen both when jumping towards a ledge or standing underneath it and doing an upwards jump. It significantly increases how high platforms can be reached as can be seen in the example in figure 111. It also adds horizontal reach, allowing the character to grab ledges in front of him instead of falling down. Ropes and ladders are used when further vertical movement is required. Platforms smaller than the height of the character are handled a little different. If the character stands next to such a platform and the player press jump input, the character will not perform a jump. Instead, the boy grabs the ledge in front of him and immediately pulls himself up. This makes sense as there is not enough space for him to hang and doing a quick climb up feels natural. Other games with a similarly limited jump height had different solutions to this, which will be discussed further later in this chapter.

General Reflection

Considering the analysis of the three case studies together a few overall reflections can be made. For all three case studies a balance between responsiveness and simulated space has been discussed. *Super Meat Boy* was very responsive, but had less focus on simulated space. *Mario* had a strong focus on conveying a feeling of weight, while the character's reaction was less immediate. *Limbo* had a pace slow enough to allow a balance. Acceleration, deceleration and direction change happens almost instantly, while the game maintains a strong sense of being physically simulated. This tradeoff between real-time control and spatial simulation makes sense in the context of movement. If the character reacts instantly to input, conveying weight through acceleration becomes challenging. Similarly, if the character should feel heavy he cannot react instantly. This relationship will influence most other aspects of movement and the game in general. It is worth considering this balance early in development.

The jump of the character drastically influences the level design. Increased jump height means higher platforms can be reached and more horizontal speed means larger pits can be traversed. Adjusting the parameters of a jump will naturally change what movement is possible. This kind of adjustments are of course important, but also what a game designer would expect. Instead of looking just at whether a jump is possible, the affordance of the jump should be considered. In other words, what is particularly ideal for a certain type of jump. As demonstrated by the analysis of the case studies, jumps can be said to have such affordances. Briefly summarised, the jump in *Mario* affords stomping and vertical landings. Meat Boy's jump affords avoidance both horizontally and vertically. The jump in *Limbo* affords horizontal movement and adaptability. Looking at the affordance of a jump is valuable because it goes beyond the obvious analysis of what is possible with the jump and instead indicates what movement will feel natural with the jump.

The affordances found in the case studies can be used when adjusting a jump. If, for example, a jump is tweaked towards the shape of the jump curve seen in *Mario,* that jump is expected to gain more stomping affordance. Similarly, adding the responsiveness seen in *Super Meat Boy* should give more avoidance affordance. Like this adding a horizontal takeoff velocity will add the affordance seen in *Limbo,* which favours horizontal jumps. Instead of intuitively experimenting, knowing these affordances provide some basic knowledge of what to expect from different shapes of jump curves. The affordances found in this project are based on analysis of only a few games, further research is required to confirm the affordances and expand on the concept.

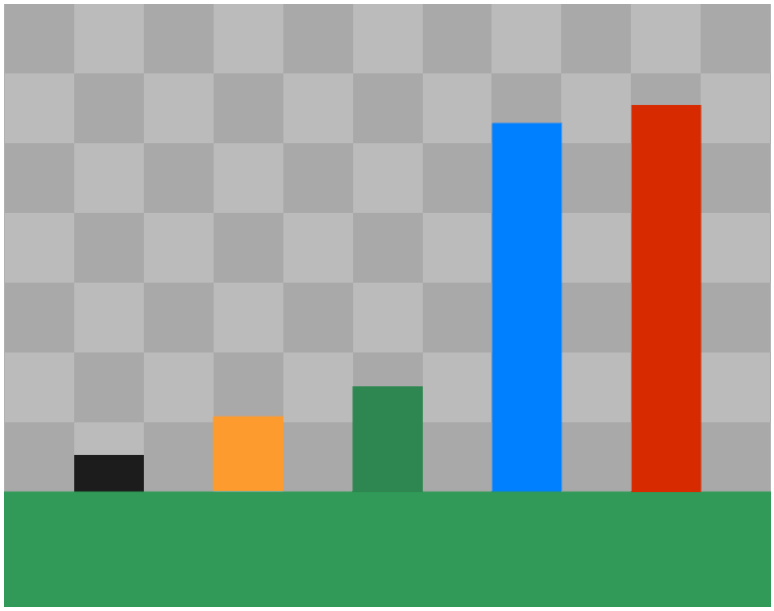


Figure 112: Platform height reachable with a running jump. From left to right: Limbo 0.529u, Braid 1.075u, Spelunky 1.512u, Mario 5.287u, Meat Boy 5.554u.

Figure 112 compares the height of platforms reachable with jump height alone. Mario and Meat Boy has significantly higher jumps than the rest. Both characters jump many times higher than their own height, which allows lots of vertical movement. In contrast, the other three games have a jump height of 1.512 units or less and appear to have a common problem. With a jump height this small vertical movement becomes a challenge. The games approach this challenge in different ways.

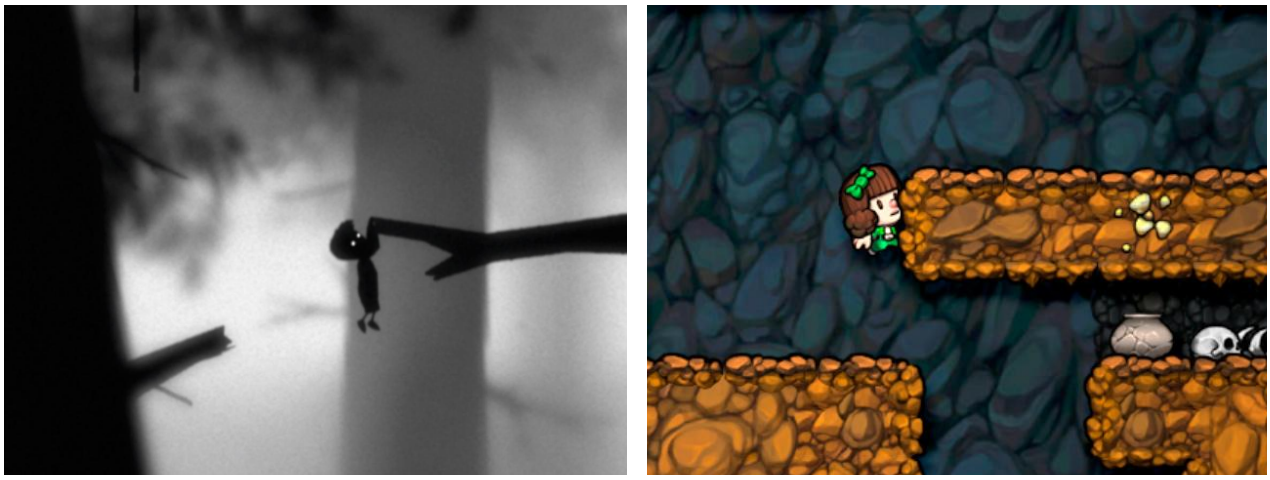


Figure 113: Ledge grabbing in Limbo and Spelunky

One way to compensate for the jump height is to allow the character to grab ledges and pull themselves onto platforms. This significantly increases the height of the platform which can be traversed. This is for example used in *Limbo* and *Spelunky* as seen in figure 113.

Another solution is to use ropes and ladders to allow the character to move vertically without jumping. Again this is the case in *Spelunky* and *Limbo*. The latter in particular uses this limitation of vertical movement to its advantage, as seen in figure 114. Getting down is usually straightforward, but moving up is challenging even with the ledge grabbing. The player can place ropes which allow upwards movement, however this is a resource which must be managed carefully. This rope resources would not work, if the jump had been higher allowing free upwards movement. Furthermore, the general flow going from the top to the bottom of a level would be altered completely.

Figure 114: In Spelunky ropes becomes a resource allowing vertical movement which is otherwise restricted

Figure 115: Small platforms in Braid reachable without ledge grabbing

Figure 116: Ladders used to move vertically in Braid

Braid similarly has a small jump with a height of just 1.074 units. However, in *Braid* the character cannot grab the ledge. The result of this can be seen throughout most of the levels. Small platforms with a height shorter than 1 unit are common as they are can be reached by jumping. Example of these platforms can be seen in figure 115. In addition, *Braid* uses a lot of ladders to allow vertical movement throughout the levels as seen in figure 116.

This demonstrates how a limited jump height can have significant influence on the level design in particular in regards to vertical movement. Taking this into consideration when designing movement should help avoid problems later. It might even be utilised as part of the game design as seen with *Spelunky*.

Future work

This project leaves a lot of questions open for further research. An obvious direction would be to measure and analyse more games to get more generalised knowledge about movement. This would naturally lead to the movement framework being challenged and extended. Another direction would be to improve the approach developed. The tools created for this project are functional, but still in an early state of development. Further work is required to take them from prototypes to final products, which should also lead to further refinement of the general approach of measuring and analysing movement.

This project contains some analysis of controllers, their limitations, and what to be aware of when handling input. Further work could involve more in depth research of controllers. For example, measuring more controllers or a larger variety of controllers. It might also be interesting to measure how controllers change over time and if the type of games played with the controller has an influence on how it wears out.

Working with the prototypes for this project, it became apparent that camera perspective plays an essential role in how speed is perceived. The movement of Meat Boy is uncontrollably fast with the camera from *Mario*. Similarly, if *Limbo* used the same camera as *Super Meat Boy,* the movement feels painfully slow. Keren (2015) wrote a detailed article about camera movement in side-scrollers and combining his research with this project could be the starting point for of a new study. The focus could be the relationship between camera perspective and perceived movement speed.

The relationship between movement and level design could also be a topic for further studies. The feeling of moving through a level is greatly influenced by this relationship. Researching this further could help game designers create both claustrophobic spaces and spaces which feel free and open.

Conclusion

Looking back at the question from the introduction, we can now explain some of the differences between the way Meat Boy and Mario jumps. As expected their movement did have a very different pace. However, the two games were not just parameters being tweaked, but instead had very different implementations. To mention just a few: Mario uses a terminal velocity and Meat Boy does not. Mario adds force during his jump, while Meat Boy uses just the initial takeoff velocity. Mario continues moving when jump input is released and Meat Boy stops instantly.

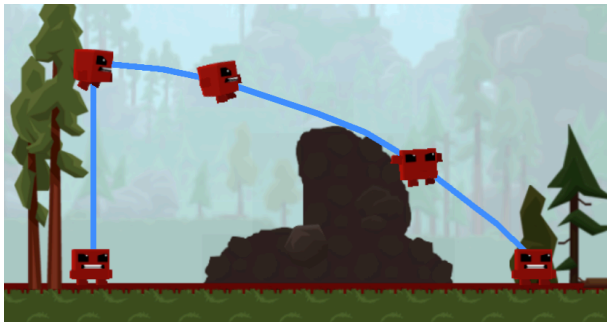


Figure 117: Jump curve based on intuition

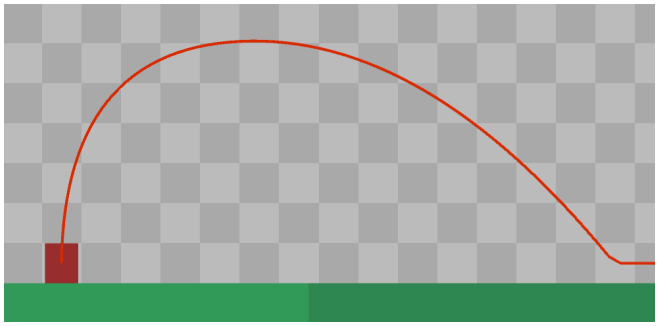


Figure 118: Jump curve based on measurements

The assumption from the introduction regarding air control in *Super Meat Boy* can now be compared to the measurements. When this project started Meat Boy was assumed to have a variable acceleration during his jump, resulting in the jump curve seen in figure 117. After measuring the jump it is possible to more accurately describe what is going on. The measured and modelled jump can be seen in figure 118. The acceleration is actually constant, but it takes 0.516 seconds to accelerate to maximum speed. Since it takes 0.499 seconds to reach the peak of the jump the player will only gradually feel the horizontal speed when going up. Only close to and after the peak will the horizontal speed significantly influence the curve of the jump. This could explain why it was initially assumed to be a variable acceleration. The analysis of the jump gave a more accurate description of what happens. This gives a better understanding of the movement both in terms of analysing the jump as well as the implementation.

The goal of this project was to provide tools and knowledge to support game designers in the process of developing movement in platform games. To establish a direction for the project existing work about jumping was researched. The focus of the project became to develop an approach for measuring and analysing jumping in existing platform games.

The first step of the approach was to simulate input to a game otherwise running normally. To accomplish this the Input Simulator was developed, which simulates a series of controller input. By adjusting the inputs various movement scenarios can be simulated and explored. The tool also draws an overlay on the game, which shows the currently simulated state as well as a timer, both needed later when measuring the movement. This tool was designed to be as generally applicable as possible and easy to extend when needed. While input is being simulated the resulting movement can be recorded using screen recording software, which records the game as well as the overlay and outputs a video. For this project GeForce ShadowPlay was used as it consistently produced high quality videos.

To measure the movement in the recording a Measurement Tool was created. The video is loaded into this tool where the user can navigate it frame by frame. A cross hair is placed on the character for each frame of the recording in order to measure position over time. Any imprecisions in timing between Input Simulator, game, and recording can be taken into consideration by checking the timer from the overlay. The overlay also provides information about the state of the controller at any given frame. When the movement has been measured the tool can export the data as a text file.

The file is imported into Microsoft Excel where the movement can be visualised and further analysed. A model of the movement can now be created to fit the measurements. This model consist of how speed changes over time and presents one way the movement of the game could be implemented. With, for example, a character running forward, the model will consist of acceleration, deceleration and maximum ground speed. The model also contains the specific values of each of these parameters.

This approach was used to measure and model the movement of three case studies *Super Meat Boy, Limbo* and *Mario.* Based on the case studies a movement framework was defined which describes the basic jump as well as left and right movement. With the framework in place the game feel of the cases studies were analysed and the affordances of their jumps were discussed and compared. The framework and its terminology can be used in general to communicate about different aspects of jumping. When designing a jump the framework can be used as an overview of aspects to consider and having this overview from the beginning of a design process saves time. For example, instead of iteratively experimenting with takeoff velocities, the framework provides a range of options which can be used as starting points.

The analysis of the case studies provide detailed information about the implementation of three classic and contrasting platform games. It is valuable to know the difference between these implementations as wells as how different aspects of each implementation influence game feel. Instead of starting from scratch a game designer can consider the solutions from the case studies. Furthermore, when implementing a specific part of a jump the information from the case studies suggest how the change might influence game feel. Having this knowledge readily available leads to more well informed choices.

The tools developed, the approach defined, and the framework can be used by game designers to measure, model, and analyse movement in platform games. As demonstrated with the case studies, this will provide detailed information about the jump in a game. A general approach is valuable as it allows game designers to conduct similar case studies, but with games particularly relevant for their production.

A prototyping tool was created for this project. It is a basic 2D platform game where the movement of the character can be adjusted with a wide range of parameters. The tool can visualise the curve of a jump while parameters are being adjusted or the

environment changed. This gives immediate feedback by showing the result of adjustments without having to start the game and play. The basic movement of the characters from the three case studies were replicated inside this tool. The prototyping tool can be used by designers to explore and experiment with the different implementation. It can also be used to check how a certain detail will influence a jump before implementing it. The prototyping tool, including the source code, will be released under an open source license. This way any code from the tool can be copied and repurposed.

The information and tools presented in this thesis should give game designers a solid foundation for creating their own platform games. The detailed information regarding jumping in three classic platform games can be used to support the development. Information relevant both in regards to the implementation and what game feel to expect when adjusting a jump. If a different game is of particular interest, this project provides an approach for measuring and analysing the movement from that specific game. Instead of spending time reinventing the jump, developers can focus on making the jump perfect for their game, or adding that new detail which will make the jump stand out from the rest.

Appendix 1, Limbo Input

This appendix will describe how input is adjusted in *Limbo* before being used to control the character. Measuring run speed with different stick input partly revealed the relationship between input and movement, but also indicated that there was more going on. Browsing the source code of the game provided information for this more detailed breakdown. The source code for *Limbo* is similarly to most games not publicly available, therefore this way of exploring a game is not generally applicable. However for this project it was possible to take a closer look.

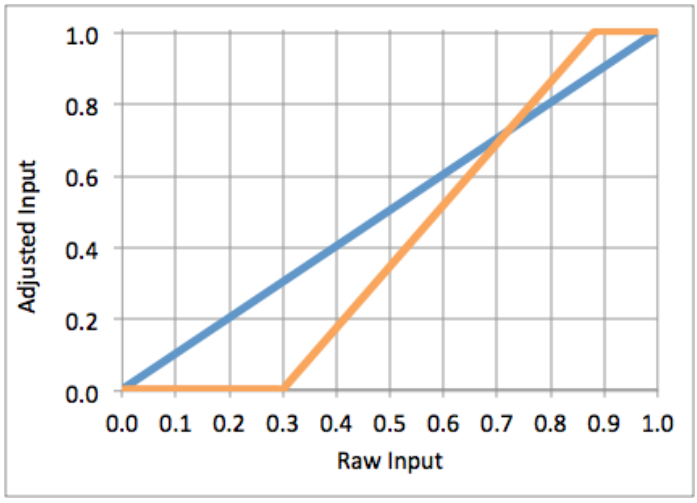


Figure 119: The blue curve shows unaltered input.
The orange curve shows input adjusted to have a lower and upper plateau.

After receiving input from the controller a spherical deadzone of 0.3 is applied and the resulting input normalized. This result in a lower plateau. Next, the input is multiplied by 1.2 and any values above 1 is clamped back to 1. The difference between the original and adjusted input can be seen in figure 119.

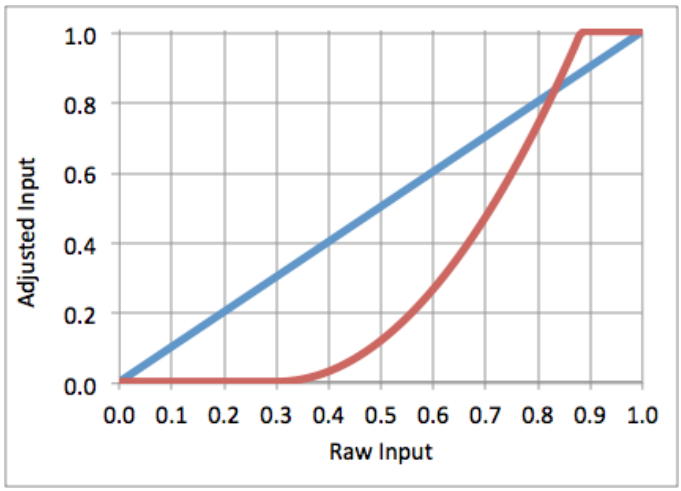


Figure 120: The blue shows unaltered input. The red curve shows input after being squared.

The adjusted input is now squared which creates an upwards curve while retaining the lower and upper plateaus as seen in figure 120. The lower plateau works as a traditional deadzone preventing unwanted input when the thumbstick is near the center. The upper plateau guarantees a consistent and maximised input when the stick is pressed 90% of the way. This also clamps any overshooting input with a length larger than 1. The movement of *Limbo* is analogue in nature and this upper plateau prevents cases where the player could get a slightly increased run speed by perfectly aligning the stick. Instead any input within the upper plateau gives maximum input, securing consistency when moving and jumping. By squaring the input the relationship between physical controller stick and the input value is no longer linear. The effect of moving the stick gets larger as input increase. This is expected to give more detailed control with lower input, while higher input will give more rapid changes. When moving the character this results in more control when sneaking, while running tends towards maximum speed.

A final detail is a check applied specifically when running left and right. Input which differ more than 63 degrees from the horizontal input axis are ignored. This works like a deadzone based on the angle of the input vector instead of the length. Without this check input going mostly upwards could result in slow forward movement. With this check the required direction of an input can be defined without altering the rest of the input adjustments previously described.

Appendix 2, Attacthments

Input Simulator

Simulates XBox controller input

Microsoft Visual Studio project + Windows executable

Input Measurer

Used to measure movement in videos, also referred to as the Measurement Tool

Xcode project and OS X executable

Prototype

Simple 2D platformer used to experiment with character movement

Unity project

Input Plotter

Visualises and plots controller input

Unity project

Fréchet_excel.bas

Excel module for calculating the Fréchet distance between two curves

RMSD_excel.bas

Excel module for calculating the root mean square deviation between two value sets

Data

Data, analyses, and model for the games measured during this project.

Excel file with movement visualised as curves and models of movement

.txt files with raw measurements of movement

References

Bibliography

Aldrich, J. (2012) *Complete Guide To SMB's Physics Engine* [Accessed: 1 November 2015]
http://s276.photobucket.com/user/jdaster64/media/smb_playerphysics.png.html

Aldrich, J. (2012) *Super Mario Bros. 3 Physics* [Accessed: 2 November 2015]
http://s276.photobucket.com/user/jdaster64/media/smb3_physics.png.html

Aldrich, J. (2012) *LUIGI'S PHYSICS (Super Mario Advance 4)* [Accessed: 2 November 2015]
http://s276.photobucket.com/user/jdaster64/media/sma4_physics.png.html

Aldrich, J. (2012) *A Guide to Super Mario World Physics* [Accessed: 2 November 2015]
http://s276.photobucket.com/user/jdaster64/media/smw_physics.png.html

Alexandre (2014) *The Making of Toto Temple Deluxe: Platforming*, Juicy Beast [Accessed: 3 November 2015]
http://juicybeast.com/2014/02/the-making-of-toto-temple-deluxe-platforming-part-1/

Begy, J. (2010) The History and Significance of Jumping in Games, Paper, Academia [Accessed: 1 November 2015]
https://www.academia.edu/1927600/The_History_and_Significance_of_Jumping_in_Games

Begy, J. (2010) *The History and Significance of Jumping In Games*, GAMBIT [Accessed: 1 November 2015]
http://video.mit.edu/watch/gambit-research-video-podcast-episode-8-part-1-the-history-and-significance-of-jumping-in-games-6238/

Butler, T. (2014) *The Rise of The Jump*, Polygon [Accesed: 01 November 2015]
http://www.polygon.com/features/2014/1/20/5227582/the-rise-of-the-jump

D'Angelo, D. (2015) *Shovel Knight: Plague of Shadows Mobility Design*, Gamasutra [Accessed: 1 November 2015]
http://gamasutra.com/blogs/DavidDAngelo/20150831/252543/Shovel_Knight_Plague_of_Shadows_Mobility_Design.php

Daniels, D. (2013) *Game Feel - Part 2* [Accessed: 2 November 2015]
https://derek-daniels.squarespace.com/blog/2013/2/23/game-feel-part-2

Eurogamer (2015) *Miyamoto on World 1-1: How Nintendo made Mario's most iconic level*, Eurogamer [Accessed: 1 November 2015] https://www.youtube.com/watch?v=zRGRJRUWafY

Fineberg, D. (2015) *Designing a Jump in Unity*, Gamasutra [Accessed: 3 November 2015] http://gamasutra.com/blogs/DanielFineberg/20150825/244650/Designing_a_Jump_in_Unity.php

Fordyce, T. (2015) *Usain Bolt delivers his greatest miracle in beating Justin Gatlin* [Accessed: 8 November 2015] http://www.bbc.com/sport/athletics/34033556

Guinness World Records (2012) *Highest standing jump* [Accesed: 1 November 2015] http://www.guinnessworldrecords.com/world-records/highest-standing-jump/

Hesselgren, A. (2012) *DeadZones: Part One* [Accessed: 3 November 2015] http://ludopathic.co.uk/2012/02/28/no-deadzones/

IndieGame: The Movie (2012) *Indie Game: The Movie - Online Extra - CONTROL* [Accessed: 1 November 2015] https://vimeo.com/34928357

Keren, I. (2015) *Scroll Back: The Theory and Practice of Cameras in Side-Scrollers*, Gamasutra [Accessed: 15 November 2015] http://gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll_Back_The_Theory_and_Practice_of_Cameras_in_SideScrollers.php

Lefky, A. & Gindin, A. (2007) *Acceleration Due to Gravity: Super Mario Brothers* [Accessed: 1 November 2015] http://hypertextbook.com/facts/2007/mariogravity.shtml

Monteiro, R. (2012) The guide to implementing 2D platformers, [Accessed: 1 November 2015] http://higherorderfun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/

Olney, M. (2013) *Model of Super Meat Boy's Physics* [Accessed: 2 November 2015] http://mpolney.galineer.com/smb.html#Frame_Step

Pajot, L. & Swirsky, J. (2012) *Indie Game: The Movie*

Pignole, Y. (2014) Platformer controls: how to avoid limpness and rigidity feelings, Gamasutra [Accessed: 1 November 2015] http://www.gamasutra.com/blogs/YoannPignole/20140103/207987/Platformer_controls_how_to_avoid_limpness_and_rigidity_feelings.php

Pignole, Y. (2013) *The hobbyist coder #1: 2D platformer controller*, Gamasutra [Accessed: 1 November 2015]
http://www.gamasutra.com/blogs/YoannPignole/20131010/202080/The_hobbyist_coder_1_2D_platformer_controller.php

Rogers, T. (2009) *let's talk about jumping*, Kotaku [Accessed: 2 November 2015]
http://kotaku.com/5420545/lets-talk-about-jumping

Rogers, T. (2010) *In Praise Of Sticky Friction*, Kotaku [Accessed: 2 November 2015]
http://kotaku.com/5558166/in-praise-of-sticky-friction

Saltsman, A. (2010) *Tuning Canabalt*, Gamasutra [Accessed: 1 November 2015]
http://www.gamasutra.com/blogs/AdamSaltsman/20100929/88155/Tuning_Canabalt.php

Sonic Retro (2014) *Sonic Physics Guide*, Sonic Retro [Accessed: 2 November 2015]
http://info.sonicretro.org/Sonic_Physics_Guide

Stenning, J. (2011) *C# – Screen Capture and Overlays for Direct3D 9, 10 and 11 Using API Hooks* [Accessed: 3 November 2015] http://spazzarama.com/2011/03/14/c-screen-capture-and-overlays-for-direct3d-9-10-and-11-using-api-hooks/

Stenning, J. (2010) *C# – Screen Capture with Direct3D 9 API Hooks* [Accessed: 3 November 2015]
http://spazzarama.com/2010/03/29/screen-capture-with-direct3d-api-hooks/

Sutphin, J. (2013) *Doing Thumbstick Dead Zones Right* [Accessed: 8 November 2015]
http://www.third-helix.com/2013/04/12/doing-thumbstick-dead-zones-right.html

Swink, S. (2008) *Game Feel: A Game Designer's Guide to Virtual Sensation,* Morgan Kaufmann: Amsterdam, Netherlands.

Thompson, T. (2015) *The Legacy of Super Mario Bros.* [Accessed: 1 November 2015]
http://t2thompson.com/2015/06/30/the-legacy-of-super-mario-bros/

Venturelli, M. (2014) *Game Feel Tips I: The Ghost Jump*, Gamasutra [Accessed: 7 November 2015]
http://gamasutra.com/blogs/MarkVenturelli/20140810/223001/Game_Feel_Tips_I_The_Ghost_Jump.php

Venturelli, M. (2014) *Game Feel Tips II: Speed, Gravity, Friction*, Gamasutra [Accessed: 1 November 2015]
http://gamasutra.com/blogs/MarkVenturelli/20140821/223866/Game_Feel_Tips_II_Speed_Gravity_Friction.php

Venturelli, M. (2014) *Game Feel Tips III: More On Smooth Movement*, Gamasutra [Accessed: 1 November 2015]
http://gamasutra.com/blogs/MarkVenturelli/20140821/223866/Game_Feel_Tips_II_Speed_Gravity_Friction.php

Warren, J. (2014) *Why Does Mario's Jump Feel So Awesome*, PBS Game/Show [Accessed: 1 November 2015]
https://www.youtube.com/watch?v=z2oV2DQ2dEA

West, M. (2013) *Measuring Responsiveness in Video Games* [Accessed: 2 November 2015]
http://cowboyprogramming.com/2008/05/30/measuring-responsiveness-in-video-games/

Ludography

Bits & Beasts (2015) *Feist*, Finji. PC

Brøderbund (1989) *Prince of Persia*, Brøderbund. Apple II
Yacht Club Games (2014) *Shovel Knight*, Yacht Club Game. PC

Delphine Software (1992) *Flashback*, U.S. Gold. Amiga, PC

Capcom (1988) *Bionic Commando*, Capcom. Nintendo Entertainment System

Juicy Beast (2015) *Toto Temple Deluxe*, Juicy Beast, PC

Konami (1987) *Castlevania*, Nintendo & Konami. Nintendo Entertainment System

Media Molecule (2008) *LittleBigPlanet*, Sony Computer Entertainment. Sony PlayStation 3

Moon Studios (2015) *Ori and the Blind Forest*, Microsoft Studios. Microsoft Xbox One

Mossmouth (2013) *Spelunky*, Mossmouth. PC

Nintendo(1981) *Donkey Kong*, Nintendo. Arcade

Nintendo (1985) *Super Mario Bros.,* Nintendo. Nintendo Entertainment System

Nintendo (1988) *Super Mario Bros. 3*, Nintendo. Nintendo Entertainment System

Nintendo (1990) *Super Mario World*, Nintendo. Super Nintendo Entertainment System

Nintendo (2003) *Super Mario Advance 4: Super Mario Bros. 3*, Nintendo. Game Boy Advanced.

Number None, Inc. (2009) *Braid*, Number None, Inc. PC

Playdead (2011) *Limbo*, Playdead & Microsoft Studios. PC

Semi Secret Software (2009) *Canabalt*, Semi Secret Software. PC

Sonic Team (1991) *Sonic the Hedgehog*, Sega. Sega Genesis

Team Meat (2010) *Super Meat Boy,* Team Meat. PC

Software

Bruun, P. (2014) *Monitor Components* [Accessed: 3 November 2015] https://www.assetstore.unity3d.com/en/#!/content/23765

EasyHook, CodePlex [Accessed: 2 November 2015] https://easyhook.codeplex.com/

Nvidia, *GeForece ShadowPlay* [Accessed: 3 November 2015] http://www.geforce.com/geforce-experience/shadowplay

OllyDbg [Accessed: 2 November 2015] http://www.ollydbg.de/

SharpDX [Accessed: 2 November 2015] http://sharpdx.org/

Stenning, J. (2012) *Direct3DHook* [Accessed: 3 November 2015] https://github.com/spazzarama/Direct3DHook

Images

Bruns, R. (2009) *SuperMarioBros3Map1-1BG,* Nesmaps [Accessed: 18 November 2015] http://www.nesmaps.com/maps/SuperMarioBrothers3/SuperMarioBros3Map1-1BG.html

Bruns, R. (2012) *SuperMarioBros3Map3-3BG*, Nesmaps [Accessed: 18 November 2015] http://www.nesmaps.com/maps/SuperMarioBrothers3/SuperMarioBros3Map3-3BG.html

Bruns, R. (2012) *SuperMarioBros3Map7MiniFortress2BG*, Nesmaps [Accessed: 18 November 2015] http://www.nesmaps.com/maps/SuperMarioBrothers3/SuperMarioBros3Map7MiniFortress2BG.html

Frogacuda (2007) *Donkey Kong Screen 3*, Wikipedia [Accessed: 3 November 2015] https://en.wikipedia.org/wiki/File:Donkey_Kong_Screen_3.png

NeoGenPT (2010) Prince of Persia (1989 video game) IBM PC Version gameplay, Wikipedia [Accessed: 3 November 2015] https://en.wikipedia.org/wiki/File:Prince_of_Persia_(1989_video_game)_IBM_PC_Version_gameplay.gif

Xbox One Wireless Controller, Microsoft [Accessed: 21 November 2015] http://www.xbox.com/en-US/xbox-one/accessories/controllers/wireless-controller