Bachelor Thesis in Computer Science

An Analysis of Platform Game Design
Implementation Categories and Complexity Measurements

Author: Adam Gustafsson
Supervisor: Tobias Andersson Gidlund
Date: 2014-06-23
Course code: 2DV00E, 15 credits
Level: Bachelor

Department of Computer Science
Abstract

This thesis addresses design and development associated problems identified within the platform-game genre. The problem described originates from the fluctuating curve of interest towards the platform-game genre that can be observed between the 1980’s and today. The problem stated in this thesis is that modern platform-game developers may often overlook and –or deprioritize important design and gameplay related components that we find reoccurring in previously popular games within the genre.

This thesis strives to address such problems by decomposing the development process of a platform game into a light framework titled *Implementation categories*. All included categories represent a set of design and development related platform-game components – primarily identified through previous research in the field. In order to create an understanding of each category’s complexity - as well as account for the possibilities to use the categories as a guideline when developing a platform game - a prototype game was developed. The *Implementation categories* included in the prototype was then measured with a set of software complexity metrics. This thesis will motivate and explain the selection of implementation categories, account for the usage of software complexity metrics as well as present a detailed documentation of the prototype development.

The result of this thesis is a thorough presentation of the *Implementation categories* - attached with complexity examples for each category as well as a complete game prototype. The complete results of this thesis will hopefully be of assistance in small-scale, independent or academic game projects in regard of design, decision making, prioritization and time planning.

**Keywords:** game development, game design, platform games, software complexity, framework, analysis, prototype
Acknowledgements

Thank to my project advisor who provided great assistance in narrowing down the thesis amplitude to an understandable and more feasible hypothesis.
Table of Contents

CHAPTER 1: INTRODUCTION 1
  1.1 Background 1
  1.2 Problem formulation 2
  1.3 Hypothesis 2
  1.4 Target group 3
  1.5 Limitations 3
  1.6 Previous research 4
  1.7 Disposition 5

CHAPTER 2: BACKGROUND 6
  2.1 Software complexity 6
  2.2 Development environment 7
    2.2.1 Design pattern 8
    2.2.2 External software 8

CHAPTER 3: METHOD 10
  3.1 Implementation categories 10
    3.1.1 Level components 10
    3.1.2 Avatar mechanics 11
    3.1.3 Level visuals 12
  3.2 Prototyping & Complexity measurement 12
    3.2.1 Prototyping 13
    3.2.2 Measuring of complexity 13
      3.2.2.1 Metric usage 14

CHAPTER 4: RESULT 16
  4.1 Implementation #1 16
    4.1.1 Output 17
    4.1.2 Complexity: Model 18
    4.1.3 Complexity: Implementation categories 18
    4.1.4 Additional Data 19
  4.2 Implementation #2 19
    4.2.1 Output 21
    4.2.2 Complexity: Model 21
    4.2.3 Complexity: Implementation categories 21
    4.2.4 Additional Data 22
  4.3 Implementation #3 22
    4.3.1 Output 23
    4.3.2 Complexity: Model 23
    4.3.3 Complexity: Implementation categories 23
    4.3.4 Additional Data 24
  4.4 Implementation #4 24
    4.4.1 Output 25
    4.4.2 Complexity: View 26
    4.4.3 Complexity: Implementation categories 26
    4.4.4 Additional Data 27

CHAPTER 5: RESULT ANALYSIS 28
  5.1 Observation 28
  5.2 Problem reflection 31
1 Introduction

This chapter will present a general overview of the thesis background and relevance. Furthermore, this chapter will present the thesis problem formulation and hypothesis, intended target group as well as disposition of the upcoming chapters.

1.1 Background

A platform game or platformer is a game where the gameplay revolves heavily around moving and jumping between obstacles and objects associated as platforms. The genre itself cannot fully be considered a standalone genre since it is often combined with other game genres such as Shooters and Role Playing Games.

The platform game genre originated from the arcade scene in the early 1980’s introducing games such as Space Panic (Universal, 1980) and Donkey Kong (Nintendo, 1981) [14].

During the following decade the platform genre produces some of the most influential games to date with titles such as Super Mario Brothers (Nintendo, 1985) - which held the title of being the all-time bestselling game for three decades - Sonic the Hedgehog (Sega, 1991) and Super Mario Brothers 3 (Nintendo, 1993) [14][15].

Though the genre followed the 3D development into the 21st century with games such as Super Mario Sunshine (Nintendo, 2002) and Rayman 3: Hoodlum Havoc (Ubisoft, 2003) the platform genre started to show decreasing popularity in the market. D. Boutros [2] states that the platform genre enjoyed 15% of the video-game market in 1998 in comparison to 2% in 2002. However, Boutros does not blame the genre itself to be the issue at hand and claims:

“*We believe It is not an issue of genre, but an issue of effective design principles of past being forgotten*”

The statement, which was made 2006, might just have been accurate since we today are witnessing a renewed interest in the platform genre with successful games such as New Super Mario Bros 1 and 2 (Nintendo 2006, 2012), Braid (Jonathan Blow, 2008), Donkey Kong Country Returns (Nintendo 2010) and Rayman Origins (Ubisoft, 2011) [27].

It is interesting to observe that the recent recurrence of interest for the platform game genre seems to be parallel with the releases of these games, since most of them essentially are remakes of some of the best-selling 2D platform games from the 1980’s and 1990’s.

This strengthens D. Boutros [2] reasoning and indicates that important design-related components in fact may have been forgotten.
1.2 Problem formulation

This thesis will elaborate on D. Boutros [2] reasoning about the cause of decreasing interest in the platform genre. In fact, the problem formulation of this thesis can for all intents and purposes be considered as equivalent to the conclusions drawn by Boutros.

Accordingly, the problem formulation of this thesis is based on the hypothesis that important design related elements - that might contribute to the success rate of a platformer - may often be overlooked or deprioritized in the development process.

D. Boutros [2] presents the following conclusion which will serve as the main problem in this thesis:

- In modern platform-game development, effective design principles of past is often being overlooked or forgotten.

D. Boutros [2] then suggest that this problem mainly originates from modern developers pursuit of “widening the market” by making their game much easier and thereby more accessible to a larger audience. He then proclaims that this may result in a backfiring effect, producing less successful games that tend to show lack of similarities with previously best-selling and popular platform-games.

It is important to stress, that this thesis will not attempt to solve this rather large and general problem on its own. Instead, this thesis will primarily target developers within the academic domain as well as independent and small scale developers in general. Because of this specific target group the decision was made to include two additional possible underlying reasons for the same problem:

- Lack of knowledge concerning recurrent design-related components and features from previous successful platform games.
- Lack of knowledge in regard to time consumption, complexity and prioritisation of these components during the development process.

Based on the identified problem and the choice of target group, this thesis will focus on the following questions:

- Which important design and development related components should be deferred to when developing a small-scale platform game?
- What is the relation and distribution between these components in regard of software complexity?

1.3 Hypothesis

The strategy to address this problem has been decomposed into two main procedures; one theoretical and one practical.
The goal of the theoretical procedure is to – primarily through previous research – identify a number of important design and development associated components within the genre.

The identified components will then function as an underlay in order to create a light framework for developing a platform game. This framework – titled *Implementation categories* – decomposes the implementation process of a platform game into three main categories; *Level components*, *Avatar mechanics* and *Level Visuals*. These categories will include a set of subcategories – representing the previously identified platform game components.

The framework will then be used in order to develop a small scale platform game prototype. This will be done in furtherance of measuring the software complexity associated with the *Implementation categories* and its subcategories.

By measuring the complexity of each category and subcategory, we wish to create a better understanding of the effort requirements and prioritisation possibilities regarding each category.

### 1.4 Target group

Due to the fact that this study must be considered as rather small-scale the targeted group is first and foremost developers within the academic domain as well as independent and small scale developers in general.

Obviously the thesis aims mostly towards game developers and platform-game developers in particular. However, the thesis may also be of interest in regard of other areas.

For example the thesis speculates and elaborates on general game design and level-design related topics which may be of interest for any individual studying the area of game-design.

I addition the thesis strategy itself - where important design related elements are identified, decomposed into implementation categories, implemented in a prototype and finally measured for complexity - may be of interest and very much applicable in other fields of study.

### 1.5 Limitations

Due to the time limitation for this thesis it must be stressed that the final prototype used for complexity measurement can be considered as rather small-scale. This does not necessarily mean that the data derived from the measurement will not be accurate in regard of more complex projects. However, the possibility of divergence in results depending on the scale of the project must be taken under consideration.

Furthermore, it should be noted that all of the subcategories presented within the *Implementation categories* were not implemented in the final prototype. This matter will be explained in more detail in chapter three.
1.6 Previous research

It is important to stress that this thesis is first and foremost a study of practical sort. The majority of time and effort was put into the development of the game prototype and the measurement of complexity from the different Implementation categories.

However, in order to ensure that no such research had be done already - and above all; in order to be able to identify and analyse important platform game components – it was highly necessary to look deeper into the field of game-design to see what had already been done.

There is plenty to be found within the area of game-design in general. That is, studies that focuses largely on game design in general covering several or all game genres. Salen & Zimmerman [3] presents a large scale theoretical framework for the whole discipline of game design with focus on “creation of meaningful play”. A. Ernests [4][5] Fundamentals of Game Design also presents guidelines for game and level-design in general, discussing elements such as core mechanics, user interface, storytelling and business models.

Björk & Holopainen [8] presents a framework where they break down various types of gameplay into patterns in order to provide a common set of concepts for designers and developers. This method of breaking down gameplay into patterns can be considered somewhat comparable with this thesis strategy of breaking down - design and development related - platform-game components.

The Game Ontology Project presented by Zagal & Mateas [7] proposes a framework for analysing and studying of games in general. The framework attempts to identify the important structural, gameplay related elements of games and their relationships.

However, the research mentioned thus far is mainly presented from a designer’s point of view, most often in a strictly theoretical manner. Though this literature has proven to be useful also for this thesis it does not supply any research regarding platform-games and its core elements specifically.

Research that does address platform games specifically and can be considered as one of the main sources for this thesis is provided in A Framework for Analysis of 2D Platformer by G. Smith et al. [1]. This study is very much genre-specific and presents an analysis of level-related platform game components along with an attempt to provide a common vocabulary for these components. The study also discusses level rhythm and pacing as a result depending on how these game components are put together.

Another genre specific study addressing platform-games is D. Boutros [2], A Detailed Cross-Examination of Yesterday and Today’s Best-Selling Platform Games. Boutros presents an examination of several bestselling platform games covering elements such as visuals, controls, rewards and challenge. The purpose of Boutros examination is to help designers and developer to create an understanding of how and why these games became so popular.
This thesis can be considered as directly related to the work of D. Boutros [2] and G. Smith et al. [1]. Whilst elaborating on Boutros examination of design related elements from previous popular platform games, this thesis will also – to large extent - use the level associated categories provided by G. Smith et al. [1], along with their common vocabulary.

1.7 Disposition

The rest of this thesis will be presented as follows. Chapter 2 includes a basic explanation of software complexity measurement along with a declaration of which measurement methods that will be used within this thesis specifically. The chapter will also describe the environment used when developing the prototype in terms of programming languages, software and design pattern.

Chapter 3 will present the approach and methods used within this thesis in order fulfil the wanted result. The chapter will present how the Implementation categories were created as well as an explanation and motivation for each selected subcategory. The chapter will then describe the practical approach of the thesis including the prototype development and the complexity measurements.

Chapter 4 will present all measured results – in terms of complexity data – derived from the game prototype.

In Chapter 5 we compile and observe the results presented in chapter four as well as draw general conclusions based on the data. The chapter will also include a short discussion regarding the results in relation to the problem formulation.

In Chapter 6 will present a general discussion regarding the thesis and its results as a whole. The chapter then continues with a speculation concerning the thesis validity and usability.

Chapter 7 will present a final conclusion followed by a discussion regarding future research.
2 Background
In order to create a better understanding of the thesis results and how they were generated, this chapter will provide a basic review of software complexity and how it can be measured. Furthermore this chapter will include a declaration of the development environment and software used when programming the prototype game destined for this thesis.

2.1 Software Complexity
Before presenting the complexity data gathered from the prototype implementation it seems justified to clarify the meaning of the concept complexity by describing the procedure of measuring different complexity-associated parts of a software program.

In software development, the measurement of a particular characteristic of a program’s performance, efficiency or quality is often referred to as a metric [17]. L.H Rosenberg et al. [10] summarize some of the most common software attributes targeted for metric measurement as; efficiency, complexity, understandability, reusability and maintainability alt. testability.

Since the primary goal of this study is to create an understanding of the time consumption and general software complexity related to the Implementation categories’, our evaluation will focus primarily on the complexity and understandability attributes. This is due to the fact that this thesis does not strive to evaluate the quality and or reusability of the prototype, but rather to present a generalization of software size and complexity in relation to each category. The metrics utilized within this thesis will address maintainability and reusability also, but the analysis of their data will not be performed as thoroughly.

For this study, the metrics found to be the most relevant in regard of the problem formulation were selected. The selected metrics, including motivation, are:

- **Lines of Code**: Perhaps the most basic of metrics in regard of measuring the actual size of a method or class [13]. The lines of code metric can be measured in a variety of ways [10]. For this thesis however, the lines of code will be represented by non auto-generated code only, excluding all comments and blank spaces as well. The amount of code lines will therefore not represent the exact number of lines in the source code. By excluding all irrelevant lines of code we hope to increase the generalization of the result, making it as general as possible regardless of framework and programming language, as well as create a more accurate understanding of the actual size of each implementation. The motivation for including lines of code is due to the fact that we found it to be the most concrete and straightforward metric in regard of measuring size.

- **Cyclomatic Complexity**: A rather complicated metric designed by Thomas J. McCabe with the purpose of measuring the logical strength of a program
The metric is defined as “measuring the amount of decision logic in a source code function” [19]. In the case of this study this metric will be used to measure the amount of decisions made within a given class or method. The motivation for including this metric is due to the fact that the amount of decisions within a class or method obviously effects the time consumption in regard of programming.

- **Depth of Inheritance:** This metric is defined as “indicates the number of class definitions that extend to the root of the class hierarchy” [17]. L.H Rosenberg et al. [10] underlines that a high depth of inheritance within a class creates a greater design complexity but increase the potential of reuse. In this thesis prototype, since the goal is to look at different parts of implementation separately, we have tried to keep the depth of inheritance at a minimum. Inheritance was implemented only when it would create an obvious design benefit.

- **Class Coupling:** This metric, also known as “coupling between objects”, measures the coupling between a given class and other unique classes. A coupling between two or several classes may for example occur via variables, parameters, return types, method calls, base classes or interfaces [17]. It is our belief that this metric serves well when it comes to create an understanding of the required relations between classes in the prototype.

- **Maintainability Index:** The last metric used in this thesis evaluation presents an index value, representing the ease of code maintainability. This metric actually uses the lines of code and the cyclomatic complexity metrics in order to calculate its index value. Any code exceeding the index value twenty is considered to have good maintainability [19]. This metric is perhaps not as relevant in relation to our evaluation as the previous metrics. However, since we already employ the lines of code and the cyclomatic complexity metrics the choice was made to include this metric as well. Also, the maintainability index may very well be of interest for any individual who desire to look deeper at the structure and programming style of the prototype itself.

Additionally, after each implementation, a notation will be made observing the amount of milliseconds spend updating the applications logics and drawing functionality as well of any change of frames per seconds. The reason for this notation is essentially to avoid any major design flaws throughout the implementations.

### 2.2 Development environment

The goal of this thesis is to present a generalized result, usable regardless of chosen programing language and- or environment. The programing environment used in this study however, is Microsoft XNA Game Studio 4.0 together with the programming language C#. XNA is a toolset for Microsoft’s IDE (integrated
development environment) Visual Studio and enables the possibility to program games for Windows Phone, Xbox 360 and Windows [18].

2.2.1 Design pattern

Before presenting the complexity data it is also important to note that the complexity of software and its amount of code may differ depending on the choice of design pattern.

In this study we practice the design pattern MVC (Model-View-Controller) which is a well-used and rather popular design pattern within software development in general [24]. The basic logic of MVC is to divide the software into three components; Model, View and Controller.

Perhaps the most central component is the Model which contains the software’s rules and logic [25]. In this study the classes within the Model namespace will contain all game logic, rules and behaviour.

The View component handles all generating of the programs output as well as informing the Controller of any user inputs [25]. In this study the classes within the View namespace will handle all graphical and sound rendering, camera handling as well as detecting any user inputs.

The Controller component functions as a link between the Model and View components by whether updating the models state or command the associated view-class to change its presentation [25]. In this study the Controller will be represented by a Master Controller; initializing all game components and a Game Controller; asking the View for any user inputs in form of avatar mechanics whilst updating the Model state associated with the user input.

2.2.2 External software

In 2D game programming the most common approach to store level data is in equally sized tiles [26]. These tiles are then stored in an array or preferably a map file. A map file can be anything from a simple text file with characters representing different level tiles to more advanced formats such as XML (Extensible Markup Language) or Binary based files. Additionally there is several custom made map file formats specifically generated from tile map editors or level editors (external software programs designed for the purpose of level creation).

In order to simplify the process of generating and drawing tiles for the prototype levels, the choice was made to use an external tile map editor named Tiled [20]. This map editor allows the user to, through a simple interface, create maps of chosen size, draw tile layers as well as object layers. The map can then be saved in an XML base format called TMX (Tile Map XML) [28].

To simplify the implementation of collision handling between the avatar and the platforms the choice was also made to implement a third party library called Xtiled [22]. This library is designed to simplify the handling of, and interaction with TMX files.
Though it is possible to design your own map editor, within this study we chose not to. The main focus of this study is to implement and measure the complexity of different platform game components; therefore it is not relevant to also present an implementation of a custom made map editor. Regardless of relevance there are several free - often well designed - tools for this matter.
3 Method

This chapter will provide a detailed coverage of the methods used within this thesis. The realisation of this thesis can be divided into two main procedures; a theoretical analysis including the creation of a light implementation framework and a practical, iterative prototype development including complexity measurements.

3.1 Implementation categories

In order to create a light framework for developing a platform game with the extended goal of measuring the complexity of its different design and development associated categories, we must first identify which these categories are.

Even though this thesis problem formulation underlines that important components often get deprioritized or forgotten - the decision was made to also include more obvious components such as player movement and – or graphics. This decision was motivated by the desire to increase the generalization of our framework, thus including more obvious design and development related components as well.

This thesis elaborates on the platform game components identified by D. Boutros [2] and G. Smith et al. [1] combined.

G. Smith et al. [1] identify several important level components of which five was considered to be of relevance for this thesis and will be used with the same vocabulary. Those components are: Obstacles, Platforms, Movement Aids, Collectable items and Triggers.

This thesis will also elaborate on D. Boutros [2] identification of the game components Controls and Visuals. However in this thesis those two components will be decomposed further as well as slightly redefined.

Based on these already identified platform game components along with additional research in the area [1.6 Previous research] we have divided all the platform game components we found relevant into three main categories: Level components, Avatar mechanics and Level visuals. In this thesis we will refer to these three categories as Implementation categories.

3.1.1 Level components

The first implementation category we chose to title Level components. This category mainly consists of the level-associated components identified by G. Smith et al. [1]. In this thesis we will refer to the components within each implementation category as subcategories. The subcategories within Level components is defined and motivated as follow:

- **Platform** – Any object that the player can move across, jump or navigate on can be defined as a platform. The motivation for including this
component is for obvious reason since it is the very core element of a platform game. Even if platforms are obligatory for a platform game we found it to be of interest to investigate the actual complexity of this functionality since it is probably together with the player mechanics the highest prioritized functionality at the start of any development of a platform game.

- **Obstacles** – Smith et al. [1] defines this component as “any object that is capable of causing damage to the player”. In this thesis this subcategory will include all enemies, projectiles and gaps causing damage or death to the player. The choice of this subcategory could also be considered as rather self-explanatory since obstacles are the main source of challenge to any platform game. However, this subcategory may not always be as prioritized as needed since it often tends to be quite time consuming depending on the amount of enemies etcetera.

- **Movement aids** – Any object that support the players’ movement capacity in addition to the players own mechanics. Examples of such objects are trampolines, ladders and ropes. This subcategory we believe is a typical example of components that tend to be overlooked. Therefore it is of high interest to investigate the actual complexity and time consumption, functionality like this might generate.

- **Triggers** – Another component identified by Smith et al. [1] and defined as “interactive objects that the player can use to alter the stage of the level, or even game rules such as physics”. We believe this subcategory to be another interesting example of functionality that might not always be prioritized.

- **Collectable items** – The last subcategory is defined as any game item that provide some sort of reward. Examples of such items are coins, extra lives and power-ups. The complexity of this subcategory is also considered to be of great interest since collectable items may enhance the gameplay in so many ways yet perhaps it is not given as much attention as it should.

### 3.1.2 Avatar mechanics

The second implementation category we chose to title **Avatar mechanics**. By “avatar” we simply refer to the player character within the game. The reason for this definition is due to the fact that the word “player” may in some studies refer to the person playing the game rather than the in-game character. In this thesis the player will always refer to the actual in-game character and never the person playing the game. Never the less, we choose to use the definition “avatar” in order to avoid any misconception.

The subcategories within this category is defined and motivated as follow:

- **Collision** – This subcategory includes all functionality related to collision between the avatar and the environment including platforms and obstacles.
This subcategory was not found in any previous research, perhaps due to the fact that it is exclusively from a developer’s point of view rather than from the perspective of a designer only. From a developers point of view however, this subcategory is of great importance since this functionality may require a decisive amount of time and effort.

- **Basic mechanics** – Elaborating on D. Boutros [2] discussion regarding the player controls; this subcategory includes the functionality that represents all the basic mechanics of the avatar such as jumping and moving.

- **Additional mechanics** – The last subcategory intend to cover any additional mechanics related to the avatar. The motivation for creating this subcategory derives from the high amount of possibilities regarding the avatar. As mentioned in the introduction a platform game may or may not also include aspects from other game genres such as shooters. With this in mind we draw the conclusion that the avatar mechanics may include a large variation of mechanics such as hitting, shooting and crawling. Accordingly, any avatar mechanics beyond the more traditional mechanics - mentioned in the last subcategory – should be placed here.

### 3.1.3 Level visuals

The third and final implementation category, titled *Level visuals*, includes all graphical and sound related elements. This implementation category is partly inspired by D. Boutros [2] observation of recurring first level graphic themes among several of the bestselling platform games to date. Also, from a developers point of view it would be highly unreasonable not to address the games aesthetic aspects.

In this thesis we are under the assumption that well-designed functionality may not be recognized accordingly if the game visuals do not appeal the individual playing the game. Since the implementation of graphics and sound may be rather time consuming and - depending on the developers artistically potential – sometimes difficult, we find it to be of great interest to investigate the complexity balance between logistic functionality and visual elements.

With these motivations in mind we simply divided this implementation category into two main subcategories:

- **Graphics** – Representing all code directly related to the games graphical appearance.
- **Sound** – Representing all code related to the games sound.

### 3.2 Prototyping and Complexity Measurement

The second main procedure of this thesis method is of pure practical sort. This procedure includes the actual implementation of the *Implementation categories* into a platform game prototype. It also covers all the complexity measurements performed iteratively on the prototype.
3.2.1 Prototyping

In order to measure the software complexity of the Implementation categories and their subcategories, the decision was made to develop a platform game prototype that would include as many of the implementation categories as possible. An additional purpose with the creation of a prototype game was to investigate the possibilities to use the Implementation categories as guidelines when developing a platform game from scratch.

The implementation process was executed iteratively resulting in four iterations. These iterations will be referred to as implementation 1-4 in Chapter 4: Results.

All iterations resulted in the implementation of one or several of the subcategories derived from the Implementation categories. At the beginning of each iteration a decision was made regarding which subcategories to address resulting in the following projection:

- **Implementation 1** - This iteration would cover the subcategories Platforms, Collision and Basic mechanics.
- **Implementation 2** - This iteration would cover the subcategories Obstacles and Collision.
- **Implementation 3** - This iteration would cover the subcategories Movement aid, Collectable items and Triggers.
- **Implementation 4** - This iteration would cover the subcategory Graphics.

As can be noted, two of the subcategories – Additional mechanics and Sound – were excluded. The prototype developed for this thesis must be considered as rather small scale. At the end it would simply not include any Avatar mechanics that could be considered or defined as “additional”. The subcategory Sound would also have to be excluded. This however, was due to the time limitation of this study.

3.2.2 Measuring of complexity

At the end of every iteration, complexity measurements were performed. The complexity data were then organized in relation to the Implementation categories including all subcategories.

The results from each iteration (presented thoroughly in chapter four) were then organized and presented as follow:

- **General overview** – An introductive text will be presented at the beginning of each result documentation. Following the introduction there will be an overview of the implemented classes along with a class diagram. The purpose of presenting a class diagram from each iteration is to create a clearer view of the structural choices made in the implementation.
- **Output presentation** – A screen dump showing the output result from the implementation.
• **Complexity data** – The software complexity measured from each implementation will be presented as follow:
  
  o **Complexity Model/View** – A table showing the complexity data derived from the classes directly related to the subcategories being addressed in the current iteration. The table will include data from the following complexity metrics; maintainability index, depth of inheritance, cyclomatic complexity, class coupling and lines of code.
  
  o **Complexity: Implementation categories** – A table showing the appraisal number of lines of code and cyclomatic complexity associated with the three implementation categories and its subcategories. The table will include data from the following complexity metrics: lines of code and cyclomatic complexity.
  
  o **Additional data** - A table showing the amount of code lines derived from classes not directly associated with the implementation categories. This data can be considered as rather design specific and may differ depending on the choice of design, structure and development environment. This section will also include a table showing the average update time (in milliseconds) for the prototypes game logics and graphical rendering. The table also present the current frames per seconds registered while running the prototype.

3.2.2.1 Metric Usage

The use of the complexity metrics described in 2.2 Complexity measurements will differ depending on what part of the implementation that is being measured. When compiling the complexity data from the implementation categories, the decision was made to examine *lines of code* and *cyclomatic complexity* only.

The *lines of code* metric is our primary metric for measuring software size. Since the size of a method or class also affects the ease of understandability of the code [10], *lines of code* must be considered to be most relevant metric (among those selected within this thesis) in regard of general complexity and time consumption. The *cyclomatic complexity* metric is our primary metric when it comes to measuring the amount of decisions made within a class or method and was therefore considered to be the second most relevant in regard of general complexity and time consumption. Accordingly, the conclusion was drawn that *lines of code* and *cyclomatic complexity* are the most sufficient metrics when it comes to measuring complexity in form of time consumption, actual application size and general complexity.

By limiting the amount of metrics used for this particular part we strive to simplify the compiling of the final results as well as increase the understandability of the results. The metrics excluded - when compiling the complexity data from the implementation categories - were; *maintainability index, depth of inheritance,* and *class coupling,* all of which is more related to testability, efficiency and reusability [10].
All iterations however, will include more detailed measurements using all the metrics described in 2.2 Complexity measurements. All classes directly associated with the Implementation categories were measured separately with all previously defined metrics: maintainability index, depth of inheritance, cyclomatic complexity, class coupling and lines of code.

We hope that the complexity data that will not be as thoroughly analysed in relation to the Implementation categories (maintainability index, depth of inheritance and class coupling) will still serve a purpose by increasing the general usability of this study. In regard of this thesis hypothesis however, the intention is to primarily look at the cyclomatic complexity and lines of code compiled from the Implementation categories and their subcategories.
4 Result

This chapter will present the documented results derived from the prototyping and its four iterations. The four iterations will be referred to as Implementation 1-4.

4.1 Implementation 1 – Platforms, Collision and Basic mechanics

The first prototype implementation resulted in functionality for three subcategories found in the Implementation categories. These subcategories were: platforms, collision and basic mechanics.

Additionally this implementation also includes one property from the subcategory obstacles in form of deadly gaps between platforms. This is due to the fact that deadly areas are essentially just platforms that result in death upon collision.

Further aspects that had to be taken under consideration were the level of complexity concerning a platform. The use of diagonal platforms will increase the software complexity due to extensive collision and the use of mobile platforms will generate additional code in form of platform mechanics.

Due to the time limitation of this study this thesis will present an implementation of moving platform whilst excluding diagonal platforms. The decision was based on the assumption that moving platforms are a more coveted feature than diagonal platforms.

Figure 4.1 shows a class diagram of the first measurable implementation, resulting in the following classes and their basic functionality:

- **Program** - default static XNA class for initializing and running the application.
- **Controller.MasterController** - Main controller-class, initializing the InputHandler, GameController, GameView and GameModel.
- **Controller.GameController** - Updating the GameModel and GameView as well as asking the GameView for any user inputs in form of avatar mechanics such as movement and jumping.
- **View.GameView** - Main view-class initializing and updating the InputHandler and Camera class as well as rendering the level and its components.
- **View.InputHandler** - Checking the keyboard, gamepad and mouse-state for any new user inputs.
- **View.Camera** - Class for handling the avatar camera; following the avatar across the level.
- **Model.GameModel** - Main model-class, initializing and updating the TMXLevel and PlayerLogic class.
- **Model.TMXLevel** - Class for loading and handling of all TMX data (map file data). This class include collision handling regarding all the associated map objects. The class also initialize and updates the MovingPlatform class.
- **Model.MovingPlatform** - Class for updating the movement mechanics of a mobile platform.
- **Model.PlayerLogic** - Functions as a platform class between the Player and TMXLevel classes in order to manage the avatars collision and positioning in regard to the map objects.
- **Model.Player** - Object class inheriting the Unit class, representing the player and its properties.
- **Model.Unit** - Class designed to be inherited by any Unit object with similar properties such as player and enemies.

It is important to emphasise that the reason for the rather large class amount of this first implementation is due to the fact that the basic structure of the software is included. However, it can be noted that the classes directly linked to the actual subcategories *platforms, collision* and *basic mechanics* is: TMXLevel, MovingPlatforms, PlayerLogic, Player and Unit. For the sake of clarity we will handle the view classes and the Game and Master-controller as game structural components and thereby focus on the model-classes related to the actual implementation of the subcategories.

### 4.1.1 Output

Since this implementation did not address the *Level Visuals* category the output simply presents a level with platforms and a rectangular avatar. Figure 4.2 shows
the output after the first implementation. The arrows have been added afterwards in order to clarify the functional mechanics.

![Figure 4.2 Screen dump from the first implementation's output.](image)

### 4.1.2 Complexity: Model

Table 4.1 shows the compiled data of complexity derived from the Model classes directly related to subcategories Platforms, Collision and Basic mechanics. The tool used for measuring this data would by default include the third party library XTiled into the coupling calculations. Since XTiled is an external library added as a reference only, we felt that this would present rather faulty data in comparison to the actual MVC design. Therefore an additional column is added showing the actual coupling relations in the implementation.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Maintainability Index</th>
<th>Depth of inheritance</th>
<th>Cyclomatic Complexity</th>
<th>Class Coupling</th>
<th>Class Coupling with XTiled</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Player</td>
<td>73</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Model</td>
<td>PlayerLogic</td>
<td>72</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>Model</td>
<td>MovingPlatform</td>
<td>63</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>Model</td>
<td>Unit</td>
<td>92</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>Model</td>
<td>TMXLevel</td>
<td>73</td>
<td>1</td>
<td>72</td>
<td>2</td>
<td>21</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 4.1 Model complexity data

### 4.1.3 Complexity: Implementation categories

Table 4.2 shows an appraisal number of lines of code and cyclomatic complexity associated with the three implementation categories and its subcategories. It is notable that two View classes are included in the Basic mechanics column; Camera and InputHandler.
4.1.4 Additional Data

As can be noted from the previous tables no data has yet been presented from the classes not directly associated with our implementation categories. Table 4.3 shows the size of the rest of the implementation. This code can be considered as rather design specific and may differ depending on how the developer chooses to structure the game. As mentioned before however, the view classes, Camera and InputHandler in particular, can be associated with the avatar mechanics.

<table>
<thead>
<tr>
<th>Name Space</th>
<th>Class</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>GameModel</td>
<td>9</td>
</tr>
<tr>
<td>Controller</td>
<td>GameController</td>
<td>20</td>
</tr>
<tr>
<td>Controller</td>
<td>MasterController</td>
<td>26</td>
</tr>
<tr>
<td>View</td>
<td>GameView</td>
<td>16</td>
</tr>
<tr>
<td>View</td>
<td>InputHandler</td>
<td>9</td>
</tr>
<tr>
<td>View</td>
<td>Camera</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.3 Lines of code derived from the remaining application classes

Finally, as Table 4.4 shows, we can observe that the first implementation, as expected, consumes higher amount of update time within the model classes than within the drawing methods. This is expected due to the fact that the implementation did add game logic whilst no graphics are yet being drawn.

<table>
<thead>
<tr>
<th>Project</th>
<th>Avg. Update Game Logic</th>
<th>Avg. Update View</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successor</td>
<td>1.05 ms</td>
<td>0.22 ms</td>
<td>60.00</td>
</tr>
</tbody>
</table>

Table 4.4 Average update time (ms) for game logics and graphical rendering

4.2 Implementation 2 – Obstacles and Collision

The second prototype implementation resulted in functionality mainly for the subcategory *Obstacles*. Since the first implementation included deadly gaps between platforms the remaining main elements of this subcategory are enemies.

This implementation includes three types of enemies with different properties. The three enemy types were named crawler, jumper and shooter. Their belonging properties are, as their name indicates, simple sideways movement, jumping and shooting.
This implementation would also have to include some additional collision functionality in order to create proper interaction between the avatar and the enemies. However, since this additional collision code is a direct result of the implementation of enemies it will be included in the *Obstacle* subcategory.

Figure 4.3 shows a class diagram from the second measurable implementation. Additional classes in regard of the first implementation are:

- **Model.Enemy** - Object class inheriting the Unit class, representing an enemy and its properties.
- **Model.EnemyLogic** - Functions as a platform class between the enemy, player and TMXLevel classes in order to manage the enemies’ map and player collision, positioning and mechanics.
- **Model.Projectile** - Object class representing a projectile. Projectiles are essentially the shots coming from the enemies of the type shooter.

![Figure 4.3 Class diagram from the second implementation.](image-url)
4.2.1 Output

Figure 4.4 shows the output after the second implementation. The arrows have been added afterwards in order to clarify the functional mechanics.

![Screen dump from the second implementation's output.](image)

4.2.2 Complexity: Model

Table 4.5 shows the compiled data of complexity derived from the Model classes directly related to the second implementation. The entirely new classes are marked as “new” whilst Unit, Player and TMXLevel are classes where additional code has been added.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Maintainability Index</th>
<th>Depth of inheritance</th>
<th>Cyclomatic Complexity</th>
<th>Class Coupling</th>
<th>Class Coupling with XTile</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Unit</td>
<td>92</td>
<td>1</td>
<td>26</td>
<td>1</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>Model</td>
<td>Player</td>
<td>68</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>Model</td>
<td>Enemy *new</td>
<td>82</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>Model</td>
<td>EnemyLogic *new</td>
<td>59</td>
<td>1</td>
<td>32</td>
<td>4</td>
<td>14</td>
<td>51</td>
</tr>
<tr>
<td>Model</td>
<td>Projectile *new</td>
<td>91</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Model</td>
<td>TMXLevel</td>
<td>70</td>
<td>1</td>
<td>84</td>
<td>2</td>
<td>22</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 4.5. Model complexity data

4.2.3 Complexity: Implementation categories

Table 4.6 shows an appraisal number of lines of code and cyclomatic complexity associated with the three implementation categories and its subcategories after the second implementation.
4.2.4 Additional Data

Table 4.7 shows the size of classes not directly related to the implementation categories of this implementation.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>GameModel</td>
<td>13</td>
</tr>
<tr>
<td>Controller</td>
<td>GameController</td>
<td>20</td>
</tr>
<tr>
<td>View</td>
<td>GameView</td>
<td>23</td>
</tr>
<tr>
<td>View</td>
<td>InputHandler</td>
<td>9</td>
</tr>
<tr>
<td>View</td>
<td>Camera</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.7 Lines of code derived from other application classes

Table 4.8 shows the average update time in milliseconds from the second implementation.

<table>
<thead>
<tr>
<th>Project</th>
<th>Avg. Update Game Logic</th>
<th>Avg. Update View</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successor</td>
<td>1.06 ms</td>
<td>0.22 ms</td>
<td>60.00</td>
</tr>
</tbody>
</table>

Table 4.8 Average update time (ms) for the game logics and graphical rendering

4.3 Implementation 3 – Movement aid, Collectable items & Triggers

In the third implementation the focus were put into finishing the remaining subcategories within the implementation category Level Components. These subcategories were; Movement aid, Collectable items and Triggers.

For the Movement aid category the choice was made to implement two rather classic elements; ladders and trampolines.

As Collectable items the choice was made to implement one financial item and one power-up item; coins and a power-up enabling the avatar to fly for a limited amount of time.

Two Triggers were implemented; one that will freeze all enemies and platforms for a limited amount of time and one that will enable an additional platform or platforms.

This implementation did not require any new classes. Instead, this implementation increased the functionality of the Player, PlayerLogistic and TMXLogistic classes to some extent.
4.3.1 Output

Figure 4.5 shows the output related to Movement Aid, Collectable items and Triggers.

![Screen dump](image)

Figure 4.5 Screen dumps from the third implementation's output.

4.3.2 Complexity: Model

Table 4.9 shows the compiled data of complexity derived from the Model classes directly related to the third implementation. No entirely new classes were created.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Maintainability Index</th>
<th>Depth of inheritance</th>
<th>Cyclomatic Complexity</th>
<th>Class Coupling</th>
<th>Class Coupling with XTiled</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Unit</td>
<td>92</td>
<td>1</td>
<td>26</td>
<td>1</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>Model</td>
<td>Player</td>
<td>74</td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>6</td>
<td>49</td>
</tr>
<tr>
<td>Model</td>
<td>PlayerLogic</td>
<td>55</td>
<td>1</td>
<td>14</td>
<td>3</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>Model</td>
<td>TMXLevel</td>
<td>70</td>
<td>1</td>
<td>131</td>
<td>2</td>
<td>24</td>
<td>187</td>
</tr>
</tbody>
</table>

Table 4.9 Model complexity data

4.3.3 Complexity: Implementation Categories

Table 4.10 shows an appraisal number of lines of code and cyclomatic complexity associated with the three implementation categories and its subcategories after the third implementation.

<table>
<thead>
<tr>
<th>LEVEL COMPONENTS</th>
<th>AVATAR MECHANICS</th>
<th>LEVEL VISUALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code:</td>
<td>Lines of Code:</td>
<td>Lines of Code:</td>
</tr>
<tr>
<td>89</td>
<td>130</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>33</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>80</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>Lines of Code:</td>
<td>Lines of Code:</td>
<td>Lines of Code:</td>
</tr>
<tr>
<td>90</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>Cyclomatic</td>
<td>Cyclomatic</td>
<td>Cyclomatic</td>
</tr>
<tr>
<td>Complexity:</td>
<td>Complexity:</td>
<td>Complexity:</td>
</tr>
<tr>
<td>305</td>
<td>189</td>
<td>71</td>
</tr>
<tr>
<td>25</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>Cyclomatic</td>
<td>Cyclomatic</td>
<td>Cyclomatic</td>
</tr>
<tr>
<td>Complexity:</td>
<td>Complexity:</td>
<td>Complexity:</td>
</tr>
<tr>
<td>189</td>
<td>71</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.10 Lines of code and cyclomatic complexity derived from the implementation categories.
4.3.4 Additional Data

Table 4.11 shows the size of classes not directly related to the implementation categories.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>GameModel</td>
<td>14</td>
</tr>
<tr>
<td>Controller</td>
<td>GameController</td>
<td>22</td>
</tr>
<tr>
<td>Controller</td>
<td>MasterController</td>
<td>26</td>
</tr>
<tr>
<td>View</td>
<td>GameView</td>
<td>28</td>
</tr>
<tr>
<td>View</td>
<td>InputHandler</td>
<td>9</td>
</tr>
<tr>
<td>View</td>
<td>Camera</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.11 Lines of code derived from the other application classes

Table 4.12 shows the average update time in milliseconds from the third implementation.

<table>
<thead>
<tr>
<th>Project</th>
<th>Avg. Update Game Logic</th>
<th>Avg. Update View</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successor</td>
<td>1.06 ms</td>
<td>0.28 ms</td>
<td>60.00</td>
</tr>
</tbody>
</table>

Table 4.12 Average update time (ms) for the games logistics and graphical rendering

4.4 Implementation 4 – Level visuals

The fourth and last implementation covered the implementation category *Level visuals* and the subcategory *Graphics* in particular. Within the *Level visual* category we also find the *Sound* subcategory. However, due to the time limitation of this study, the subcategory *Sound* will not be addressed within this thesis.

The final graphical output of the prototype can be considered as rather basic. The level environment is generated with a small number of open-source tile-sheets and is mainly static. There are but one element of dynamic graphic implemented in form of levitating leaves. This implementation also includes a set of animated elements such as the player and enemies, coins and power-ups.

In order to maintain a clean and understandable structure of the game, all discernable game components were given their own view class for rendering of its graphics. Furthermore, two new view classes were implemented for the purpose of handling all animations of the game and another two classes were added for the purpose of particle effects – in this case smoke effects. This resulted in a rather large amount of additional classes.

It can be stressed that this implementation would also result in some additional code within the model classes in order link the logical behaviour of units and objects with the correct animations. This code however is not considered to be of any significant matter and will not be taken in consideration when the complexity of this implementation is measured.
Figure 4.6 shows the class diagram after the fourth implementation. Additional classes in regard of the fourth implementation are:

- **View.SmokeParticle** - Object class representing a smoke particle.
- **View.SmokeSystem** - Class responsible for generating smoke effects.
- **View.Animation** - Object class representing a sprite texture.
- **View.AnimationSystem** - Class responsible for animating sprite textures.
- **View.GraphicView** - View class responsible for rendering of separate graphical components, such as environment effects and backgrounds.
- **View.ItemView** - View class responsible for rendering all items, power-ups, movement-aid objects and triggers.
- **View.UIView** - View class responsible for rendering the user interface such as life and coin count.
- **View.UnitView** - View class responsible for rendering all unit objects such as player and enemies.
- **Model.State** - Class containing a list of necessary object states in order to determine how to change the animation of units and items depending on its state.

![Figure 4.6 Class diagram from the fourth implementation.](image)

### 4.4.1 Output

Figure 4.7 shows the output result from the fourth and last implementation.
Figure 4.7 Screen dumps from the fourth implementation.

4.4.2 Complexity: View
Table 4.13 shows the compiled data of complexity derived from the View classes directly related to the fourth implementation.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Maintainability Index</th>
<th>Depth of inheritance</th>
<th>Cyclomatic Complexity</th>
<th>Class Coupling</th>
<th>Class Coupling with XTiled</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>SmokeParticle</td>
<td>81</td>
<td>1</td>
<td>18</td>
<td>2</td>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>View</td>
<td>SmokeSystem</td>
<td>75</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>View</td>
<td>Animator</td>
<td>72</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>7</td>
<td>39</td>
</tr>
<tr>
<td>View</td>
<td>Animationsystem</td>
<td>55</td>
<td>1</td>
<td>25</td>
<td>3</td>
<td>9</td>
<td>77</td>
</tr>
<tr>
<td>View</td>
<td>GraphicView</td>
<td>67</td>
<td>1</td>
<td>14</td>
<td>3</td>
<td>19</td>
<td>52</td>
</tr>
<tr>
<td>View</td>
<td>ItemView</td>
<td>48</td>
<td>1</td>
<td>22</td>
<td>4</td>
<td>14</td>
<td>47</td>
</tr>
<tr>
<td>View</td>
<td>UnitView</td>
<td>43</td>
<td>1</td>
<td>31</td>
<td>5</td>
<td>19</td>
<td>62</td>
</tr>
<tr>
<td>View</td>
<td>UIView</td>
<td>74</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Model</td>
<td>State</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.13 View complexity data.

4.4.3 Complexity: Implementation Categories
Table 4.14 shows an appraisal number of lines of code and cyclomatic complexity associated with the three implementation categories and its subcategories.
4.4.4 Additional Data

Table 4.15 shows the size of classes not directly related to the implementation categories.

<table>
<thead>
<tr>
<th>Name-space</th>
<th>Class</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>GameModel</td>
<td>14</td>
</tr>
<tr>
<td>Controller</td>
<td>GameController</td>
<td>60</td>
</tr>
<tr>
<td>Controller</td>
<td>MasterController</td>
<td>26</td>
</tr>
<tr>
<td>View</td>
<td>GameView</td>
<td>31</td>
</tr>
<tr>
<td>View</td>
<td>InputHandler</td>
<td>19</td>
</tr>
<tr>
<td>View</td>
<td>Camera</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.15 Lines of code derived from the other application classes

Table 4.16 shows the average update time in milliseconds from the fourth implementation.

<table>
<thead>
<tr>
<th>Project</th>
<th>Avg. Update Game Logic</th>
<th>Avg. Update View</th>
<th>Frames Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successor</td>
<td>1.06 ms</td>
<td>0.50 ms</td>
<td>60.00</td>
</tr>
</tbody>
</table>

Table 4.16 Average update time (ms) for the game logics and graphical rendering
5 Result analysis

In this chapter we will examine the collected data presented in Chapter 4: Result in order to investigate if any immediate and — or obvious conclusions can be drawn. The result analysis will then continue to speculate about the results in relation to the problem formulation declared in chapter two.

5.1 Observation

When examining the collected complexity data there are a handful of facts that can be established without speculations. In regard of the Implementation categories; Level components, Avatar mechanics and Level visuals it can be observed that Level components and Level visual generated the largest amount of combined complexity. In fact, those two categories resulted in a similar amount of code lines; slightly above the number three hundred. The Level components category however showed a higher number in cyclomatic complexity indicating that the logic functionality of the prototype represents somewhat higher amount of decision-making processes than the graphical rendering.

Figure 5.1 shows a diagram representing the compiled amount of lines of code measured from the three Implementation categories whilst Figure 5.2 shows a diagram representing the compiled cyclomatic complexity.

![Diagram](image.png)

*Figure 5.1* Compiled complexity results: lines of code - implementation categories
When examining the collected complexity data derived from all the subcategories we find that the subcategory Graphic is by far the most complex. However, this fact can be considered as slightly misleading since the Level visuals category only consists of two subcategories of which Graphics is the primary one.

When studying the subcategories, presented in Figure 5.3 and 5.4, we find that lines of code and the cyclomatic complexity is fairly correspondent. Obstacles show the second highest values after Graphics, followed by Platforms and Basic mechanics. The remaining subcategories; Movement aids, Triggers, Collectable items and Collision, all show similar, comparatively low complexity values.
In Figure 5.5 and 5.6 the complexity data is compiled even further by combining the *lines of code* with the *cyclomatic complexity*. The diagrams now represent both size and the amount of decision-making derived from each category.
5.2 Problem reflection

In regard of the problems formulated in 1.2 Problem formulation we believe that the results do achieve several of the purposes of our hypothesis.

When reviewing the underlying reasons for the problem formulation – where focus was put towards lack of knowledge regarding design-related features, time consumption and complexity – it is our belief that the results presented in this thesis could very well serve as a contribution. By following the framework described as Implementation categories - provided from the theoretical part of this thesis – we successfully developed a small scale platform game offering several popular components rediscovered throughout the genre. Additionally, the complexity data compiled from the prototype provides a rather clear – and hopefully understanding – view of the different categories combined complexity as well as their individual complexity.

When reviewing the two main questions formulated in the problem formulation; which important design and development related components should be deferred to when developing a small-scale platform game?, and what is the relation and distribution between these components in regard of software complexity?, it is our statement that the prior question does get answered within this thesis, though perhaps not fully. We have in fact – through previous research – presented a handful of design and development related components within our Implementation categories. However, the possibility of identifying additionally
components that could be motivated as important must be considered as rather high. Therefore, in regard of this first question, we consider it to be partly answered rather than fully answered. The latter question however - regarding the software complexity – we state to be answered in a quite detailed fashion. It should be noted though, that this question can of course only be answered in relation to the answer of the prior question.

When observing the compiled results it becomes quite clear to us that the subcategory Graphics appears to be too large. Our immediate conclusion drawn by this fact is that the Level visuals category should perhaps be divided into more subcategories - much in resemblance to the Level components category.

Whether the results can be used in order to prioritize and plan a development process in regard of time consumption is yet to be discovered.
6 Discussion

This chapter will present a general reflection regarding the thesis and its result as a whole. It will then continue with a discussion regarding the thesis validity and usability as well as a short reflection regarding the social and civic aspects in relation to the thesis.

6.1 Reflection

Reflecting upon the thesis and its results as a whole we consider this study to be rather successful. We did manage to create a light framework attached with complexity examples and a complete prototype which hopefully could be of aid in small-scale, independent or academic game projects in regard of decision making, prioritization and time planning.

With that said, it will be uncertain whether our hypothesis will be as useful as we hope until it has been utilized.

6.2 Validity

It is important to emphasize that the problem of this thesis is primarily based ion the conclusions drawn by D. Boutros [2] and cannot be considered as a confirmed and -or official problem. However, it is our belief that the recent trend - where remakes of previously successful platform-games generate a remarkable increase of interest within the genre - cannot be ignored. Though the reasons for the fluctuating curve of interest towards the platform-game genre could be many, it is still our belief that the problem formulation of this thesis is valid and very much worthy to address.

Concerning the theoretical analysis that resulted in our implementation framework we believe our Implementation categories to be of high validity. This statement is based on the high recurrence of similar or identical categories found in previous research regarding this specific genre. Collision and Sound are the only subcategories that is not mentioned in the previous research utilized within this thesis and could thereby be subjects for questioning.

Even though we strived to keep the generalization – in regard of environment, structure and language - of the prototype to a maximum it is still difficult to anticipate the complexity divergence that might occur when developing in a whole other environment, design pattern or programming language.

Another variable that might have affected the outcome of this thesis is the usage of third party libraries. In this thesis such libraries facilitated the implementation of for example collision to some extent. If such functionality instead is programmed from scratch the complexity outcome may differ.

Additionally, the scale of our prototype must be taken under consideration. As mentioned before our prototype game must be considered as rather small scale. It is unclear how the complexity from the different Implementation categories may differ in relation to the game size.
6.3 Usability

The very purpose of this thesis is that it will be utilized by others. We sincerely hope that our small framework of implementation categories - attached with the complexity examples and a complete game prototype - will be of assistance to students and –or independent developers when creating small scale games for any platform.

Hopefully this thesis will aid the discussion regarding which components to include, which to prioritize and which to exclude due to possible time limitation.

Perhaps the thesis can be of use in other game genres as well. We find no reason to why our method of identifying important implementation categories – and measuring their complexity – could not be adopted to any other game genre as well.

To stretch it even further, perhaps the method used within this thesis can be performed to software not related to games at all. As already mentioned, in 1.4 Target Group, the thesis strategy itself - where important design related elements are identified, decomposed into implementation categories, implemented in a prototype and finally measured for complexity - may be applicable in other fields of study.

6.4 Social aspects

One of the main motivations for doing this thesis was the possibility to target small-scale developers while still retaining a certain thesis-validity. In recent years independent and small-scale developers have had notable impact on the gaming industry as a whole. It is our belief that this group of developers, together with developers of academic sort, should be recognized as a distinguished target group in regard to modern game development. It seems like in today’s game development - more so than ever - an architect, a designer, a developer and a tester, may at several occasions be the very same individual. It is our belief that we are witnessing the rise of a new type of common developers that differentiate themselves by requiring a lot more general knowledge (of game development in general) in comparison to more field specific developers, designers or architectures.

When looking at this thesis from an even broader perspective it can be stated that it does not address any specific or current social aspects such as games for educational purposes, effects of video games in general nor the ongoing debate about games contribution to asocial or social behaviour. Since this thesis addresses a rather classic and by tradition non-violent game genre - and since the primary goal of the thesis is to improve the development of games within that genre - we hope this thesis will be perceived as very much neutral in regard of any of these larger social aspects.
7 Conclusion
This chapter will present a final conclusion regarding the thesis as a whole followed by a short discussion regarding possible future research based on this thesis.

7.1 Conclusion
Despite a handful of concerns regarding the thesis validity we proclaim this thesis to be successful in regard of its original goals. We believe that the thesis results can be of assistance to any small-scale project and that its method could be applied within software engineering in general.

7.2 Future research
In order to increase the validity of this thesis it would be of great interest to see a similar study done with a different programming environment, design pattern and –or programming language.

Even though we strived to keep our prototype as generalized as possible it is likely to be more appreciated by developers using the same environment and programming language as in this thesis.

A similar study performed with another programming language and environment – perhaps compared or combined with this study – we believe would eliminate much of the validity concerns currently present within this thesis.

It is our belief that an extended study similar to this thesis, covering a larger amount of game components, developing a larger prototype, could be of interest to a significantly larger target group.

Additionally, we believe that a study similar to this thesis could be performed to any game genre. Doing a similar study, perhaps targeting a 3D game genre, we believe could generate a large amount of attention.
References

Scholarly Articles & books


**Online blogs and official web platforms**


[18] Microsoft Developer Network, *XNA Game Studios 4.0*,


[23] Online Magazine Article, RacketBOY,


[25] *Model-View-Controller*, online paper,


Appendixes
Appendix A presents all the classes implemented in the game prototype, including variables, properties and methods. Appendix B provides an online link to the complete source code of the prototype.

Appendix A: Classes

Figure A.1 All classes within the Model namespace.
Figure A.2 All classes within the Controller and View namespaces.
Appendix B: Source code

https://github.com/ag222hd/XNA-platform-game/blob/master/Successor-1.3.rar