Procedural Generation of 2D Levels for a Motion-based Platformer Game in Unity with Large Amount of Movement

Niklas Erhard Olsson  <niker418@student.liu.se>

Supervisor : Aseel Berglund, Erik Berglund
Examiner : Arne Jönsson
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Niklas Erhard Olsson <niker418@student.liu.se>
Abstract

This thesis mainly describes and implements a new way of analyzing motion generated when playing a motion controlled game. It also describes the implementation of automatic level generation together with the utilization of Unitys excellent new 2D tools. The motion controller used to play the prototype game is ported and implemented with Unitys own shader language and stored as a reusable prefab for any Unity project. A new specific method of analyzing the motion mapped to the level is implemented in Unity. Some game specific analyses is presented with the said method, and examples of how the method can be used for more and richer analysis's is discussed.
Acknowledgments

First of all, I would like to thank Erik Berglund for the continuous feedback during the project and his will to always want to jump in and help remove any obstacles in my way. Whether it be related to code, Unity, or basis for theory, Erik always were inclined to help. I want to thank Aseel for the great and quick feedback of the report, to keep it on the right track, making sure it turned out well. I want to send an extra appreciation to my friend Andreas Järvelä who’s always keen in discussing appropriate solutions to both minor and bigger problems.

To Johan Nåtoft, Fabian Petersen, Sebastian Lindmark for helping me gather the required data. Thank you for your patience during the long play sessions.

I also want to thank my lunch mates that I’ve gotten to know much better during the course of this thesis. Sara Bergman and Jonatan Pålsson, thank you for interesting discussions.

If this is also a time to thank the people during the course of my studies, I want to thank my wonderful mother, Vivianne Erhard, whom I often speak to when walking to the supermarket, who’s always there for me. My father, Stefan Olsson for the support and everything he’s done for me. Thank you Mariana Olsson for your monthly support, it has meant a lot. Thank you Karin Fegraeus for the warm welcomes when I finally come home, for the holidays and everything you do for us. I want to thank my twin brother, Dan Erhard Olsson, for being my brother, and for the opportunity to work and help with his latest and biggest project during my studies. My dearest friends from back home, Carl Skoglund and Max Sydorw for keeping me grounded throughout all the stress.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>0.1 Glossary</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Aim</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Delimitations</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Background</td>
<td>4</td>
</tr>
<tr>
<td>2 Theory</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Motion-controlled games</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Game design</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Procedural content generation</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Graphics programming</td>
<td>9</td>
</tr>
<tr>
<td>3 Method</td>
<td>10</td>
</tr>
<tr>
<td>3.1 The Game</td>
<td>10</td>
</tr>
<tr>
<td>3.2 The level generator</td>
<td>10</td>
</tr>
<tr>
<td>3.3 The motion controller</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Measurement of movement</td>
<td>16</td>
</tr>
<tr>
<td>3.5 Method of data gathering</td>
<td>19</td>
</tr>
<tr>
<td>4 Results</td>
<td>24</td>
</tr>
<tr>
<td>4.1 The Game</td>
<td>24</td>
</tr>
<tr>
<td>4.2 The level generator</td>
<td>25</td>
</tr>
<tr>
<td>4.3 The motion controller</td>
<td>27</td>
</tr>
<tr>
<td>4.4 Measurement of movement</td>
<td>28</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>44</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>44</td>
</tr>
<tr>
<td>5.2 Method</td>
<td>47</td>
</tr>
<tr>
<td>5.3 The work in a wider context</td>
<td>49</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>6.1 Future work</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Figures

3.1 Level layout - list of enums visualized .................................................. 11
3.2 Level layout - visualized as 2d array ...................................................... 12
3.3 Grid used for checking for collision ......................................................... 12
3.4 Path generated ......................................................................................... 13
3.5 Block filled with tiles .............................................................................. 13
3.6 Empty 10x10 block generation ................................................................. 13
3.7 A single block generated with added decor .............................................. 14
3.8 Output texture, black pixels are transparent ............................................. 16
3.9 Calculating total movement method ......................................................... 17
3.10 Normalizing total movement grid ............................................................ 17
3.11 Heat map example .................................................................................. 17
3.12 Line motion intensity visualization example .......................................... 18
3.13 Circle motion intensity visualization example ........................................ 19
3.14 Life: 1=green, 2=yellow, 3=blue .............................................................. 19
3.15 Overview of the iterative process ............................................................ 20
3.16 The level generator parameters ............................................................... 21
3.17 Level 1 ....................................................................................................... 22
3.18 Level 2 ....................................................................................................... 22
3.19 Level 3 ....................................................................................................... 22
4.1 The game ..................................................................................................... 25
4.2 The level generator window tool ............................................................... 26
4.3 The web camera prefab in editor ............................................................... 27
4.4 Level 1: Score improvement over times played excl. failed attempts .......... 29
4.5 Level 1: Score improvement over times played incl. failed attempts .......... 29
4.6 Score improvement over times played excl. failed attempts ..................... 30
4.7 Worst performance total movement image(1 205 576) .............................. 30
4.8 Best performance total movement image(1 177 609) ............................... 30
4.9 First run ..................................................................................................... 31
4.10 Best run ................................................................................................... 31
4.11 Score improvement over times played excl. failed attempts .................... 32
4.12 First run (3 630 025) ............................................................................... 32
4.13 Second run (3 334 254) ......................................................................... 32
4.14 Next best run (1 364 354) ................................................................. 32
4.15 Best run (1 249 076) ............................................................................... 32
4.16 Worst run ................................................................................................. 33
4.17 Best run ................................................................................................... 33
4.18 Score improvement over times played excl. failed attempts .................... 34
4.19 Worst run (6 229 934) ......................................................................... 34
4.20 Next worst run (4 849 190) ................................................................. 34
4.21 Next best run (2 552 547) ................................................................. 34
4.22 Best run (1 519 336) ........................................................................... 34
4.23 Worst run ................................................. 35
4.24 Best run .................................................. 35
4.25 Score improvement over times played excl. failed attempts .............................................. 36
4.26 First run (5,783,984) ....................................... 36
4.27 Next worst (2,773,430) ................................. 36
4.28 Next best run (1,673,295) ......................... 36
4.29 Best run (1,176,747) .................................... 36
4.30 First run .................................................. 37
4.31 Best run .................................................. 37
4.32 Score improvement over times played excl. failed attempts .............................................. 38
4.33 First run (1,612,777) .................................... 38
4.34 First completed run (1,139,575) ..................... 38
4.35 Next best run (1,227,902) ......................... 38
4.36 Best run (1,072,861) .................................... 38
4.37 Next best .................................................. 39
4.38 Best run .................................................. 39
4.39 Score improvement over times played excl. failed attempts .............................................. 40
4.40 Worst performance (1,865,063) .................... 40
4.41 Worst performance (1,073,432) .................... 40
4.42 Next best performance (1,347,143) ............... 40
4.43 Best performance (1,516,359) .................... 40
4.44 Player 0 - Motion intensity on map .................. 40
4.45 Worst run .................................................. 41
4.46 Best run .................................................. 41
4.47 Score improvement over times played excl. failed attempts .............................................. 42
4.48 Worst performance (2,054,698) .................... 42
4.49 Next worst performance (1,987,782) .......... 42
4.50 Next best performance (1,558,895) .............. 42
4.51 Best performance (1,420,903) .................... 42
4.52 Worst run .................................................. 43
4.53 Best run .................................................. 43
List of Tables

4.1 Player 0 - Score and motion ................................................. 30
4.2 Player 1 - Score and motion ................................................. 33
4.3 Player 2 - Score and motion ................................................. 35
4.4 Player 3 - Score and motion ................................................. 37
4.5 Player 4 - Score and motion ................................................. 39
4.6 Player 0, Lvl 2 - Score and motion ........................................ 41
4.7 Player 0, Lvl 3 Score and motion ......................................... 43
0.1 Glossary

Below is some important key words and terms with their respective definition that is used frequently throughout the report. The glossary is meant to aid the reader through all of the acronyms.

**Assets**
Assets in Unity refers to anything used in the development. Game objects, scripts, 3D models, Sprites etc.

**Prefab/Prefabricated**
Unity has a Prefab asset type that allows you to store a GameObject all set up and complete with properties and components. It acts as a template from which you can create new object instances in the scene.

**SceneInstantiator**
The SceneInstantiator is a component made in tandem with the level generator in this project. It holds the game objects, the prefabs and instantiates the actual scene in Unity from the level data generated by the level generator.

**EditorWindow**
You can create any number of custom editor windows in Unity. If you are at all familiar with Unity, they work exactly like the inspector. They are great to add sub-systems for your game. They can aid you alot in the development, by providing a GUI for executing some code that you have written by just a click.

**GPU**
Graphics processing unit.

**HLSL**
High-Level Shader Language.

**GLSL**
The OpenGL shading language, based on the programming language C.

**Cg**
Nvidia Cg shading language.
The gaming industry is still growing rapidly to this date. New and different ways of interaction is constantly improved and explored. Although new ways don’t always mean improved ways of interaction where for example the market of Kinect has stagnated. Microsoft pulled the plug on the Kinect in 2017 \cite{23}, where it was said that the Kinect wasn’t reliable enough, the games produced weren’t as good as they could’ve been and the hype around it faded. The motion controllers really looked like the future as an interesting way of interaction with obvious positive effects for the health. Instead of sitting down playing games, you had to move in some way to interact, often standing up. The idea of having a category where body motion controls the game is still an attractive one to keep alive. The use cases of such interaction methods can be of great use where people needs an extra motivation for exercise. As we spend more and more time indoors, an opportunity of physical activation would therefore could be of great complement to some groups of people.

But what did the games previously lack? According to Matt Weinberger in \cite{24} the games were not what they could’ve been, the interaction needed much space and overall, the gaming experience didn’t work as well as predicted. Movement is strongly connected to feelings, and ideally one would want the movement to induce good feelings.

Automatically generated game content is always an interesting field of game programming. Building levels, models and creating textures is very time-consuming and therefore also expensive \cite{6}, \cite{25}. It would certainly be more efficient and more profitable for the game creators to have some of these created automatically or procedurally, offline or online. So why don’t all game producers use methods for automatically generating game content? Because procedural content generation is difficult and not always applicable. Firstly the game on paper needs to have some or a big part of it applicable to automation that will benefit in terms of development cost for it to be worth implementing. The generator has to give credible and satisfying content/material for the gamer and at the same time also has to be consistent with what the artist had in mind. A well-made real-time generator can be extremely fulfilling for the ever-exploring player, adding to the replayability of the game. Other useful scenarios include generators that aid the developers by generating uncompleted or imperfect, yet interesting, content that the developer then can modify to perfection.

This report will specifically look into the correlation between game layouts and elements of a 2D platformer game and the physical movement the levels within it produces. Weaved
into the algorithm generating the 2D levels, is the desired amount of movement produced by
the level.

1.1 Motivation

There are several reasons to develop a good procedural game content producer. The obvious
one that it would be more cost and time efficient for much of the content to be generated
automatically rather than being created by the hand of a human artist. Humans are slow in
comparison. From a technical stand point, if the content is being generated online it lowers the
on-disk memory demand for the game, but may decrease the performance during game play
when the generating algorithm kicks in. On the other hand if the content is being generated
offline, it can decrease the cost and time of development, should the content generator be well-
implemented. It is however a question of how much it would be profitable in the long term
of the game development to have such a tool, counter to how much time it takes to develop
those tools. Another reason is that procedural game generators can add to the replayability
of the game\footnote{25}, with the generator being able to produce unique content over and over again.

Traditional games and most games dominating the market have carefully composed level
design which the players have little control over. The levels, weapons, objects, items and other
content is often all predefined and fixed with a small set of possible variations to be made by
the player. The amount of time spent creating varied content by hand makes up a big part
of the overall development\footnote{25}, for instance when creating hundreds of items with predefined
stats and capabilities. Or for when the level designers design the levels broadly, when they just
need to put some random elements to it, to when they have to carefully put some elements
at the right place at the right time. The content creation of the top games has become the
bottleneck when it comes to development cost and time\footnote{25}\footnote{6}. The part where you need
a rough level design and randomization to it could profitably be generated by a generator
and leave the careful design of some elements to the human artist. Enabling the artist to
spend more time on the important aspects of the human designers skill set. Unity has new
and excellent 2D creator tools, making editing and adding of tiles and game objects for a 2D
platformer fairly easy, and it is why the generator is developed with that engine.

1.2 Aim

The aim of this project is first to build a robust 2D level generator in Unity that utilizes Unitys
new 2D creator tools. The goal is a generator that procedurally generates varying levels.
Another important part of the project is to implement the web camera motion controller in
Unitys own shader language HLSL and enable it to be easily reused in different projects. The
levels generated combined with the web camera motion controller is the basis for analysis of
human motion mapped to the levels. The thesis aim is to provide data as to what different
designed levels and layouts of a traditional 2D platformer, controlled with human motion,
produces in terms of moving intensity.

The goal of the generator is not to realize a generator that procedurally produces flawless
content, rather a generator that procedurally generates content where delicate details of the
level can be perfected afterwards by a human level designer in the Unity editor. The aim
however, is to keep the human designer interference as low as possible. By robust meaning
that the generator can reliably produce levels that can be completed and does not contain
unreasonable parts. For instance, checkpoints positioned where they can't be reached. A
successfully implemented generator would allow more focus on game design, and polishing of
critical game components as level design can be a tedious and time consuming task. Some
handmade finishing adjustments to the generated content is most likely inevitable to please
the senses of a human level designer.
1.3 Research questions

- How to generate variated 2D levels for a platformer in Unity, while maintaining large amount of movement?

1.4 Delimitations

As procedural game content generation is quite broadly defined there are two big differences; online versus offline generation meaning that either the algorithm is running in real-time during game play or beforehand or in between games, as an instantiation phase. This report will only investigate how to generate content offline.

1.5 Background

LiU Active lab conducts research on games and ways of interaction with an orientation towards learning and health. They are based at the Department of Computer and Information Science (Institutionen för datavetenskap, IDA) at Linköping University. Many of the motion controlled games in LiU Active lab’s research aim to be fun and physically activating where the body is the controller and what kind of movement constitutes good movement, the kind of movement that induces positive feelings when performed.

Many of the games where previously developed for the web, played with a web camera in the web browser. The motion controller were therefore developed with WebGL. Unity is an excellent game engine that recently got new powerful 2D creator tools and it was considered beneficial for future work to incorporate the same method of image processing into Unity. The motion controller could then, for instance, be stored as a component enabling it to be simple to reuse in different projects.

Similar tile-based 2D generators have been developed under LiU Active lab, but with limited possibility to alter the level postpone generation. Viktor Andersson and Johan Classon successfully developed in 2016 a level generator with a genetic approach that could generate 2D platform levels with a desired difficulty. The target platform were an open-source framework called Phazer and were to be played in the web browser. The game developed was an extension to Tim Ziegenbein work in 2015, where he also developed a level generator for the 2D platform game that combined predefined chunks together to produce a complete level. The chunks could be of size 10x10 of single 1x1 tiles that had to be manually constructed by a human designer. The levels produced in Anderssons and Classons work could be exported and loaded in a program called Tiled, where you could alter and save a new representation of the levels. Unity’s editor, and the belonging 2D creator tools, could provide powerful aid when altering the levels after generation in an improved way compared to previous work. One would not need a representation of a level outside of Unity as it , arguably, would not obviously add more value in our case.
This chapter exposes some important theory and information needed to follow along in the work done in this thesis. It provides theory for motion-based games, some general game design, specific theory regarding motion-controlled games and a more thorough section about procedural content generation.

2.1 Motion-controlled games

As the revealing title, motion-controlled games are controlled by motion, allowing players to interact with the game via different body motions. Motion games was first brought to the market when Nintendo entered the market in 2006 with the Wii-console. The Wii-console was shipped and controlled with a handheld controller that tracked the movement of gestures. Motion controller comes today in different shapes and forms, competitors like Sony Playstation Move resembles the Wii-controller while Microsoft Kinect for instance uses a camera that can track the human skeletons.

There are now several motion controllers on the market, some of them mentioned above, most of which contains really advanced technologies. The Kinect can, as said, track skeletons among other things. It includes a color camera and even a depth sensor and facial recognition. What they all have in common is ways of tracking a human or parts of a human, allowing the human to interact with the game through different gestures.

The question is how advanced the technology needs to be in order function well or as intended in the interaction with a game. In this thesis and associated project there isn’t really a need for a depth sensor, nor skeleton tracking for example. The power of simplicity is often overlooked and many games don’t utilize the full potential of the technologies provided.

2.2 Game design

What constitutes good games and game design in general is a challenging field of game research for its subjectivity. An interesting direction of the research is the aim to automate the generation of entertaining game content. Yannakakis, Togelius and Pedersen aims to construct a computational model of player experience, derived from gameplay interaction that can be used in a fitness function for a level generator. They claim to have trained a model that
2.2. Game design

could predict the frustration of a player with an accuracy of 88.66, but that it is harder to
predict player emotions based solely on controllable features. A more extensive report from
the same trio does a similar attempt in modeling the player experience with regards to con-
trollable features (e.g. number of gaps in the level). The test-bed were a Super Mario Bros
platformer. They statistically observed some positive correlation between reported fun and
some specific game features. The results indicated that the players enjoyed a fast paced game
that included almost constant progress in the level, a lot of running, many enemies killed
and coins collected\cite{10}. They also found, among other correlations, that standing still lead to
frustration with high accuracy, yet lowering the feeling of challenge in the level.

Although this thesis do not focus on contributing to what constitutes good games, it is
necessary to acknowledge some basis for what constitutes a relatively good game in order to
create a general level generator that will produce interesting results. Especially to identify
relevant parameters that need to be adjustable to let the user of the level generator have more
control of the level generation turn-out.

Game design in motion controlled games

We discussed shortly what might contribute to fun levels in general, but what about levels in
games that are controlled by motion controllers? Richard M Ryan et al. states two types of
autonomies motivations in their research in Self-Determination Theory (SDT), intrinsic and
extrinsic motivation\cite{13}.

The former is the type of motivation that are engaging to people because they are simply
interesting, fun or enjoyable. Whereas the extrinsic motivation is goal oriented, where one
engage in a behavior not necessarily enjoyable on its own but needed in order to reach a more
desirable outcome as a result \cite{13}. In the research of SDT, three main variables are identified,
autonomy, competence and relatedness. Accoring to (Ryan et al., 2006, p. 349), autonomy
refers to an individuals’s “sense of volition or willingness when doing a task”, competence to
the “need for challenge and feelings of effectance”, relatedness to the need of being connected
to others and the feeling of being involved in a social environment. In a study by Wei Peng,
Jih-Hsuan Lin, Karin A. Pfeiffer and Brian Winn they map the enjoyment as a satisfaction of
intrinsic needs \cite{12}. If the movement or interaction of a motion controlled game is enjoyable
intrinsically, this could be the foundation of allowing the player to further enjoy the game play
with little frustration. Satisfaction of intrinsic needs of the motion controller combined with a
fast-paced 2D platformer where there are a lot of movement as running, jumping, and clearing
obstacles could be a good test-bed for the data gathering of this thesis. An important aspect
to consider when designing games with motion interaction is how motion affect the gaming
experience. Motion can either lift the gaming experience positively or have a negative impact
on the experience. Static motions where, for instance, a player has to hold his or her hands
in an uncomfortable position for longer periods of time can lead to frustration and impact
the mood of the player negatively, ultimately making the player quit the game. Lindsey et
al. concludes that an increase in body movement leads to a higher engagement level for the
players \cite{7}. Although there is no certain correlation between engagement and positive emotion,
more body movement leads to a higher engagement level, but does not mean that the gaming
experience gets better. When designing the motion controller it is also important to match the
motion that generates the movement of the avatar as closely as possible. As Alissa N. Antle et
al. points out, the design of controllers should take into account the role-related movements
and interaction that is more likely to be performed by the player when wanting to induce a
specific movement \cite{1}. A study conducted by M. Slater et al. showed that by having the
participants walk in a virtual environment, the feeling of presence increased in comparison to
letting the participants walk by interacting with their hands. They concluded that if one aims
for gaming engagement or the feeling of presence it is important to design the interaction in
a way that involves whole-body movement that are as natural as possible \cite{15}. Some obvious
challenges is that feedback of the interaction is hard to produce naturally and it might require large spaces to reflect the virtual space of the game.

2.3 Procedural content generation

As defined by Julian Togelius et al. in an extensive taxonomy, procedural content generation (PCG) refers to creating game content automatically, through algorithmic means \[22\]. PCG can have slightly different definitions, in a book about PCG, written by Julian Togelius, Noor Shaker and Mark J. Nelson, they define it as "PCG is the algorithmic creation of game content with limited or indirect user input" \[21\]. The content in question can be of anything present in a game; terrain, maps, objects, textures, quests, characters, dialogs or even whole 3D worlds etc. The definition used in this thesis align well with the second definition. There are several different approaches in generating content for games. The chosen approach will depend on the aim, constraints or requirements the given problem it should solve. The first thing that categorises PCG is if the algorithm is executed offline (prebuilt) or online (in real time). Another thing to distinguish is the two methods of generating content procedurally; constructive procedural generation and search-based methods. One could further distinguish several more properties of PCG as discussed by Togelius et al in the book \[21\]. Some of which could be; necessary versus optional content, degree and dimensions of control, stochastic versus deterministic etc.

PCG can be very useful and there are several reasons why it should be adopted. As Togelius et al argues in \[22\] the first reason is memory consumption since the content can typically be compressed until needed. Another reason is the same mentioned earlier in this thesis, the cost of manually creating extensive game content. Some examples of games using PCG is Minecraft, where the whole world and content is procedurally generated and in theory, unlimited and unbounded. Another interesting application is the PCG developed in Spore, where the players designed creatures are animated using procedural animation techniques. \[21\]

Constructive generation approaches

Let’s first consider constructive procedural game generation. Basically, what it means is that the generation algorithm is proceeding in only one way, from start to finish with no backtracking or sophisticated evaluation during the generation. This can be well suited for games where the generated maps are valid solutions and where the goal is to generate variation rather than searching for an optimum with regards to some criteria or property.

An example of this is the game Civilization. For you that are familiar with the game Civilization, the game is not a competitive game, and therefore don’t pose balancing challenges in the same way a competitive game would. A strategic game that is highly dependent on positioning and the positioning of resources like the StarCraft series, could then instead benefit from a search-based method to where you evaluate the balance of the map. A rough summary of the way Civilization generated maps are similar to something like the following. You first place small blocks of land, and having them grow in random directions a certain number of steps. One can add, on top of that, functions for generating what kind of land should occupy that part. Something similar could be adapted to a 2D platformer as well, where you place blocks of tiles that grow randomly in the same way until it hits another growing block of tiles. Although it would be of low probability that the resulting platform base is viable due to the strong randomization. \[8\]

Another interesting, and easy to wrap your head around, approach is the use of software agents when generating terrain. The idea is to have several agents with each agent having a specific task, and works on the map much like erosion in nature. As Parberry and Doran suggested in \[15\] you let loose many software agents on an untouched canvas of terrain, having them shaping it collectively. For instance, one agent draws a rough path of a river from higher altitude to the lower point by simple calculation of the terrains angles in the first phase. The
second agent might smooth the path out, simulating many years of friction. The purpose of all agents is to work as natural forces of nature.

**Search based approaches**

The search-based approach is one where you make use of some stochastic search or optimization algorithm or an evolutionary algorithm to search for desired content to be produced. In contrast to constructive procedural content generation mentioned above, the search-based approaches all have some kind of evaluation method during the generation. It has a generate-and-test approach, making several attempts in creating desired content, and keeps only the versions that pass some objective function.

**Evolutionary algorithms**

Evolutionary algorithms are somewhat quite logical to understand if you understand the basics of evolution itself. Its implementation is namely derived from Darwin’s evolution theory. The core idea is to keep a population of individuals, also called candidate solutions or chromosomes. Each generation is run through an evaluation function that determine the score, often called ‘fitness’ of the solutions. The best solutions of the generation get a much higher chance to reproduce and evolve into the next generation. This is done repeatedly. An abstract example of how it may work can be the analogy of sheep. The slowest of sheep is most likely to get eaten by a wolf, thereby the next generation of sheep might be on average slightly faster than the previous generation. The design can be seen as a search process. The precondition for this to work in a desirable manner is firstly that an accepted solution exists. If we then keep iterating over the potential solutions, keeping the improved solutions each iteration and discarding the others will eventually leave us with one of the best solutions for the designed problem.

The core components of search-based algorithms or generic approaches is:

- A search algorithm You need some algorithm to traverse through the search space. Usually a relatively simple algorithm is sufficient to work well enough.

- Content representation You need to represent the artifacts and content of which you want to be able to generate, e.g. resources, objects like trees or things like quests.

- At least one evaluation function The most challenging task for the developers is to define a well-functioning evaluation algorithm that produces the desired results. The output is a kind of overall verdict of how well a generation performs, or how well the solution performs.

Standard evolutionary algorithms or generic algorithms leaves the responsibility for determining the quality of the solutions to only one score function, or fitness function as it is often called. But for generating more complex data structures or models like procedural maps, one fitness function might be insufficient for good results. Although Cameron Browne successfully produces entertaining even whole games with only genetic algorithms using a fitness function that consists of the weighted sum of 57 objectives. The intuitive approach in obviating this problem might be to use this approach with weights for the different values and adding the variables altogether. But the obvious drawback of this approach is that it’s difficult to set the appropriate weight values. And in many cases, the objectives might partially conflict with each other, e.g. say we have three dimensions for a car we want to optimize. Speed, safety and cost. It would be difficult or impossible to explore how these variables or objectives interact with each other, there might even be conflict, in a single dimension. Increase in speed will most likely decrease the objective about safety.

The genetic applicability in our 2D platform case is however questionable. The main purpose of our generating algorithm is to produce varying and valid content, and the complexity of creating a score function that will reduce a completely random noise of tiles that fills up the
whole tilemap, to a playable and viable level all to problematic, and the algorithm would’ve
to iterate through tons of generations. The search space would need to be limited in some
way, narrowed down to a simpler problem. An appropriate application would be to apply the
genetic algorithm to an already established random base level, to evaluate it with regards to
some objective. Objectives could be difficulty, variation or as in our case, a target motion
activity that the level is estimated to produce. The crossover would also need to be thought
through carefully, not make the next generation completely useless.

Offline vs Online generation

Roughly speaking, offline generation is about generating content during development, often
used as an aid to the developers and designers, whereas online generation refers to content
generation in real time, during the actual game play. An example of the former is where an
algorithm creates for instance a base terrain, with texture and vegetation in a 3D world, where
the designers can perfect the world post generation by altering existing design or structures,
adding more details like houses, streams etc. An example of online generation is for instance
when a player walks into a dungeon entrance, and the dungeon is generated seamlessly at the
time the player enters. Naturally, the two approaches have different requirements in terms of
memory usage and speed. As pointed out by Togelius et al in \[22\] online algorithms typically
need to be very fast with a predictable runtime and predictable qualitative results. Offline
algorithms have usually much looser requirements in terms of speed and predictable qualitative
results, since it does not run in real-time nor does the content created need to be playable
right of the bat.

2.4 Graphics programming

Computer graphics, the term, includes pretty much everything that is not text or sound.
Almost all computers can now do some graphics\[4\]. Some of the many topics included in com-
puter graphics are user interface, vector graphics, 3D modeling, GPU design, sprite graphics
and shaders. Originally, shaders in computer graphics was used for shading, refered to the
production of applying the appropriate level of light or color within an image. There are no
reason shaders can not handle other effects and now, shaders are utilized in many different
fields including special effects, post-processing video or even for functions not at all related to
graphics. Elegantly described by Omar Shehata, a shader is simply a program that runs in the
graphics pipeline and tells the computer how to render each pixel \[14\]. Shaders are written
in special shading languages such as OpenGL, or Nvidia Cg. Although there are a bunch of
different shading languages, they are somewhat similar since they all run on the GPU.

A shader’s sole purpose is to return four numbers that represents red, green, blue, and
alpha. That is all they ever do.\[14\]. The main function of the shader runs for every single
pixel on the screen, and returns a color value for that pixel. This allows a graphics programmer
to do calculations and modify and set the pixel colors from the GPU.

There are many different types of shaders, and you can delegate functionality to different
shaders and have them run in a specific order. The CPU can order several rendering passes, for
each frame, and more complex processing can then be done before the final pixels are drawn
to the screen. In our case, we will be using four different shaders, four passes, for each frame
in order to generate the final image of the motion controller.

Textures are useful in computer graphics, the often hold color information and can be
mapped to objects to enhance detail and improve realism. The texel data for a texture in
computer graphics can have many different forms. For example it may contain purely gray
level information which would require only one byte per texel, or it may contain red, green,
blue and alpha (RGBA) that would require 4 bytes per texel. In our case we use the latter
although we are only interested in the alpha value.
This chapter presents the method of implementation of the level generator and implementation of the motion controller, utilizing Unity’s own shader language, powerful engine and 2D creator tools. It also includes the method of measurement and analysis.

3.1 The Game

The game produced suitable for analysis is a 2D platformer where the objective is to travel from start to finish, with as much score as possible. As default, you get more score the faster you complete the level and if you eliminate enemies without losing much time. The score system is setup so that it generally is more punishing to walk backwards to kill an enemy, than to just skip it if missed on the first encounter. You control the game via the motion controller from which the analyzing of your movement will be gathered. The timer-based score is chosen to encourage the player to move as quickly as possible through the level. This will hopefully yield in a more consistent measurement, since it will probably make for less unnecessary pauses throughout the gameplay.

3.2 The level generator

The main goal of the generator together with Unity was to generate 2D levels for a platformer with variations that could easily be altered or changed after the level has been generated. The goal isn’t to get rid of human design, rather to aid the level designer. This is especially true with Unity’s powerful editing tools that allows you to edit the generated version straight away. These simplistic goals or requirements does make search-based methods appear like over engineering. The search-based methods where one uses a genetic approach may in this case not provide significant improvements as to the time it takes to implement.

Not only as the main goal states above, we also want to utilize Unity’s ability to create tools, or editors. An editor window where one could tweak parameters and easily generate levels in the Unity inspector simply by typing and clicking.

The real gain of using methods under, for instance, the category of search-based methods is not obvious in this case. One of the challenges in that approach is to model a suitable and true model of the problem. The requirements of the generator in our case are not as complex
that it would necessarily be beneficial to apply such methods. If we want to generate levels in regards to something, for instance, optimizing a certain measurable property of a level like the amount of motion a level generates or the difficulty of a level, one could benefit from using search-based methods since it is not obvious what to explore nor how to find these solutions yourself.

The triumphing approach in mind is a constructive generation approach, divided into several steps and data structures to limit the amount of randomness. The approach is similar to the one applying software agents acting like erosion of nature on a blank canvas of terrain. Unlike in 3D, starting with an empty plane, where the agents could first start of by generating a height map and maybe draw a coastline, we start by generating an interesting starting level layout in 2D, which is the base platforms upon which the player will walk. Rather than simulating erosion, we will use agents in steps that works with the level, alters it and adding game objects and components to it in different ways. A rough overview of the steps are presented below.

The algorithm

The algorithm produced is divided into several steps:

1. Rough low resolution overall layout (General)
2. Concrete plain base blocks (Base level)
3. Add pitfalls (Carve base level)
4. Adding game objects and decorations (Populate)

Rough low resolution overall layout

In order to produce valid levels, meaning levels that is desirable or playable, the algorithm first reduce the endless possibilities of a tile placements by generating a rough layout that represents different larger blocks of the level. You can view a block as a two-dimensional collection of 1x1 tiles of specified block width and height. The blocks are of enum types NORTH, SOUTH, EAST and WEST and this step only requires a desired number of blocks, and a probability of changing the layout direction.

It starts with a coin flip of which direction the level will go first, EAST or WEST. After that the algorithm generates a new random direction that satisfy some rules, for instance, that it does not collide with any previous layout block nor disrupting a path that needs to stay free. In Figure 3.1 the green square marks the starting point in this example of the algorithms step.

The layout collision is determined by translating the list of enum directions to a two-dimensional grid of integers, performing more primitive calculations on that representation.

![Figure 3.1: Level layout - list of enums visualized](image)
The layout could also be seen as a 2D array of ints if that helps the reader, although the list of enums is used in the actual algorithm.

```
0 0 0 0 0 0 0 0
0 0 0 1 4 0 0
0 1 1 3 0 1 4 0
0 0 0 0 4 2 2 0
0 0 2 2 2 0 0 0
0 0 0 0 0 0 0 0
```

Figure 3.2: Level layout - visualized as 2d array

The layout collision check is performed by translating the list of enums into a 2D array that marks the WEST and EAST going platform blocks to ones. The twos and threes marks the translation areas between NORTH and EAST going blocks. The marking of these leads to knowing what blocks should stay free. In this representation they could’ve been ones as well, but different numbers are chosen in case we need that information available later. The above examples, Figure 3.1 would be translated into the 2d grid shown in Figure 3.3.

```
0 0 0 0 0 0 0 0
0 0 0 2 1 3 0 0
0 1 1 1 0 1 3 0
0 0 0 0 3 1 1 0
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
```

Figure 3.3: Grid used for checking for collision

Concrete plain base blocks

The next step generates bigger blocks of specified width and height which are the main holder of detailed information. It contains information about everything from where the path is located, and the rest of platform tiles to decoration positions. Most of the information stored are different arrays of positions for the game components and tiles. First, a path is generated with a starting position. Similar to the layout generation, the tile to be placed next in the path have a probability to change in height, clamped between a minimum and maximum value of y. When the path has been generated through the block, it is filled with tiles underneath the path. In addition to this, it generates passages upwards if the blocks direction is north-going. The 2D creator tools allows you to skip the part of storing the exact type of tile you want. The tile used are rule based tiles where the sprite is decided dependent on rules you specify. For instance, the rule of when there is a tile above, and to the right, but nowhere else, the rule tile sets that tile as a left down corner.
3.2. The level generator

Add pitfalls

This step assumes that we have a list of blocks to work with, and will, with a certain probability add pitfalls to the blocks by a given minimum and maximum width of the pitfalls. It identifies possible positions where it can remove tile positions to create a drop, that fulfills the minimum and maximum constraints of a pitfall.

Adding game objects and decorations

Here is where much of Unity's engine come into great use. The information stored in the list of blocks are still very primitive. There is no need to store for instance more than a decor position and what type, as in "TREE". Because the use of the 2D creator tools allows you to create brushes that draws different game objects with a probability you can alter. All you need to instantiate different trees, as in this example, is its positions and the fact that you want to draw a tree and the variation is handled by the game object brush. The algorithm now searches, given some minimum and maximum width constraints, after suitable positions for obstacles like spikes, spawn points for marching enemies and checkpoints etc. Adding them with a specified probability.
3.2. The level generator

Figure 3.7: A single block generated with added decor

The editor

Unity has great support for making the life of a game developer easier. The game engine allows for creation of own editor tools. You are free to create windows and panels for the editor to which you can attach scripts for generating things like assetfiles or creating other game objects like different kinds of objects. You can really make the tools do anything within the frames of Unity’s game engine. Ultimately the level generator algorithm is connected to such a EditorWindow.

This tool has the level generator script attached to it where you can alter the parameters in the window and execute the algorithm. The level generator script only generates the data of which a level is composed of. The editor tool generates a Unity prefab from the data and stores it as Assets to the project.

The level data is generated and nicely packed, how do we turn this data into an actual level scene in Unity? In order to draw the levels to the active scene in Unity a SceneInstantiator component was created. It holds Unity specific types such as prefabs or brushes that in turn represents the actual game objects and components of the complete level. It holds what grid to draw to, parent objects under which to instantiate the different kinds of objects. The parent objects main purpose is order, to keep the scene hierarchy as clean as possible. The SceneInstantiator take as input the prefab output from the level generator editor. The grid mentioned earlier is what holds the tilebased components and features of some of the new 2D creator tools. When the level has been instantiated in Unity, the designer can view exactly how the level will be viewed and played in the final game. The designer is free to change anything about the level. Remove or add tiles in the editor, add or remove objects, ultimately resulting in a tailor made level. After modifications you use the tool to save the final level as a new final prefab that can simply be placed in any game, or any scene.

Unity and relevant tools

Unity is a very popular game engine developed by Unity Technologies that is available for anyone who wants to practice their game development skills. Unity was originally released in June 2005 and was at the time a OS X-exclusive game engine, according to an interview of John Riccitiello, CEO of Unity Technologies, by Dean Takahashi from VentureBeat. It does now support 27 platforms and games can be developed for both 2D and 3D, to everything from mobile phones to smart TVs. 

14
Unitys licensing model makes the game engine available for everyone. In short, they have different brackets of monthly payments, categorized on how much revenue the game brings. For instance, all games that have a revenue of under 100 000 USD have a free licence, but with some limitations and unavailable services. This allows for many small studios or indie game developers to develop their ideas to games with a smaller cost, strengthening their position on the market, making them more capable of competing with the big game producing companies.

The Unity editor allows the developer to create custom editor windows that can aid the developer tremendously. The initial thought is to utilize this functionality when creating the final level generator, making it possible to generate levels with simply using the tool in the editor window.

Unity also lets the developer to store created components, that makes it easy to store a complete game object with other attached components and behavioral scripts. The idea is to store, among other things, the motion controller as a prefabricated game object in Unity. Making it as simple as a drag and drop to include the motion controller in other projects.

Relevant 2D tools

The game is to be developed as a 2D platformer together with the support of the Unity engine and other relevant 2D tools. Tools such as the Tilemap, allows you to define brushes of 1x1 tiles or select several tiles and draw them directly in the scene, attached to a mandatory grid. You simply define the objects and the brushes, point, click and paint with desired objects and sprites. Another neat thing is that the tiles that you draw can have rules attached to them, called RuleTile. A RuleTile can be programmed to adjust the sprite or gameobject of the newly placed tile depending on the surrounding tiles. This will allow us to generate and store only primitive data, for the generator, as we outsource the responsibility of what kind of tile or game object should be placed to the elegant tile and brush-functionality provided by Unity. The colliders for the tiles are also calculated and merged to one by Unity if you choose to use the Composite Collider. For instance, consider a 2x2 block of tiles, instead of having a collider for each of the single tile, this block will have one big collider surrounding the whole block of the four tiles, and will also be adjusted according to the form of the sprite.

3.3 The motion controller

LiU Active lab have developed a method of web camera interaction which is based on simplicity. All that is needed is a camera, enabling the interaction to be used on many different devices. Earlier the method was implemented in WebGL since most of the previous games where developed for the web browser. Now the equivalent method was to be implemented in Unitys own shader language. The shader language is written, according to Unitys official documentation, in a variant of the HLSL language, also called Cg, but for most practical uses the two are the same.

According to Unitys official documentation it is recommended to only use raw GLSL for testing or if you know the target machines inhabit Mac OS X, OpenGL ES mobile devices or Linux. It is not clearly stated why. Even though it is possible to run OpenGL code in Unity shaders, another reason why the motion controller was ported to HLSL instead was because Unity cross-compile HLSL into optimized GLSL code when needed anyway.

The basic idea of the motion controller is to read the web camera texture and run it through a couple of passes in the shaders before reading and using the final produced texture for the interaction. In total there are four passes through four different shaders.

1. Luminosity pass
   The luminosity pass is the first pass for the web camera texture. It makes the image brighter, for further processing.
2. Gaussian blur Y pass
   Gaussian blur applied to the image in y-axis.

3. Gaussian blur X pass
   Gaussian blur applied to the image in x-axis.

4. Frame difference pass
   Lastly, the final texture is run through a frame difference pass where the last frame and the current are compared pixel to pixel. The distance of the pixels are calculated and compared to a threshold deciding if the final pixel should be drawn as white, or as transparent.

Figure 3.8: Output texture, black pixels are transparent

The web camera texture is processed each frame and the final output is written to a texture in the game scene as an overlay on top of the whole game scene, mirroring the players movement as shown in figure below, making the interaction transparent for the player. The transparency is of crucial utility, it allows the player to get instant feedback of how the player is being perceived by the game. The final texture can then be read and used by the motion controller.

3.4 Measurement of movement

As mentioned above, the interaction is controlled via a final produced texture of pixels either colored white or completely transparent. Ideally, the pixels that are white are pixels that got triggered by intentional motion. Given that, we have a somewhat true measure of movement at our disposal already. If we count the number of white pixels for a frame, we get a value corresponding to how much movement that was generated on that frame. But as frames are quickly generated, there will be a great deal of frames that don’t provide any information, that are simply empty, or transparent. We don’t want to process and visualize unnecessary information. It would be a well suited option to have the ability to choose between capturing each frame, or by a sample time.

Total movement

With the 2D textures of motion at our disposal, we can iterate over all the captured frames and summarize the total pixel count of the triggered pixels as the example in Figure 3.9.
we then normalize the final 2D array with the pixel count to floats between zero and one as shown in Figure 3.10, we can generate a sort of heat map, that shows the overall human player movement produced while playing the game.

Figure 3.9: Calculating total movement method

| 0.32 | 0.54 | 0.67 | 0.33 | 0.15 |
| 0.62 | 0.66 | 1.00 | 0.39 | 0.57 |
| 0.24 | 0.39 | 0.94 | 0.82 | 0.83 |
| 0.61 | 0.86 | 0.68 | 0.64 | 0.52 |
| 0.87 | 0.78 | 0.32 | 0.19 | 0.16 |

Figure 3.10: Normalizing total movement grid

The normalized values from the values in Figure 3.9 in the normalized grid corresponds to the alpha value the pixel at that position will have in the resulting total movement heat map. An example of a final heat map for one frame can be seen below in Figure 3.11.

Figure 3.11: Heat map example

**Motion per frame**

The total movement metric will just track how much the player moved in total across the whole session of the level, regardless of how long the player took to finish the level. We could
use another metric that measures a rate of motion. As an addition to the total movement metric, we will calculate a motion per frame metric by dividing that number with how many frames the game were played. This metric could reveal any overall changes in motion behavior as the player becomes used to the interaction.

Motion intensity over 2D level

One of the more important measurements of this report is how we can utilize the methods described above to analyze what different level compositions generates in human motion. And how to visualize this in a comprehensible manner. This is yet another case where we benefit from the power of the Unity editor that supports line segments and different standard graphics that we can use to visualize the intensity. The idea is to track the player avatars position in the game at the same rate as we capture the motion textures. By doing that we can calculate a number of motion intensity for a captured frame, and link it to the position the player avatar had on that frame. This results in a mapping between the player avatars position and how much motion was generated by the human when interacting with the game.

This data can then be read and visualized immediately in the editor via a written editor script. But how to visualize this in the best possible way within Unity? The data can be read from elsewhere and plotted in for instance matlab, but the goal here is to map the motion intensity directly on the level. Using the line tool in Unity made it hard to scale with the motion intensity, and when the player respawned the line was drawn from where the player died and to the spawn point, cluttering the scene.

![Figure 3.12: Line motion intensity visualization example](image)

A better idea is the use of circles spawned at the players positions with matching size dependent on the given motion intensity. The problem is still that the scene gets cluttered with alot of data that is hard to distinguish from one another. This is especially true if the player dies and respawns several times. It is worth noting that the data in this example is captured every 0.025 seconds and not by every frame. That is why you can see some irregular spots of empty spaces where there logically would be some motion. This is however not important for the point of visualization technique.
3.5 Method of data gathering

To fix that problem, I also store the current life number to the data, and depending on that number, draw the circles with a different color.

In this particular example where the player has died multiple times involving difficult jumps, the data can still get a little cluttered, but works great in most cases.

3.5 Method of data gathering

There are a lot of variables that can make unwanted variations to the test of motion intensity measurement. So in order to minimize the effect of outside change that could have an affect on the data we need to recognize the related variables. As a quick recap of the motion controller,
it triggers for pixel changes between the current image of the web camera stream and the previous. Thus we could immediately identify that changes to how close you are standing to the camera, lighting in the room, background, what texture and color your sweater has, as well as how similar the sweater is to the background has a direct effect on the motion controller and need to be kept as static as possibly during a play session. Therefore, every data sample will be performed at the same spot, with the same background and lighting. All samples per person are gathered in one session of roughly 15 to 40 minutes. This will also keep the players clothes unchanged during the session, which is of particular importance since that variable is the one that leads to the most difference in motion intensity. If the shirt is plain colored compared to a shirt with a lot of textures, the motion intensity varies greatly from those samples. The parameters of the web camera that decides the sensitivity of the interaction is configured, if needed, to fit the person playing, before data is gathered and is then kept static for each person throughout all the runs. The data is captured every frame for the most precise measurement.

### Iterative process

To gather data for the motion analysis, the first thing is to find appropriate parameters for the level generator that will produce the test levels. The generator generates enough variation from the same set of parameters that we don’t need to change these once they are set. By doing so, it might also simplify the detection of patterns or similarities in the motion data over similar level segments. The overview of said process is shown in Figure 3.15.

Firstly some starting parameters are set for the generator, five levels were generated and evaluated on beforehand if the levels generated would suit both experienced players and beginners. The goal was to find parameters that generated levels rather easy to finish, but that could be a challenge to finish quickly. One level were chosen by random and tested and practically evaluated by the experienced player with the same goal in mind as mentioned earlier. The parameters were updated according to the evaluation from the experienced player.

![Figure 3.15: Overview of the iterative process](image)

**Update generator parameters**

The first iteration, some relatively random parameters were set for the level generator. The choice of new parameters were a combination of the overall analysis of the pattern of the levels generated and how the chosen test level in the last step were perceived. The goal was to find parameters that generated levels that would suit both experienced and non-experienced players. More specifically, the levels should be rather easy to complete, but challenging to complete quickly. The number of blocks, block width and height were already decided and kept to twenty and ten by ten respectively, and the parameters not yet decided were mainly
3.5. Method of data gathering

the probability parameters. Figure 3.16 shows the resulting choice of parameters for the level generator.

![Base level settings table]

**Generate test levels**

When new parameters were set, five levels were generated where one were chosen for the next step.

**Test generated levels**

The experienced player made an overall evaluation by first making a judgment of the levels by walking them through without playing them. Then one level of the generated test levels were chosen by random to be played by the experienced player. The overall judgment of what parameters should be tweaked for the first step were a combination of the overall assessment and how the test level was perceived.

**The Levels**

The three levels chosen for conducting user tests can be seen in Figure 3.17, 3.18, 3.19. We wanted both left and right going levels to see if the patterns that may occur are similar in each direction. The levels generated are challenging for beginners and also for experienced players striving to reach the highest possible score. The levels generated with the chosen parameters had nice timing between jumps, obstacles and the time gating moving platforms, a variation of parts that were easy to run through and parts that were much more difficult. The perceived difficult parts were less frequent than the easy parts.
3.5. Method of data gathering

User tests

User tests were conducted to gather motion data that would be used in analysis of what elements in the level produces in terms of motion intensity. This is especially important to see if discovered patterns is true for more than one player.
3.5. Method of data gathering

The three auto generated levels will be played twenty times each by the experienced player and the motion frame data for each run is stored. The runs are to be played in sequence from run one to run twenty with as little interruption as possible. This hopefully will keep the surrounding variables of the environment as static as possible as well as keeping the level and interaction control fresh in memory of the test players, ultimately keeping the noise of the data down.

Four more people with little experience will play the first out of those three levels also twenty times each. The four new players all play the same level and the same motion frame data is stored for analysis.

The levels were designed to take around 40 seconds up to 2 minutes, which will lead each player to have test sessions of around 15 to 40 minutes. The test users are each told how you get the best score, that time is more important than killing the enemies, but if you could kill the enemies without loosing much speed, that will lead to a higher score.

Five men between the ages of 23 to 28, where all have been playing computer games earlier are the testers for this report. However, only one of them can be considered more than a beginner when it comes to motion-based games.
4 Results

This section presents the results obtained in several domains of the project. Briefly about the web camera prefab, the level generator tool utilizing Unity’s 2D creator tools. Last but not least, the motion intensity results gathered from multiple sessions, with both experienced and non-experienced players, is presented.

4.1 The Game

The game produced for analysis is a 2D platformer where the objective is to travel through the level as quickly as possible, neutralizing enemies and avoiding other obstacles. The sprites are borrowed from an official 2D platformer project made by Unity. To the left and right you can see the interaction boxes, that you touch with your hands, tentatively, to generate movement for the player. The texture displayed on top of the game is the feedback texture of how the game interaction perceives the player. To generate a jump in the game, the requirement is that there is a lot of movement in the middle of the screen and the simplest way to generate a lot of changing pixels in that area is to simulate a real jump, or a quick curtsy might do the trick.
4.2 The level generator

A big part of the motivation behind the level generator being implemented in Unity is the possibility of changing looks, function or layout of the level after the level has been generated. Unity’s 2D creator tools is excellent for handling sprites, colliders and custom setups of game components and objects. This allow us to store quite primitive data for the levels and let Unity handle what should be instantiated or drawn to the level. This also leads to very loose coupling between the level representation and the looks of it. Another big part of the motivation were to also make use of the fact that you can create custom editor tools in Unity. Figure 4.2 present the window tool that is connected to the level generator algorithm.
4.2. The level generator

The meaning of each variable is self-explanatory in most cases, but they are as follows.

- **Start position**: The first position of the level path.
- **Number of blocks**: How many level blocks that should be generated.
- **Block height/width**: How many tiles a block should consist of.
- **yMin/yMax**: The maximum height variance for the path.
- **P of new y path**: The probability of randomizing a new y-value for the path generated.
- **P of new direction**: The probability of a new direction for a block in the construction of the level layout.
- **P of pit per block**: The probability that a pit will be carved out of each block.
- **Pit min/max width**: The minimum and maximum width requirement for an area of a pit.
- **Spikes min/max width**: The minimum and maximum width requirement for an area of spikes.
- **Spikes min/max depth**: The minimum and maximum height requirement for an area of spikes as they are placed in hollows.
4.3 The motion controller

- **P of spikes:** The probability that a spike area will be spawned.
- **Enemy area min/max width:** The minimum and maximum width of an enemy area.
- **P of enemies:** The probability that an enemy is spawned on an enemy area.
- **Number of blocks between checkpoints:** How many blocks there should be between checkpoints.
- **P of decor:** The probability of spawning decorations.

After specifying the parameters, the developer presses "Generate level" and a level is generated where the data is stored in a Unity prefab. In our example it is displayed as "Level1" in the field above the button. The developer can then hit "Draw scene" that draws and instantiates all game objects and components from the level data prefab. In order to do that, the tool needs to hold a SceneInstantiator that holds all the prefabs, game objects and brushes that the developer wishes to draw and instantiate the scene with. It is the SceneInstantiators responsibility to provide and instantiate the level from the data generated by the level generator. The developer is then free to make any changes in the editor using brush tools, moving or placing new enemies, spikes or whatever comes to mind. When satisfied, the final level is saved as a new prefab and placed in the build folder of Unity, ready to be played.

4.3 The motion controller

The web camera motion controller is responsible for two main things. It holds the shaders that process the stream of images supplied by the web camera, and manages human interaction with the game. Each frame the shaders process the web camera image and writes the output to a 2D texture as explained earlier. We can read from this texture and apply logic in any way we see fit. An image of the prefab is shown in the figure below.

![Web Cam (Script) in Unity](image_url)

**Figure 4.3:** The web camera prefab in editor

In this game, you control the player via UI interaction boxes. These boxes can be easily changed or swapped. Those are the Left-, Right-, and UpInteraction objects on the bottom of the image, given as a reference to the web camera controller. From the top we have an
interaction sensitivity that can be adjusted to suit the current player, the environment played in, or to adjust for the patterns on your shirt since more distinct patterns results in more pixels being activated in the controller. This variable will lower or heighten the pixel thresholds you can see just beneath the sensitivity slider. The rest of the variables shown in the editor are the variables used in the shaders, made public to be easy to change. The two RenderTextures could be used as one, but have been separated, where one is the texture displayed in the scene to serve the player with feedback, and the other (collisionTexture) is used for the logic. The choice of having two is to enable the possibility of having one fairly detailed texture to be displayed to the player, and one in the background with desired resolution, for instance a lower resolution, to speed up interaction in certain cases. It’s a natural split of textures since the logic should work in parallel with the visual feedback, this adds to the reusability of the web camera prefab for other games as well where the collision accuracy requirement is different.

This package of shaders and controller are implemented as a prefab in Unity. Prefab is short for prefabricated and encapsulates the functionality neatly, also enabling it to be reused in a simple way in Unity in different projects.

4.4 Measurement of movement

The following section will present the results related to motion intensity.

Overview

The players that played the game have all different gaming experiences, although pretty similar experience when it comes to motion based games. Figure 4.4 shows score improvement over times played on the same level, namely level 1. Only the successful runs are displayed in this figure, making it easier to see the improvement over times played.

The rate of level incompletion were generally greater in the beginning. Figure 4.4 includes the failed attempts on level 1 as well as the successful. Although this figure can be used to track players improvement in consistency rather than only with score, it also clutters the graph.

Player 0 is the experienced player, and the rest are beginners, both with the interaction, and the game mechanics.

The colors in the motion intensity figures, where the intensity is mapped to the level stands for what number of lives that has been used. The colors are

2. Yellow: Second life, one death.
3. Cyan: Third life, two deaths.
5. Red: Fifth and last life, four deaths.
4.4. Measurement of movement

Figure 4.4: Level 1: Score improvement over times played excl. failed attempts

Figure 4.5: Level 1: Score improvement over times played incl. failed attempts

**Player 0 - Experienced**

**Total movement and improvement**

This section presents the improvement and the relationship between the total movement throughout the level.
Figure 4.6: Score improvement over times played excl. failed attempts

Figure 4.7: Worst performance total movement image(1 205 576)

Figure 4.8: Best performance total movement image(1 177 609)

Table 4.1: Player 0 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>1 205 576</td>
<td>536</td>
</tr>
<tr>
<td>253</td>
<td>1 177 609</td>
<td>492</td>
</tr>
</tbody>
</table>

Motion intensity on map

This is how the motion intensity were mapped to the actual level played.
4.4. Measurement of movement

Figure 4.9: First run

Figure 4.10: Best run

Player 1 - Inexperienced

Total movement and improvement

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.11: Score improvement over times played excl. failed attempts

Figure 4.12: First run (3 630 025)

Figure 4.13: Second run (3 334 254)

Figure 4.14: Next best run (1 364 354)

Figure 4.15: Best run (1 249 076)

Motion intensity on map
This is how the motion intensity were mapped to the actual level played.
Table 4.2: Player 1 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>3 630 025</td>
<td>410</td>
</tr>
<tr>
<td>53</td>
<td>3 334 254</td>
<td>408</td>
</tr>
<tr>
<td>202</td>
<td>1 364 354</td>
<td>503</td>
</tr>
<tr>
<td>230</td>
<td>1 249 076</td>
<td>520</td>
</tr>
</tbody>
</table>

Figure 4.16: Worst run

Figure 4.17: Best run

Player 2 - Inexperienced

Total movement and improvement

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.18: Score improvement over times played excl. failed attempts

Figure 4.19: Worst run (6 229 934)

Figure 4.20: Next worst run (4 849 190)

Figure 4.21: Next best run (2 552 547)

Figure 4.22: Best run (1 519 336)

Motion intensity on map

This is how the motion intensity were mapped to the actual level played.
4.4. Measurement of movement

Table 4.3: Player 2 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>6 229 934</td>
<td>489</td>
</tr>
<tr>
<td>42</td>
<td>4 849 190</td>
<td>526</td>
</tr>
<tr>
<td>111</td>
<td>2 552 547</td>
<td>569</td>
</tr>
<tr>
<td>152</td>
<td>1 519 336</td>
<td>438</td>
</tr>
</tbody>
</table>

Figure 4.23: Worst run

Figure 4.24: Best run

Player 3

Total movement and improvement

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.25: Score improvement over times played excl. failed attempts

Figure 4.26: First run (5 783 984)
Figure 4.27: Next worst (2 773 430)

Figure 4.28: Next best run (1 673 295)
Figure 4.29: Best run (1 176 747)

Motion intensity on map
This is how the motion intensity were mapped to the actual level played.
4.4. Measurement of movement

Table 4.4: Player 3 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>5 783 984</td>
<td>590</td>
</tr>
<tr>
<td>87</td>
<td>2 773 430</td>
<td>507</td>
</tr>
<tr>
<td>230</td>
<td>1 673 295</td>
<td>723</td>
</tr>
<tr>
<td>230</td>
<td>1 176 747</td>
<td>492</td>
</tr>
</tbody>
</table>

Figure 4.30: First run

Figure 4.31: Best run

**Player 4**

**Total movement and improvement**

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.32: Score improvement over times played excl. failed attempts

Figure 4.33: First run (1 612 777)
Figure 4.34: First completed run (1 139 575)

Figure 4.35: Next best run (1 227 902)
Figure 4.36: Best run (1 072 861)
4.4. Measurement of movement

Table 4.5: Player 4 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>1 612 777</td>
<td>363</td>
</tr>
<tr>
<td>57</td>
<td>1 139 575</td>
<td>295</td>
</tr>
<tr>
<td>253</td>
<td>1 227 902</td>
<td>543</td>
</tr>
<tr>
<td>253</td>
<td>1 072 861</td>
<td>477</td>
</tr>
</tbody>
</table>

Motion intensity on map

![Figure 4.37: Next best](image)

![Figure 4.38: Best run](image)

Level 2

Level 2 was only played by the experienced player. The results are as follows.

Total movement and improvement

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.39: Score improvement over times played excl. failed attempts

Figure 4.40: Worst performance (1 865 063)

Figure 4.41: Worst performance (1 073 432)

Figure 4.42: Next best performance (1 347 143)

Figure 4.43: Best performance (1 516 359)

Figure 4.44: Player 0 - Motion intensity on map
4.4. Measurement of movement

Table 4.6: Player 0, Lvl 2 - Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>135</td>
<td>1 865 063</td>
<td>423</td>
</tr>
<tr>
<td>228</td>
<td>1 073 432</td>
<td>398</td>
</tr>
<tr>
<td>285</td>
<td>1 347 143</td>
<td>609</td>
</tr>
<tr>
<td>323</td>
<td>1 516 359</td>
<td>796</td>
</tr>
</tbody>
</table>

Motion intensity on map

Figure 4.45: Worst run

Figure 4.46: Best run

Level 3

Level 3 where only played by the experienced player. The results are as follows.

Total movement and improvement

This section presents the improvement and the relationship between the total movement throughout the level.
4.4. Measurement of movement

Figure 4.47: Score improvement over times played excl. failed attempts

Figure 4.48: Worst performance (2 054 698)

Figure 4.49: Next worst performance (1 987 782)

Figure 4.50: Next best performance (1 558 895)

Figure 4.51: Best performance (1 420 903)
Table 4.7: Player 0, Lvl 3 Score and motion.

<table>
<thead>
<tr>
<th>Score</th>
<th>Total Movement</th>
<th>Motion per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>165</td>
<td>2 054 698</td>
<td>596</td>
</tr>
<tr>
<td>192</td>
<td>1 987 782</td>
<td>622</td>
</tr>
<tr>
<td>250</td>
<td>1 558 895</td>
<td>626</td>
</tr>
<tr>
<td>256</td>
<td>1 420 903</td>
<td>577</td>
</tr>
</tbody>
</table>

Motion intensity on map

Figure 4.52: Worst run

Figure 4.53: Best run
The chapter discusses the method and results regarding the level generator and the measurement of movement.

5.1 Results

The level generator

The level generator resulted in a window tool with the algorithm embedded. The generator successfully generates variated content and can be altered directly in Unity's editor upon generation. Although the generator only generates data, decoupled from actual representation of game objects, the responsibility of much of the variation is left to some of Unity's great tools. For instance in the case of enemies, the level generator algorithm only generates positions for enemy spawns throughout the level, and the SceneInstantiator instantiates enemies with an even random distribution. The brush holds in our test case three different enemies and instantiates them with equal probability. In that sense, the level generator is dependent on a SceneInstantiator in order to get a working representation in Unity. The SceneInstantiator holds the resources or prefabs for the level components. Even if the level generator as a tool is therefore dependent on both the algorithm and the SceneInstantiator to work as a whole, the generation of data and the representation, what kind of objects that should be put in the final level, is independent. In its current state, it doesn’t support any other ways of altering the level content other than within Unity, and is therefore only a tool for Unity. The utility of having the possibility to alter the level outside of Unity is not necessarily an improvement. One could add functionality for storing the level data on disk in another format that could be read by other level editors. The algorithm works well in the editor window and when the SceneInstantiator is setup, it’s very easy to generate a level and draw it to scene for further customization.

Measurement of movement

The interaction worked smoother than expected and what could be seen overall is the score improvement of most of the players. The better the players got, the less they moved in total. This is expected since better score means less playtime, less play time will reasonably lead to
lesser movement in total. It was an amusing observation that all players noted and expressed that they found the game physically stimulating, with sweat running down their backs, as they not only activated the upper body, but the lower body as well many times. This is in line with what was concluded by Elaine Biddiss and Jennifer Irwin in [2], where there was a significant increase in activity level in games that needed the player to involve the lower body in the interaction.

An interesting discovery is that all the inexperienced players had a decline in total movement as they got better at the game and their score increased, while the opposite were true for the experienced player. A likely reason for this is that the experienced player already has his interaction well adapted and settled for the game, and can therefore explore the level without having to learn how to control the avatar. The experienced player is therefore allowed to focus and react to the level instead of learning the controls of the game and the avatar. Eventually when the experienced player had made the level related mistakes, he could focus on finishing the level with speed to get a higher score. In his case initially, that meant moving more in order to keep the avatar moving at all times. At the last couple of runs you can see that the total motion intensity stabilizes a bit more towards a optimal value for that player in those conditions. In Figure 4.5 where the samples included failed levels, you can also see that the experienced player had a higher consistency and score than most of the inexperienced players from the beginning, showing that he completed the level in all the runs.

The motion intensity per frame do not seem to have any obvious patterns among the inexperienced players. Meaning that the player moves with similar amount of movement throughout the samples. However, the score improves in most of the cases, which could mean that although the amount of movement is roughly the same, the precision is not, and the player might get better at finding productive movement that moves the avatar as the player want. It could also mean that the player gets better at the actual level, instead of getting better at controlling the avatar. Likely, a little of both is true. However, it could be argued that if you were to draw a straight line that would fit the sample data for the motion per frame the best, the line for each player would increase from the beginning to the end. Looking at the patterns of the experienced player and Player 4 that beat the level with the best score, the motion intensity per frame slightly increases as the score increases. The reason for this can be argued along the same lines as why the total movement increases in the same way for the experienced player, discussed earlier. Why it does not for the inexperienced players might be that they are still in the process of refining their interaction technique and may, after many more samples, follow the same development in motion analysis as the experienced player.

If you look at Figure 4.6 and Figure 4.32 you can argue that there is a small relation between higher motion per frame and score for Player 0 (experienced) and Player 4. Player 4 learned the interaction quicker than the other inexperienced players and were really going for an optimal speed run of the level. This may indicate that when the players have gotten used to the interaction and the game, when trying harder and really try to push the limits of you current ability, you move more. Likely to ensure that the avatar is at all times moving at full speed, and that a intended jump is triggered for sure.

The pattern continues in Level 2 as you can see in Figure 4.39 where the motion intensity per frame increases quite significantly as the score increases. The player is likely cautious and careful while learning the level at first but later trying to move as quickly as possible. For the last level, the pattern previously seen is discontinued, as can be seen in Figure 4.47. This level was not as smooth as the other two, here the player was forced to cancel some momentum in order to hit enemies and it was perceived that while learning the timing of the level, the player often had to overcompensate left and right in order to hit the enemies. This can explain the broken pattern. It could also be the case that the true pattern is this, as the player gets better at the game or level, the less he has to move. More data and more analysis would be required in order to make a decent conclusion of why this is.

As expected, most motion were generated when encountering some obstacle that required the player to jump, and even more if the player had to jump and move to the left or the right
at the same time. This is especially true when the player had a short acceleration space before a big jump. The intensity got concentrated in those areas. This can be seen in the last tricky elevation section in level 1 Figure 3.17 where the player has to make big jumps from small areas of acceleration and then decelerate to change direction. This was the hardest part for all the players, it required both timing, good control of the movement and some coordination since you had to decelerate quite quickly after jumping up on the third block in order not to fall down again. The spikes under the first block were the cause of death for all inexperienced players at least once and likely added an extra stress since it could be fatal if you would mistime the jump from the second block up to the third. Those spikes underneath were the cause of many deaths even for when the players jumped to the first block. What happened were that the players jump a little to far and tried to compensate back to the left, missing the first block and ended up falling to their death onto the spikes.

This can be translated to, levels generated with high probabilities of changing the path in the y-axis, high probability of obstacles, pits, and enemies will naturally lead to more jumping interactions that in turn generates more human motion. This answers the research question, but without considering if the level would be playable. We can then tweak the parameters of the generator to produce levels with a lot more required motion intensity. However, by doing so we will sacrifice momentum and might disrupt the important feeling of progress the players feel by simply running at all times, as pointed out by Yannakakis et al. [10]. More research could be done to find the appropriate balance between how large the amount of movement should be and where the levels generated would start to be perceived as non-enjoyable or too difficult and physically demanding. Perhaps it could be combined with Anderssons and Classons work of generating levels with a predictable difficulty [3]. You could possibly even add to their fitness function a desired amount of movement for the level. This thesis and the motion intensity tool that maps the motion intensity could be the foundation of creating a table of what components or entities in the level generates in terms of motion intensity.

Level 1 were already physically challenging, especially if played with the goal of beating the high score. More forced jumping interactions might interrupt the flow of the level as well as being to physically demanding. This might result in a non-enjoyable level filled with frustration and exhaustion. Level 1 turned out to be a very fun level, and it made the players sweat. Why? First, throughout the level, the required motion intensity varied, but you could keep a constant movement towards the goal if you had learned the game relatively well. Looking at for instance 4.10 you can see that there are three sections of the level that generates more movement than the rest of the map. The first part is where you get on the moving platform and have to generate movement before you jump off it in order to jump far enough to kill the enemy waiting above and keep the momentum needed to reach the moving platform in time. This is followed by running and timed jumps with little challenge. The mid part is the other section with a moving platform where you also have to generate some movement and jump early to time the next jump over the spikes and also land on the checkpoint. This is followed by immediately jumping again to clear the next set of spikes and the last elevation. After clearing that part, there is again some travel time where you can recharge your focus before the last but not least trickiest section where you need momentum, timed jumps and direction changes as quickly as possible. After this, you just run and time the last jump. This variation turned out to be physically challenging enough while the players got to enjoy the momentum mixed together with tricky parts where you really had to step up your performance.

The pattern of a jump can easily be recognized with the motion intensity mapped to the levels. The typical jump is a fast knee bend, and the where the player jump in the map, you can spot that the jumps are two blobs of increasing and decreasing motion intensity. The first blob is where the players start to accelerate his or her body downwards to trigger motion. It is then followed by some empty frames, corresponding to the turning point of the players knee jerk where they are starting to move upwards again to the original posture.

You can also see by the total movement images how the motion is distributed. The level played tilts the motion heavily to one side. If one were to add coins that the player had to jump
up to in the opposite direction of the level, you could get a more even motion distribution.
You could also generate levels that goes both right and left by having levels that grows more
upwards and downwards every now and then, to get the same result.

5.2 Method

The level generator

The initial thought were more heavily tilted towards creating a genetic algorithm that could
generate levels with variation, while maintaining large amount of movement for the player
playing the game. It was discovered quite early that the genetic approach was unnecessary
with regards to the requirements. It could be done but the suitability was not obvious since you
need to specify, in our case, what amount of movement we want. We did not want the generator
to be controlled with that amount, nor did we know what value would be appropriate. The
variation requirement of the generator could more easily be met with an algorithm written
from scratch with some specified randomness, it seemed like an overly complicated approach
to fit the problem to a genetic approach. The end result would be very similar, yet writing the
algorithm from scratch in Unity, and make the variables public in the editor allows for more
control over specific settings, like more or less enemies, how bumpy the path should be for the
player. All those kinds of variables are now isolated and can be tweaked independently. The
choice of method seemed therefore quite logical and the focus shifted more towards utilizing
Unity’s 2D creator tools and combine these with Unity’s support for creating custom editor
tools. The level generator is completely decoupled from how the data is turned into an actual
level and game objects in the scene. The level generator tool however needs a reference to a
SceneInstantiator in order to instantiate the level in the editor window. The SceneInstantiator
needs to be present in the Scene of the editor and it needs to have references to all prefabs
of game the objects the level is composed of in order to successfully instantiate the level. As
of now the level generator tool is divided into steps one could say. You setup the wanted
parameters and generate the data into a prefab that holds the level data for the level, if
the SceneInstantiator is setup, you can then draw the level from the recently generated data
prefab, or any other data prefab, to the scene to have a look at it. In the scene you can alter
the level as you please and then save the final level that will be included in the game. With or
without wanting to change the final level, you still need to save the final level that have been
instantiated to the scene. The old data prefab is not changed nor deleted.

Perhaps one would have wanted less steps for the tool, and to have the SceneInstantiator
more invisible and coupled to the level generator tool instead of a object in the scene that the
level generator tool holds a reference to. Also, the level designer maybe would have wanted
to draw only the base level, without decorations, since they may not have been designed yet.
As of now, there are no option to exclude certain game components, and all game components
are needed for the SceneInstantiator to be able to draw.

Measurement of movement

Given the method of the motion controller, the logical way of measuring the amount of move-
ment was to use the already written to texture. The measurement is therefore an exact
mirroring of what the player sees as the interaction texture, as well as what the computer
reads from to perform the calculations for the interactions. We perform the measurements on
the same texture that both the game and the human player sees, which could be considered
as quite a true metric.

It can be problematic to compare small local segments of levels and motion between differ-
ent players and draw reasonable conclusions. The amount of movement is dependent on the
environment and what kind of clothes the player has when playing the game. A stripy shirt
would trigger many times more motion than a shirt with a single color. The value in compar-
ison lays therefore in seeing a motion pattern over bigger segments where one can establish some overall patterns that seem true for all players, for instance that the jump segments leads to an increase in motion intensity in all cases, that the motion pattern of the jump interaction looks very similar, and that the segments of the level activates the physical movement of the players in a similar manner.

Another problem is that the score and the total movement measurement is somewhat related to each other. If the player dies, he spawns at the last checkpoint and has to run a part of the level once again. This will definitely add to more movement for the run through and will yield in obvious results that the overall motion increased, as well as the score. That is why the measurement of movement per frame was added, bring forth another measure of how the overall motion changes. It aims to measure how the intensity changes across score improvements and to see if changes as the players get better at the game. This was the original goal of the total movement measure, but it falls short if the player dies and has to run through same segments again during the same sample. One downside with this measurement is now that if a player dies during the play through, the web cam interaction is paused for 2 seconds upon respawning, which can skew the data, making the value drop a little more than if the map is completed without dying. Thus, as the score drops, the motion per frame will also get a small extra drop. Without the delay upon respawning, this would not be the case. It is still necessary since if you’d spawn immediately you could be in big trouble where you are still trying to move the avatar in a direction before the avatar died, that would now lead to yet another death when respawned. This is believed to be of little matter, but should be mentioned.

As the difference of clothes and appearance for the players meant that the settings and sensitivity of the motion controller had to be tweaked. The goal were to tweak the sensitivity so it appeared that the same amount of movement between the players triggered roughly the same behavior of the avatar. This was done subjectively and may contribute to some data offset for the measurement of movement since some players might need to move less or more than others to trigger movement. For instance, some players may had to bend their knees more and with more explosiveness in order to trigger a jump than the next one. This will not hinder the ability to discover patterns however in the broader sense.

Twenty samples for beginners may also be to little to discover more interesting patterns on, since most had no previous experience with this type of interaction, making the first samples or maybe all of them, still contain variables that could be less significant when the players had gotten more used to the interaction. Most of the beginners where still in the process of learning the interaction when they obtained the final score, and the total movement measure where still on the decrease. Perhaps it would then again increase, as it did for the experienced player, as the interaction is no longer a problem, and they too could focus even more on optimizing the way through the level.

Another thing that would have been really interesting is to have a bigger sample size of people playing the game. Maybe some pattern would stabilize or some new would emerge. A sample size of 100 persons for instance that played for more than 50 times each could have brought many interesting findings to light. It would certainly strengthen the research since five persons testing the game only really scratches the surface. It is not certain that the patterns are really appearing, it could be just a fluke. I suspect that some of the patterns might be true, but we need a bigger sample size to conclude that it is likely to be so. Different age groups could have also been of interest when testing the interaction. For instance, a group consisting of children up to 12 years old would probably produce very different results in terms of movement. My initial guess is that the total movement in general would go up, more exaggerated jump interactions, and the improvement consistency would probably be less stable. It would most likely be harder to see the pattern findings emphasized in this report.

The interesting way of visualizing the motion intensity in the editor was better than expected when using circles and can be used in the future to analyze more level composites and scenarios. One drawback is that it works best when the levels are completed with one
5.3 The work in a wider context

run through, without dying, as more deaths really clutters the 2D space, making it harder to follow along and to distinguish the order of events. The idea is not complicated and could be replicated by anyone who has a somewhat decent Unity experience. The big problem of replicability of this report is the use of Unity. One has to have an understanding of Unity as a tool, and how to utilize its power in creating own custom editor tools in order to replicate this visualization and the level generator tool.

The measurement of movement is described in detail throughout the report. Although the main thing left out is how the shaders were implemented that produces the texture used for interaction. The first three shaders, luminosity and gaussian shaders are straight forward shaders that could easily be replicated. The one left a little in the dark is the final difference shader. The details are left out but the main concept should be understood.

Source criticism

The method brought forward by Parberry and Doran in [5], where the software agents you let loose form the world much simulating erosion and forces of nature, could be very suitable for a 3D world where one would like realistic results, it needs some modification for the 2D case, and more specifically, in our 2D platformer since the same level of detail is not needed nor wanted. One could see the agents as steps instead, where the first major step is to create a suitable canvas, a 3D height map as Parberry and Doran suggests, or in our case, a 2D platform layout with base platforms that forms a walkable basic level. That basic level layout can then be altered with “agents” that for instance, carves out fall pits or holes for the appearance of more variations. Other agents could find potential spawn points of enemies and store these in some suitable way.

As the bigger blocks are generated by chance, they will very unlikely repeat over the course of a level. Tim Ziegenbein refers to an interesting approach in [26], where part of the method is to predefined blocks of tiles and fit these together appropriately to generate functional levels. In our case, we may loose some interesting parts that a more randomized generator would produce. If we leave more of the generation to the human by predefined blocks, the levels are more or less limited to his or hers imagination. This could lead to blind spots in the motion analysis where we loose some interesting combinations of level components. We believe the chance is an important part in our case in order to find patterns or new motion data where we can look into what generates different kinds of motion intensity.

5.3 The work in a wider context

The contribution in this thesis is the motion controller implemented in Unity, the measurement of movement that can be used to analyze what generates movement in different games. In our case it was specialized in visualizing movement over a 2D platformer level. The motion measurement can be used to design motion interaction that promotes movement or certain types of movement. It can be used to analyze parts of levels and what kind of motion they produce. Games in general are played inside, but often sitting down or at least with little bodily activation. By analyzing the motion produced by the motion controlled game, it can be used to tweak the motion controller to fit a specific target group, to encourage exercise or some specific movements. Whether it would be elderly people, physically rehabilitating people, people with some physical disability that have difficulties in regular exercise or any other reason people need to activate themselves more. This thesis adds to the research of the field of motion controlled games and the motion produced in those kinds of games. As the game industry is growing, an alternative for sitting inactively while playing can be of good contribution, whilst it can also be argued that more games and gaming alternatives still leads to an even more inactive lifestyle, pulling people inside rather than letting them play, exercise and explore the outdoors. Elaine Biddiss and Jennifer Irwin concludes in [2] however that motion controlled games is not a good substitute for real exercise and traditional physical activities that an
active lifestyle entails as it seems to be very important in sustaining weight loss and overall fitness. They also make the case for motion based games as a good alternative and an extra opportunity for indoor physical activity as we spend more time indoors.

Automatically generated game content can, in the future, replace many positions of level designers. Although our implementation still benefits from a human designer to overlook the level design and alter it to subjective perfection. The field of procedurally content generation may still lead to a lot of abandoned level designers. This thesis don’t really cover new ground within that subject, but shows how it can be combined with the power of Unity. Which can of course lead to more optimized level generations and ultimately still reduce the need of level designers. The positive side is that procedurally content generation of different sorts can strengthen small studios of production, making the market even more competitive.
In the project associated with this thesis, a level generator that utilizes relevant 2D tools provided by Unity, and Unity's ability to create custom tools was created. The level generator tool produced provide the user with the possibility to generate variated levels by just changing variables and clicking buttons in the editor.

A motion controller has been implemented with Unity's own shader language given the method LiU Active lab currently use. The motion controller is stored as a prefabricated in Unity and can thus easily be reused with customized settings between different projects.

A new method of visualizing the motion intensity directly on the level is found and can be used for further and deeper analysis in similar or other future projects. Methods of gathering data and generating the heat map is also reusable and added as a add-on optionality to the motion controller.

One of the purposes was to research how to generate levels while maintaining large amount of movement. It has been demonstrated how the motion intensity maps to the levels, and what basic segments in our case that generates most motion intensity. Jumping over big obstacles generated the most in motion intensity, whether it be jumping over spikes or pitfalls, the players were highly motivated to make sure that the avatar jumped, at the same time they were generating motion for the velocity of the avatar sideways. Especially intense were the parts where the player had to jump one direction, and upon landing jump again in the opposite direction.

### 6.1 Future work

The building blocks for analyzing motion patterns and levels are laid, although there is room for much deeper analysis. With the data and visualization, one can see patterns and classify what kind of motion that was produced at one point of the level. With that, one could continue with deeper analysis and start mapping a motion intensity value to a certain game component, object, level layout or sequence of tiles, what it relates to in terms of motion intensity on an average. This could for instance allow one to develop and add a genetic algorithm to the level generator as it is, where the fitness function evaluates how close the level is to the wanted motion intensity that the composed level result in. This could be used to tailor levels towards a specific target group that has different physical conditions and needs in terms of motion and motion intensity.
Bibliography


Nathan Sorenson, Philippe Pasquier, and Steve DiPaola. “A generic approach to challenge modeling for the procedural creation of video game levels”. In: IEEE Transactions on Computational Intelligence and AI in Games 3.3 (2011), pp. 229–244.


