Video Games as a Medium for Software Education

Bradley R. C. Marques

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering

Johannesburg, 2012

Declaration

I declare that this dissertation is my own unaided work. It is being submitted to the Degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Bradley R. C. Marques

Signed this _____ day of _____ 20 _____

Abstract

The work presented contributes to the field of serious and educational games. Specifically, the use of video games as a medium for visualising and teaching computer programming is investigated. This research contextualises video games within existing taxonomies of software visualisation techniques. The factors of educational game design that should be prioritised are elicited from perspectives of both students and educators. As part of the investigation, an educational game that teaches basic computer programming concepts has been iteratively designed and developed. It was tested on 130 university-level students. An analytics system collects data about how players engage with the game; a survey was used to collect qualitative user experience data; and a multiple-choice software skills test was administered before and after students played the game. It is shown that novice computer programmers increased their marks by 8% in the skills test after playing the game for under 45 minutes.

Keywords: Educational Games, Games, Gamification, Serious Games, Software education, Software pedagogy, Software visualisation, Visual programming

Acknowledgements

I wish to extend my sincere thanks to my supervisors, Ken Nixon and Stephen Levitt, for their invaluable support, guidance and assistance; I have learnt a lot from them both. I would also like to thank Jessie Liu for her help with the testing of the project, and Hanli Geyser, Estelle Trengrove and Ivan Hofsajer for their assistance.

I would also like to acknowledge the financial assistance of the National Research Foundation (NRF) for the funding of this research.

I would like to thank my parents and Megan for their constant support and belief in me.

Contents

List of Figures

List of Tables

CHAPTER 1

Overview

1.1 Introduction

> “Software is hard.”
>
> — *Donald Knuth*

This was the first statement made during the first lecture in my Software Development II course; and for many students, it was completely accurate. They battled to grasp the concepts and material in the course. The perceived level of difficulty of software is partly due to the intangibility of computer programming, and many students find it difficult to visualise what their code is doing. It has a negative impact on the number of young people, especially females, choosing computer science and software development as career paths [1, 2]. Indeed, there is a globally declining interest in computer science and engineering as fewer high school students are choosing careers in these fields [1, 2]. Concurrently, the global video game industry continues to grow and reach a wider demographic, particularly with the rise of mobile and casual games [3]. By the age of twenty-one, the average person in the developed world has spent around ten-thousand hours playing games: the same amount of time spent in their entire schooling career [4]. Software visualisation is the rendering of a software system into a tangible, visual format to aid code comprehension [5, 6]. Software visualisation techniques are thus employed in the professional software development industry, but can also be used to

augment the teaching of software development and computer programming [7, 3, 8]. Many aspects of software visualisation mirror those of digital games: the use of visual metaphor; human-computer interaction considerations; user interface design; and the use of audio and graphics. This research examines how digital games may be leveraged as an effective medium for the teaching of basic computer programming, and what aspects of such educational games are most important from the perspective of a student and an educator.

As part of this investigation, an educational computer game that teaches a small scope of software development principles has been created. The game – the final version of which is called *if(traffic)else{win}* – is a puzzle game that requires players to programme the solutions to its levels. Players are required to direct traffic to the correct locations by using conditional statements and logical operators. It also teaches `C`-like programming syntax. It was iteratively designed, developed and tested in three iterations. Testing data shows that the game can be used to increase the understanding of the areas of conditional statements and flow control by about 8% in under 45 minutes.

This chapter introduces the research by means of:

- a broad overview of the research area and research questions (see Section 1.2);
- an overview of the research process in terms of activities, experimental methods and data collection and analysis (see Section 1.3);
- a description of the importance of the research and a brief contextualisation within existing research (see Section 1.4);
- a guide to the rest of the dissertation (see Section 1.5).

1.2 Research Area and Research Questions

This research examines the application of digital games as a means of visualising and thus aiding the learning and teaching of a few basic computer programming concepts. As a result, the fields of digital games, software visualisation and visual programming are considered, as shown in Figure 1.1. Encompassed within this research is the field of software visualisation and the related concept of visual programming – which are examined in detail in Section 2.3. Software visualisation is the process by which an intangible or complex software system is analysed and presented in a simplified, easy to understand, visual format [7, 5, 6]. This visualisation may be for a variety of purposes, such as highlighting some useful derived metrics, elucidating the high-level software structure, or visualising the low-level algorithmic

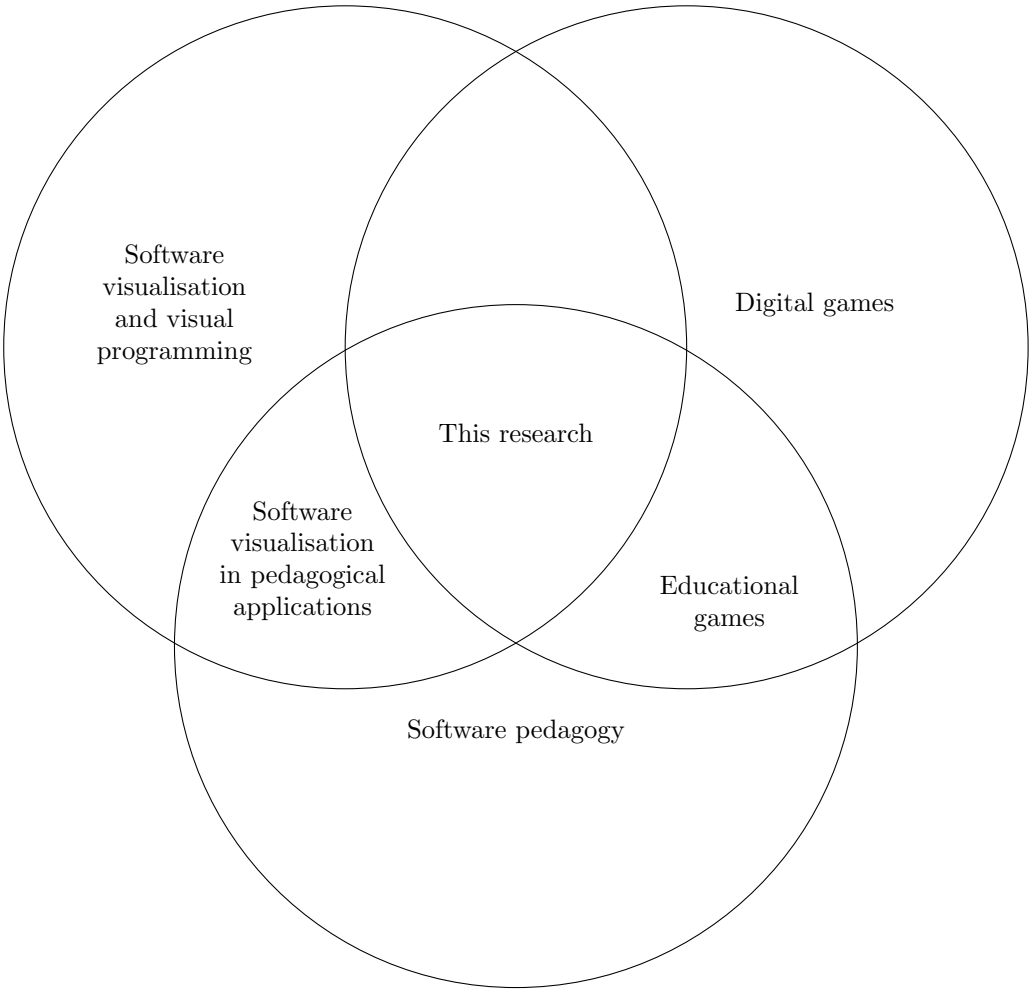


Figure 1.1: The main research areas of the investigation.

functioning, function calls, logging information or memory usage data of a software system [9]. In all cases, software visualisation aids code comprehension [5, 9]. Visual programming, on the other hand, is the generation of a software system through graphical means [10]. Software visualisation and visual programming are widely used in professional software development, but can also be employed in software pedagogy [7, 8]. The realms of digital games, serious and educational games and gamification also fall within the research area – these are investigated in Section 2.4. Digital games – any game played on an electronic device – are examined from the perspective of software visualisation, and contextualised within an existing taxonomy of software visualisation techniques (see Section 2.3.2 and Section 2.4.4). 'Serious games' are games in which the main purpose is not entertainment. This includes socio-political commentary, art, and training or education. Educational games, the main focus of this research, are thus a subset of serious games. A related field that is examined is that of 'gamification' – the application of game elements and game design techniques to non-game

related areas [11]. These areas of the research are examined fully in Chapter 2.

This research examines the use of educational games to provide software visualisation and visual programming environments for the purpose of teaching novice computer programming. Specifically, the research questions are posed as:

1. where do digital games fit within a taxonomy of software visualisation techniques?
2. when designing an educational game for teaching novice computer programming skills:
   (a) what factors should be prioritised, from a student's perspective?
   (b) what factors should be prioritised, from an educator's perspective?

The method of investigating and answering these questions is examined in the following section.

1.3 Research Overview

This section provides a summary of the entire research process by examining the main research activities, the experimental methods employed, and techniques of data collection and analysis. Where applicable, key dates and periods of work are specified. The system that was developed, including the game, is also broadly examined.

An investigation into the literature on software visualisation and visual programming was performed and a first-draft literature review completed in February 2012 (the reader is referred to Appendix A for a Gantt chart of research activities). Further investigation into serious and educational games was also performed. The main findings of this survey are documented in Chapter 2. A very simple, small-scope educational game, named *if(traffic)else{win}*, was designed, developed and tested – over three iterations – as part of this investigation. The computer game is a puzzle game in which players are required to direct random input traffic to the correct destinations. Players do this by programming the traffic lights in the virtual road network. Traffic can consist of different vehicle types (cars, taxis and trucks) of different colours. Directing the traffic thus represents a simplistic sorting algorithm by which players query the vehicle's colour and type, and perform certain actions depending on these conditions. The game thus introduces basic syntax, and the *if*, *else* and *else if* conditional statements. The logical *&&* (*AND*) operator is introduced in the later levels. Players win a level, and are thus permitted to proceed to the next, by completing a correct sorting algorithm for the specific puzzle presented. The design of the game is discussed further in Chapter 3.

The three iterations of design and development took place from April to August 2012. For

Table 1.1: The participant groups with which the game was tested

Iteration Number	Student group	Number of Students
1	Game Design	39
2	Applied Computing	14
3	Electrical Engineering	84

each iteration, the game was also tested on a group of students from the University of the Witwatersrand. Ethics approval (protocol number H120407) was applied for and granted by the research office at the University of the Witwatersrand. Such experiments involved students playing the game (see Chapter 3), completing a test and filling out a survey (see Chapter 4) in a period of 45 minutes. As shown in Table 1.1, the first iteration was tested on a mixed group of students – 39 in total – including Electrical, Biomedical and Information engineers from different years, but the group mainly consisted of first-year Game Design students from both engineering and arts streams. The second iteration was tested on a group of 14 first-year Applied Computing students, and the third on a group of 84 first-year Electrical Engineering students.

The game was developed using the *Unity3D* engine as a web-based game for Windows [12]. A number of auxiliary systems were designed to collect data about the game. Figure 1.2 shows these systems in a client-server configuration. In order to judge the usefulness of the game and its educational value in teaching basic computer programming, a multiple-choice software skills quiz was administered to research participants both before and after playing the game. The quiz was created with *LimeSurvey* [13]. Although it is difficult to measure educational value, the increase in test results of participants is, for the purpose of this research, assumed to be an indicatation of the game's educational quality. An improvement is suggested in Section 5.2, in which the game's impact on student marks is measured in relation to other activities, such as reading a textbook extract or attending a lecture on the same content. An integrated analytics system was also developed to collect data about how players engage with the game. The data collected varied between iterations; however, data such as the time taken per level, the number of reattempts and the number of lines of code used in level solutions are captured and stored in a server database. Participants also completed a survey – also created with *LimeSurvey* – which was used to profile the participants in terms of their gender, occupation, year of study, and previous programming and gaming experience. The survey was also used to collect qualitative data about how students felt about the game. Participants

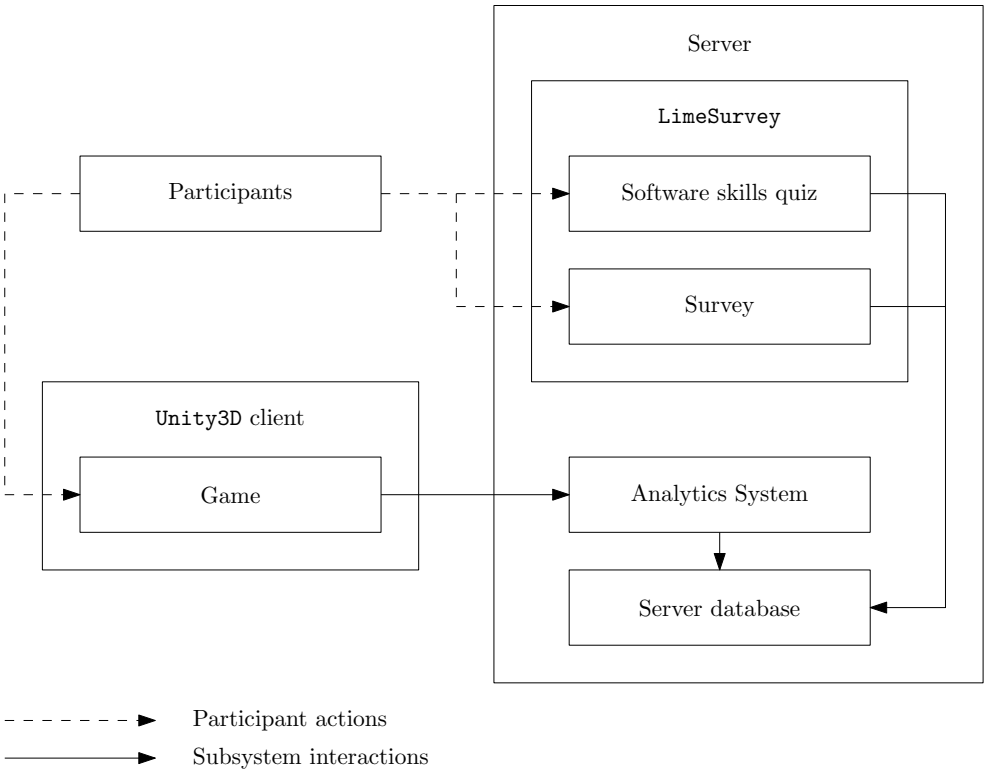


Figure 1.2: A high-level overview of the system components.

were able to judge aspects such as their perceived usefulness of the tool, the quality of audio, user interface and graphics. The survey also allowed participants to give open-ended feedback about the game and research. A semantic differential and Likert scale are used to present and analyse the data; the reader is referred to Chapter 4 for more details on the testing methodology and results [14].

This research has been presented at conferences and events both internally (at the University of the Witwatersrand) and externally. The research proposal was presented at an internal conference, PEIES (Postgraduate Electrical and Information Engineering Symposium) in April 2012. The work of the first iteration was compiled into a paper and submitted to the SAICSIT (South African Institute of Computer Scientists and Information Technologists) 2012 conference and presented in October 2012. This paper may be found in Appendix B. The paper was presented at a non-technical local games festival, *Amaze / Interact*, in August 2012. A technical paper covering all three iterations was written for an international peer-reviewed conference, the IEEE IGIC (International Games Innovation Conference). The paper – which may be found in Appendix C – was accepted to the conference and presented in Rochester, New York in September 2012. The work was also presented at the Fourth Annual Cross-Faculty Postgraduate Symposium in the prestige plenary session category. Finally, the work was

presented at the Serious Games Institute of South Africa at North West University in October 2012 by invitation.

1.4 Research Justification

> “How do we get people to understand programming? We change programming. We turn it into something that's understandable by people.”
>
> — *Bret Victor*

As mentioned, many students find software intangible and difficult to visualise [7, 8]. This contributes to the declining interest in computer science and engineering as young people feel daunted by these fields [1, 2]. In turn, this decreases the number of engineering and information technology graduates which greatly hampers global technological development [1, 2]. However, the computer and video games industry continues to grow, especially with the rise of mobile and app-based games. This research thus examines the use of digital games to visualise software in order to aid the traditional teaching of software development, and stimulate an interest in the field. The research is further justified in the literature review (of Chapter 2). The following section provides a guide to the remainder of the dissertation.

1.5 Organisation of the Remainder of the Dissertation

The remainder of this dissertation is presented as follows:

Chapter 2: A review of relevant literature is presented. The declining interest in computer science mentioned briefly in this chapter is expanded upon in Section 2.2. The fields of software visualisation and visual programming are discussed in Section 2.3. Existing taxonomies of software visualisation techniques are examined in Section 2.3.2. The realms of serious games, educational games and gamification are investigated in Section 2.4, and the contributions of games to academic teaching are discussed in Section 2.4.5.

Chapter 3: The implementation methodology of the research is presented. The taxonomy of software visualisation tools is used as a conceptual model for the presentation of material in this section. For instance, Section 3.2 discusses the task of the tool (its purpose) and its intended audience. The particular aspects of software that are visualised by the system are

discussed in Section 3.3, and how this visualisation takes place in terms of representations and media used are investigated in Section 3.4.

Chapter 4: The testing methodology and results are presented. An overview of the entire experimental procedure is given in Section 4.2. Each component of the testing framework is discussed in more detail: Section 4.3 provides a description of the analytics system; Section 4.4 a description of the software skills test, and Section 4.5 presents the qualitative survey. The results obtained from these test methods are presented and analysed in Section 4.6. Section 4.7 presents some lessons learnt and observations made from the research process.

Chapter 5: Recommendations for possible future work are discussed. Specifically, recommendations for the immediate future are presented in Section 5.2. A number of alternative ideas that were ultimately not implemented are discussed in Section 5.3. Finally, the limitations of the developed tool are documented in Section 5.4.

Chapter 6: This chapter concludes the dissertation by recapitulating the main research process and summarising the main findings of the paper – see Section 6.2. Section 6.3 restates the research questions and summarises how the presented work leads to the answering of each. Finally, Section 6.4 postulates on the possible future of educational games and how this research could be used.

Additional supporting material may be found in the appendices of the dissertation. These appendices are presented as follows:

Appendix A: This appendix presents a Gantt chart showing the time breakdown for the research.

Appendix B: The technical paper accepted for and presented at SAICSIT 2012 is documented.

Appendix C: The technical paper accepted for the IEEE IGIC 2012 is presented.

Appendix D: This appendix presents the software skills test administered to players

before and after playing the game.

Appendix E: The survey that was used to collect qualitative data about the game, audience and user experience is presented in this appendix.

1.6 Summary

This chapter introduces the main components of the research that is investigated in more detail in later chapters. The salient points are as follows:

- this research is an investigation into the use of an educational game to teach a few novice computer programming skills;
- the research covers areas of software visualisation, visual programming, serious games, educational games and gamification;
- the research attempts to answer two research questions:
  1. where do digital games fit within a taxonomy of software visualisation techniques?
  2. when designing an educational game for teaching novice computer programming skills:
     (a) what factors should be prioritised, from a student's perspective?
     (b) what factors should be prioritised, from an educator's perspective?
- an educational computer game was iteratively designed, developed and tested on university students;
- both qualitative and quantitative data were collected by an in-game analytics system, a survey, and a software test that was completed before and after participants played the game.

In order to contextualise the research implementation and results, the following chapter presents a review of relevant literature in the research areas.

CHAPTER 2

Literature Review

2.1 Introduction

Chapter 1 provides a broad overview of the research, and introduces a few of the main concept areas such as software visualisation, visual programming and serious games. This chapter presents a review of existing literature in these and related areas. This chapter thus provides a more detailed analysis of the research problem domain, and contextualises the research within existing work. The chapter provides:

- an overview of the declining interest in computer science and the consequences of this (see Section 2.2);
- a detailed investigation of the field of software visualisation in terms of its definition, process and relation to similar fields (see Section 2.3);
- a discussion of a few existing taxonomies of software visualisation tools, and the presentation of a unified taxonomy (see Section 2.3.2);
- an investigation into the applications of software visualisation in industry and computer science pedagogy (see Section 2.3.3);
- an analysis of the field of digital gaming, with emphasis on serious games, educational games and gamification (see Section 2.4);
- a discussion of how digital games may fit within a taxonomy of software visualisation (see Section 2.4.4);

- an investigation of the benefits and downfalls of using games in education (see Section 2.4.5).

2.2 The Declining Interest in Computer Science

There is currently a globally declining interest in the fields of computer science and engineering [1, 2]. Whilst the number of students entering tertiary institutions has increased since the year 2000, the number of students choosing computer science and engineering has decreased in the same period [2]. Many factors (social, economic and political) have been contributing to the decline in these areas. For instance, the number of schools offering computer science has decreased, resulting in the lessening of a young individual's exposure to the field [2]. This lack of exposure has exacerbated a high level of perceived difficulty in the field [1, 2]. There is also a highly negative image of the field, especially among females, and a perception of low salary and job opportunities [1, 2]. The declining interest in computer science has a profound impact on the future of technological advancement. For instance, the decrease in the number of technology experts results in the increase of the cost of technological advancement [1]. The number of educators in technology areas is also decreasing, further exacerbating this decline of skills [1].

It is thus important to investigate possible ways of re-engaging young people in the learning of computer science and engineering skills. It is important to lessen the barrier to entry to computer programming and provide experiences to students that are relatable, understandable and engaging. Software visualisation, for instance, can present software concepts in these ways; indeed, software visualisation techniques have been applied to the domain of academic teaching since as early as the mid nineteen-seventies [8].

To understand digital games as a means of visualising software, the structure and methodologies of software visualisation techniques are investigated in the following section.

2.3 Software Visualisation

2.3.1 Software Visualisation and Associated Fields

Software visualisation is the abstraction of information about software systems for visual portrayal. According to Price *et al.* [5], it is the application of "typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and the effective use of computer software". The

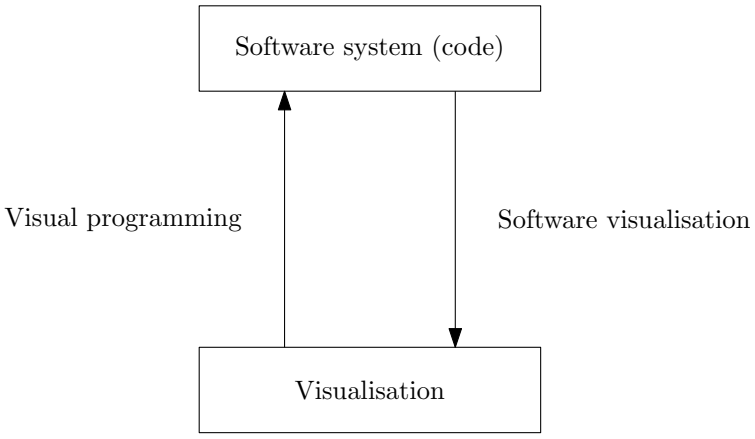


Figure 2.1: Relationship between software visualisation and visual programming techniques.

goal of all software visualisation tools is to add useful information, or present data in way that augments rapid code comprehension [15]. Often, the data sets involved (such as a code base) are large and complex, and software visualisation thus represents a cognitively efficient way of ordering this vast data into useful information. It is thus a subset of the field of data visualisation, which is typically concerned with the transformation of large, meaningless, invisible data sets into useful, visible information. The definition given by Price *et al.* [5] also indicates some possible media for the visualisation of software. Indeed, even simple chalkboard sketches or pen-and-paper diagrams of software are considered rudimentary forms of software visualisation [5]. Also noteworthy is the mention of cinematography – one of the first pedagogical uses of software visualisation was a thirty-minute motion capture film entitled *Sorting out Sorting* [8] – see Section 2.3.3. This implies that other sensory faculties – such as hearing, spatial reasoning, haptics, memory and even smell – can be used in the portrayal of software systems [5]. Software visualisation systems thus involve some sensory transduction of the raw data. It is vital that during this transformation the density of information is compressed; this is referred to as the cognitive economy of the tool. Software visualisation thus represents an overlap of the fields of software engineering, data visualisation, and human-computer interaction.

Software visualisation may concern some static aspect of the code – such as the high-level architecture or class design – or the dynamic behaviour of the code – such as run-time function calls, algorithm animation or project management data [10]. This is discussed in more detail in Section 2.3.3. Software visualisation is closely related to a field known as "visual programming". A distinction is found in the taxonomy laid out by Myers [10]. Visual programming is the *specification* of a computer program using graphical techniques, whereas software visualisation

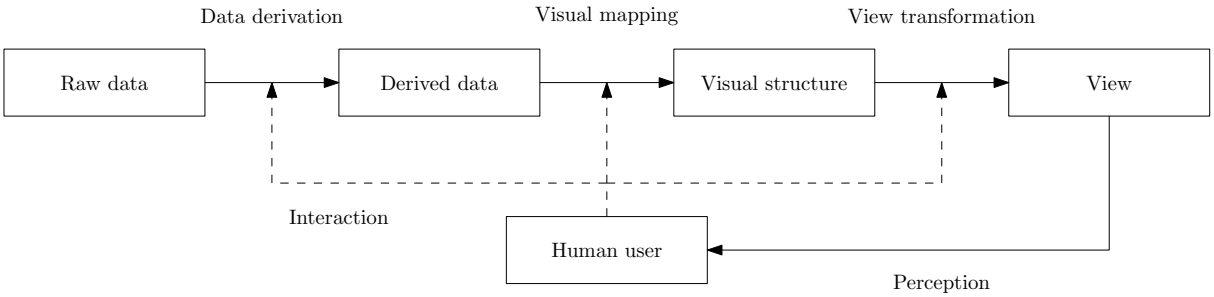


Figure 2.2: The process of software visualisation. Adapted from [16] and [6].

is the *illustration* of a pre-existing software system in a visual rendition. This relationship is depicted in Figure 2.1. As discussed in Section 2.3.3, many digital games aimed at the education of computer programming – including the game developed for this investigation – are both visual programming environments and software visualisation tools.

Another important aspect of software visualisation is that it is largely interactive. It involves a dialogue between the user and the underlying raw data, the source code [16, 6]. This relationship is shown in Figure 2.2, which was constructed from two models of software visualisation tools [16, 6]. The process involves the derivation of secondary data from the raw data. This data is then mapped into visual structures. It is vital that the tool makes a direct link between the visual structures and the underlying raw data; many software visualisation tools are lacking in this regard, resulting in this interface being termed the *missing link* of software visualisation [17]. The visual structures are, in turn, transformed into views. Views may offer: different levels of granularity; a different angle in a 3D visualisation; or features such as zoom or filters [6]. It is these views, not the visual structure, that are perceived by the human user, and thus cognitive aspects of software visualisation tools are important at this stage. Since the exploration of a software system (by features such as filters or layout options) aids the user in understanding its structure, the user must be able to interact with the visualisation [5, 6]. He or she must be able to control the processes of data derivation, visual mapping and view transformations within the above process.

This section provides an overview of software visualisation in terms of its goal, its relation to visual programming, the visualisation process and some possible media for this. Software visualisation systems may be described further by examining a taxonomy of these tools, as in the following section.

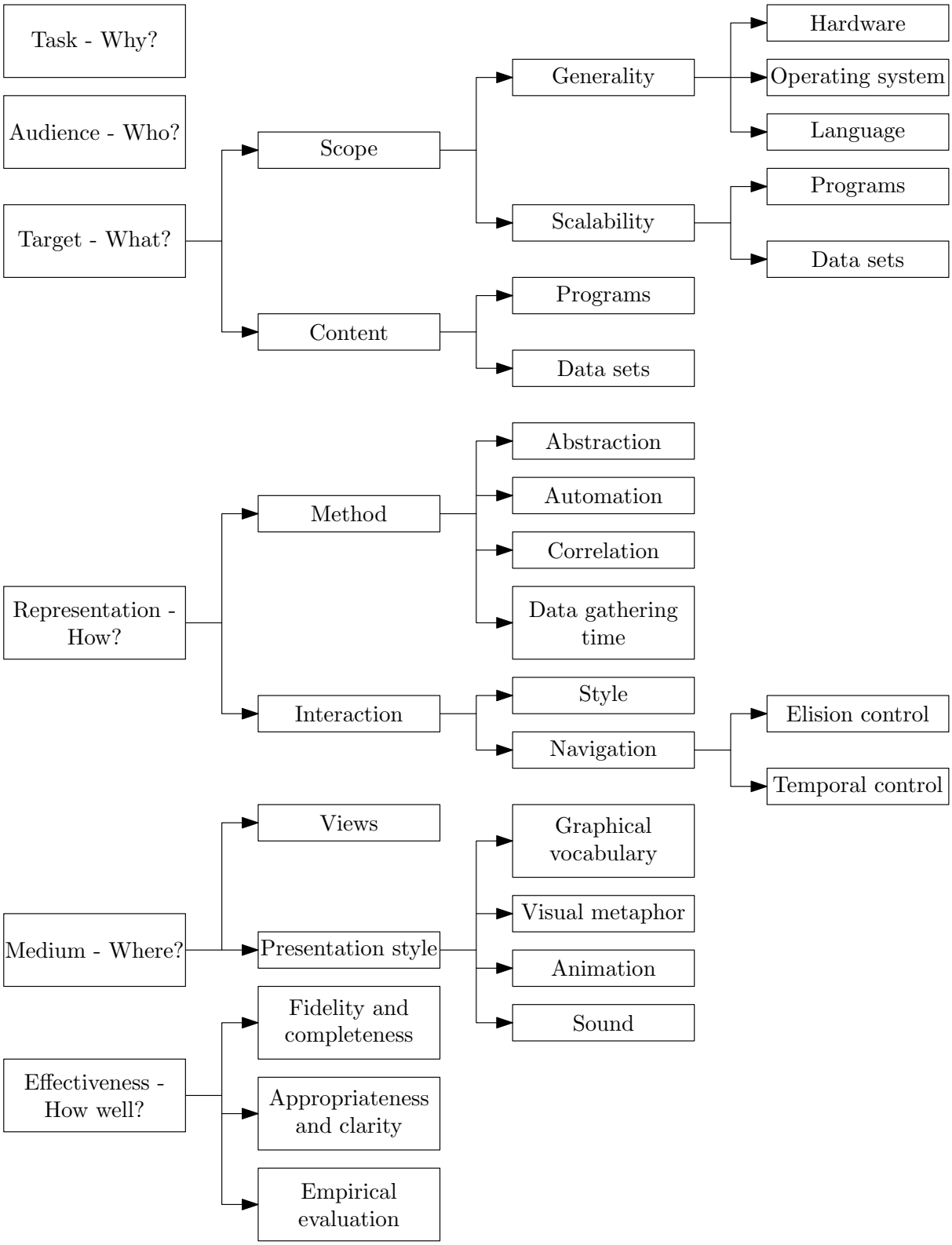


Figure 2.3: A unified taxonomy of software visualisation techniques. Adapted from [5], [6] and [10].

2.3.2 A Taxonomy of Software Visualisation Techniques

A number of taxonomies exist which define the main areas of software visualisation tools [5, 6, 10]. Such taxonomies help identify the main components of these tools, which is useful to distinguish them, and can also be used as a framework for their evaluation. This section provides a taxonomy that represents a union of previously-described taxonomic systems [5, 6, 10]. The presented taxonomy is structured as per Maletic *et al.* [6], but contains additional taxa that were found in the taxonomies of Price *et al.* [5] and Myers [10].

Figure 2.3 defines six main taxa of software visualisation tools: task, audience, target, representation, medium and effectiveness. The major taxa are further subdivided where necessary. Each taxon attempts to answer a particular question about the software visualisation system, as described below.

Task: This taxon describes why the software visualisation exists and what task it is serving; it describes the purpose of the system [5, 6].

Audience: The audience taxon specifies for whom the tool is intended [6]. The audience thus relates to the human user of Figure 2.2. The intended audience greatly affects the design of the tool in many ways: the level of abstraction, institutional culture, and language use. A tool intended for use by students and novice programmers would also have to assume a much lower skill level than a tool developed for professional developers. The developed game, for instance, is aimed at high-school and university-level students, and has very different requirements to a software visualisation tool to be used by professional software developers.

Target: Target describes what aspects of the software system the tool visualises [6]. Target is further sub-categorised into scope and content. Scope is the subset of software systems that can be visualised using the software visualisation system; it thus encompasses the generality – in terms of hardware, operating system and language – and scalability – in terms of supported programs or data sets – of the software visualisation tool [5]. Content is the subset of data or programs that are visualised [5]. The content thus correlates to the raw data of Figure 2.2.

Representation: Representation describes how the derived data are visualised; this includes the visual mapping, visual structure and view transformations of Figure 2.2. Representation may be further subdivided into method and interaction. Method describes the abstraction or level of granularity that the visualisation shows [5, 10], the degree to which the visualisation is automated [10], the method of correlating the visualisation to the underlying data [5, 10], and whether the data is gathered at compile-time or run-time [5]. Interaction

describes the style of interface, and specifies how much control the user has over temporal and elision (how much data and which data are shown) aspects [5, 10]. The concepts of abstraction and navigation are also important in digital games, and thus representation is of particular importance when applying games as a means of visualising software.

Medium: The medium specifies where the visualisation is presented. It encompasses the idea of views of Figure 2.2, and defines the presentation style: the use of graphical vocabulary, appropriate visual metaphor, and the use of animation or sound [5]. This aspect of software visualisation is also important when using games as a medium of software visualisation.

Effectiveness: Effectiveness attempts to evaluate how well the software visualisation tool fulfils its task and other requirements. Fidelity and completeness is required to ensure that the visual representations fully and truthfully portray the underlying software [5]. Appropriateness and clarity ensures that such portrayals are suitable and unambiguous [5]. Finally, the effectiveness of a software visualisation tool may be measured by empirical evaluation. Research has shown that proper empirical analysis is a rarity in this field [5, 7]. This research has presented a model for empirically evaluating game-based software visualisation tools for pedagogy – see Chapter 4.

The components of the above taxonomy are interrelated and dependent [6]. For instance, in order to define a target, it is necessary to know the task and the desired audience. In Figure 2.3, each major taxon is dependent on those above it. The taxonomy presented in this section is used to discuss the elements of the developed game as in Chapter 3. The taxa of task, audience and target together encapsulate the specific application of the software visualisation tool. There are many possible applications, and a few of these are discussed in the following section.

2.3.3 Applications and Examples of Software Visualisation

Unlike many other engineering pursuits, software development does not result in an easily tangible or visible artefact [9]. Thus, as mentioned in Section 2.3.1, the primary reason for visualising software is to aid comprehension by creating a tangible rendition of the software data. There are many aspects of software systems that may be visualised in order to aid this comprehension. Although this research focuses on the problem domain of software education, software visualisation tools are used extensively in industry, as discussed below.

Examples in Industry: Contemporary software projects are typically much larger in

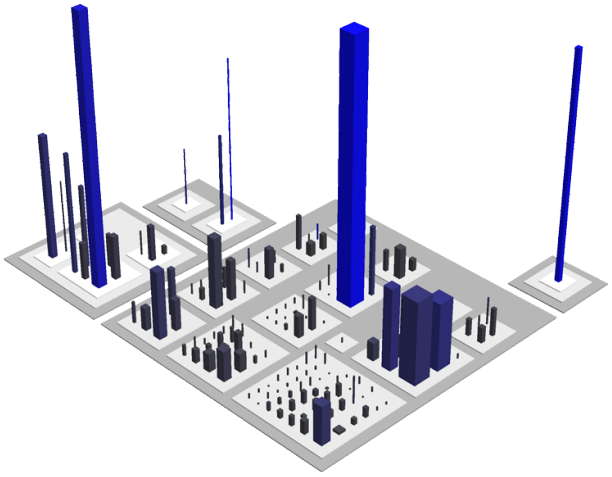


Figure 2.4: A *Codecity* visualisation showing classes as buildings in a virtual city [21].

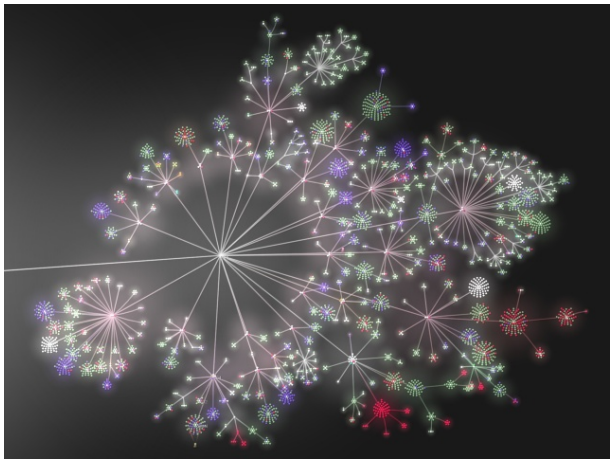


Figure 2.5: A *Gource* visualisation of the `Linux` kernel development [22].

scope – and involve the interaction between much larger teams of developers – than those of the past [18, 17]. There is thus the need for the continuous development of verification and validation tools for software projects. Software visualisation tools assist developers in the assessment of products and act as facilitators in decision-making processes [19, 20]. Software visualisation tools afford developers:

- comprehension of large software architectures;
- comprehension of legacy systems for re-engineering;
- design reasoning by visualising static code structure;
- debugging by visualising runtime behaviour;
- performance tuning by code profiling;
- a view of code evolution and developer contribution.

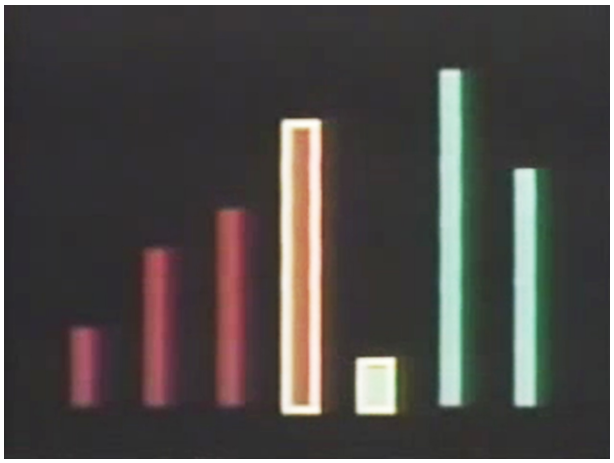
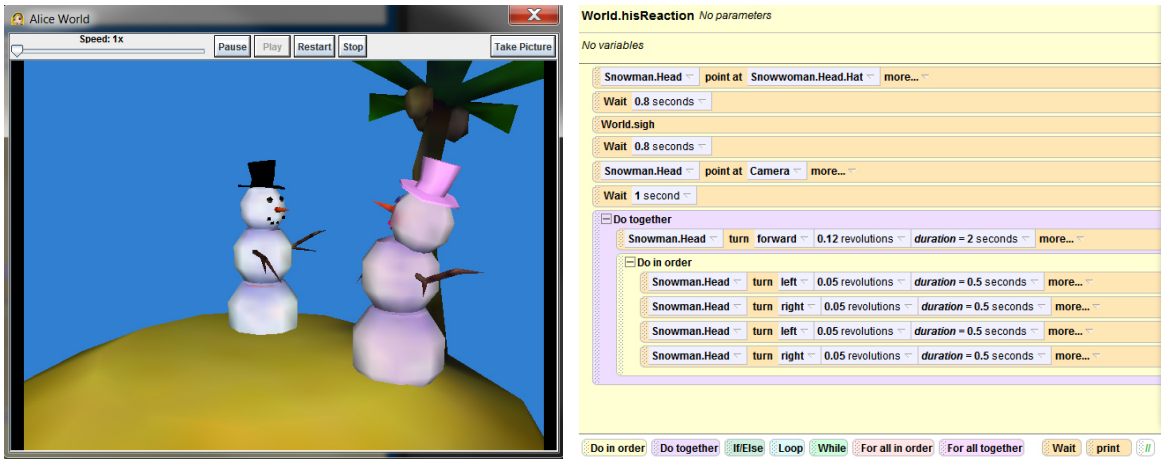


Figure 2.6: The film *Sorting out Sorting* visualises the run-time behaviour of various sorting algorithms [8].

Software visualisation may be used to visualise large software architectures, thus enabling easier maintenance and allowing developers new to a team to acquaint themselves with the system [9, 23, 24, 25]. An example of such a tool is *CodeCity* which enables the visualisation of classes, packages and namespaces within a large software project, allowing for design re-engineering and rapid comprehension [21]. Each class, as in Figure 2.4 is depicted as a virtual building in a city; the width is proportional to the number of data members and the height proportional to the number of member functions. Tools such as *Imagix4D* allow for a lower-level view of source code and includes views of class design and data flow [26]. Software visualisation may also be used to depict software projects' evolution and developer contribution [9]. The software visualisation tool *Gource*, for instance, allows for the visualisation of the evolution of a code base with time, and includes details of developer contribution [22]. The image in Figure 2.5 is a snapshot from an animation of the Linux kernel development.

Examples in Software Pedagogy: Software development and computer science students find it easiest to grasp what their program is doing if they are able to visualise what is happening [7]. Traditionally, this visualisation takes the form of a chalkboard sketch. As mentioned in Section 2.3.1, the primary purpose of software visualisation tools is to facilitate the comprehension of software systems. Software visualisation tools are thus not only applicable to software development professionals, but can be used to facilitate a student's learning of software and computer programming.

Software visualisation techniques have been applied to the realm of academic teaching since the late nineteen-seventies [5, 8]. *Sorting out Sorting*, for instance, a thirty-minute



(a) A scene in *Alice* **(b)** The coding interface in *Alice*

Figure 2.7: The tool *Alice* incorporates both software visualisation and visual programming elements [30].

film developed in the late seventies, has been successfully used to teach principles of sorting algorithms to school and university students [8]. The film's success is attributed to its use of colour, sound and animation to support a voice-over describing the basis of the algorithms [8]. The film, as shown in Figure 2.6, portrays sorting algorithms in a much more cognitively efficient (the film covers in thirty minutes what would take much longer through traditional lecturing) and understandable way than any chalkboard sketch can achieve [8]. Cognitive aspects such as synchronised sound effects, consistent, meaningful use of colour, and the portrayal of only necessary information also contribute to the film's success [8]. The film is an example of how a different medium (cinematography, in this case) may be used for the visualisation of software.

Interactive software visualisation applications may also be used in the teaching of software. Because of their ability to incorporate human interaction (see Section 2.3.1), software visualisation tools are used to assist exploratory learning, a vital part of a student's education in software [7]. Such tools as *Dr Java* [27], *LOGO* [28] and *Karel J Robot* [29] visualise portions of software and serve as open-ended learning environments which enable students to learn at their own pace, and encourage self-motivated learning [7]. It is important to note that many of these tools provide both software visualisation and visual programming components.

Alice (shown in Figure 2.7), developed at Carnegie Mellon University, is another example of a visual programming and software visualisation environment [30, 32]. It is designed to be a prospective computer science student's first exposure to programming, and allows students to create simple scripted scenes in a virtual world [30, 32]. Students are given a collection of 3D

(a) A scene in *Robomind* **(b)** The coding interface in *Robomind*

Figure 2.8: *Robomind* is both a visual programming and software visualisation environment [31].

characters and scenery with which they can compose a scene. Students then programme how these characters interact and behave through drag-and-drop commands. The tool is presented as a gender-neutral and highly visual way of representing and learning software. Students are able to visualise the effect of their code by viewing the scene and seeing how the characters behave. The drag-and-drop commands also mean that students can grasp the concepts of programming (such as looping, for instance), without having to worry about the syntax behind these commands. This further decreases the barrier to entry for learning programming. Despite the interface's simplicity, *Alice* enables students to learn basic programming concepts such as logical statements, iteration, functional decomposition, parameters and variables. Since every object in the virtual world is represented by an object in the scripting interface, students also gain a very concrete understanding of object-oriented programming.

Similarly to *Alice*, *Scratch* – developed by the MIT Media Lab – provides a simple visual programming environment that allows students to drag-and-drop lines of code to create simple games, interactive digital artworks and presentations [33, 34]. This application allows students to piece together lines of code (which are visualised as blocks that fit together) in order to programme the behaviour of objects on the screen. *Scratch* provides a highly intuitive and visual environment for the learning of code flow control, variables and iteration.

Robomind is another example of a pedagogical tool that combines software visualisation and visual programming [31]. This tool, which is shown in Figure 2.8, is similar in task and audience to *Alice*. It is also intended to be a young student's first exposure to computer programming. Robomind provides users with a virtual robot and environment [31]. The robot

(a) The game attempts to provide the entertainment value of other action and tactical games.

(b) The game also provides an environment for training.

Figure 2.9: Screenshots from the serious game *America's Army*.

can be programmed with a simplistic scripting language (see Figure 2.7b). Whilst there are no actual goals in the environment, students can programme the robot to solve tasks such as following a line or navigating a maze.

Tools such as *Alice* and *Robomind* bear marked similarities with digital games. For instance, they both present highly interactive and visual environments in which users are able to experiment in so-called "free play". Indeed, such tools may be considered game-based learning tools or educational games. The concepts of digital games with focus on serious and educational games is thus investigated in the following section.

2.4 Digital Games

This section provides a very brief review of literature about the fields of serious games, educational games and gamification. The use of games as a form of software visualisation is examined. The benefits of games as a means of education are presented.

2.4.1 Serious Games

A 'serious game' is a game that goes beyond mere entertainment to provide a 'serious' message. This message can take the form of a healthcare message, a governmental or corporate agenda, advertising (so-called 'advergaming'), a socio-political commentary, or could represent a simulation, training or educational exercise [35]. For instance, the first serious game, *America's Army*, released by the United States Military in 2002, is a free-to-play simulation

game representing the actual training exercises and scenarios encountered by recruits in the army [35]. As shown in Figure 2.9, the game aims to both entertain by emulating other first-person action games and provide a simulation tool for new recruits and people considering a career in the military. It is also considered to be the U.S. army's most effective recruitment programme [35]. Such games have to be didactic; they have to both deliver the intended message in a transferable way, and they have to be entertaining, and there is much debate about whether this is actually possible in the gaming medium, or which aspect should take precedence [35, 36]. Serious games are generally intended to have some positive impact on the player, or to develop players' skills. A subset of these games is the educational game, discussed in the following section.

2.4.2 Educational Games

> “Frankly, most existing edutainment products combine the entertainment value of a bad lecture with the educational value of a bad game. But what if we could turn that around?”
>
> — *Squire and Jenkins [37]*

Educational games are a subset of serious games focussing on simulation or the development of player skills. The term “educational game” is relatively new, and related to the term “edutainment”, which was popularised in the 1990's. However, early edutainment games are highly criticised for their boring nature, and reliance on rote learning and memorisation, as the above quote exemplifies [35, 36]. As the field has matured, educational games are beginning to move toward more higher-level skills such as the comprehension, application, analysis, synthesis and evaluation taxa of Bloom's taxonomy of learning [38]. To do this, these games must focus on player engagement, motivation and the role-playing capabilities of games [35]. Often, it is not feasible due to safety, cost, or time concerns to provide training scenarios for the development of skills [35]. Games may thus be used as a alternative medium for delivering this simulation, *America's Army* being a prime example of this.

Open-ended learning environments such as *Alice* use highly visual and highly interactive interfaces to lessen a student's learning curve of software in a fun and enjoyable way. Video games are also fun, visual and interactive, and lend themselves to goal-based exploratory learning. The benefits of games as an educational medium are discussed in Section 2.4.5. Games and game-based learning environments are being employed in the teaching of computer

programming. For instance, the popular game *Minecraft* is being used as a platform to teach practical artificial intelligence through game 'mods' [39]. Examples of online multiplayer games that teach programming include *.Net Terrarium* [40], *RoboCode* [41] and *DroidBattles* [42]; these games allow players to program virtual creatures that populate a virtual world and compete with other player's creations [40, 41]. An online interactive tutorial for Ruby on Rails, *Rails for Zombies* makes use of game-like achievements within the context of instructional screencasts and a hands-on editing window [43]. *SimSE* is a virtual world centred around the education of software engineering lifecycles [44]. Players are able to assume the role of a software developer in the simulation, and role-play to see the effect that decisions have on the budget and life-time of the virtual project [44]. *VIMAdventures* is an adventure role-playing game which teaches players the keyboard shortcut commands of the *VIM* text editor, an important tool for professional software developers [45]. An example of a non-digital collection of games or activities that enable the learning of programming concepts is *CS Unplugged* [46]. The activities (which include learning about binary numbers and information theory) are non-digital games designed for secondary education.

2.4.3 Gamification

The term "gamification" is used when game design principles and game elements are applied to non-gaming applications [11]. Examples include social and healthcare applications, such as *Zombies, Run!*, which aims to promote exercise. The game consists of audio files that a player listens to while jogging. The audio indicates when a player should run (to avoid zombies), or when they can walk. A player gains points for successfully avoiding zombies, which he or she can use to build and improve a 'base' in the game [47]. The *Speed Camera Lottery* is an example of gamification applied to the problem of traffic speed monitoring and control [11, 48]. In this application, traffic control cameras are used to identify vehicles that are speeding, and those that are under the speed limit. Whilst the drivers of speeding vehicles are fined as normal, the drivers who obey the speed limit are entered into a lottery with the chance of monetary reward. The prize is funded from the speeding fines. This example demonstrates the ability of gamification to motivate desired behaviour by providing a tangible, extrinsic reward. Gamification has also been used in the classroom; *The Multiplayer Classroom* is an application of role-playing game elements to a game design course, in which marks are replaced with 'experience points' and groups with 'guilds' [49, 50]. This further demonstrates the ability of gamification to change human behaviour and to enhance buy-in.

2.4.4 Games as a Medium for Software Visualisation

Examining the taxonomy of software visualisation tools in Section 2.3.2, it is possible to see that video games encompass the representation and medium taxa. The task, audience and target may remain the same, but choosing to use a game as a medium of software visualisation will greatly impact both representation and medium. For instance, the method of abstraction in *Alice* involves having all program objects represented by objects in the virtual world. All code in *Alice* is highly correlated with the game world. Games can also offer a high degree of interactivity: for instance, the temporal control available in *RoboMind*. The presentation style of games (user interface, graphics, sound, animation) is another key aspect, and links directly to the medium of software visualisation systems. This answers the first research question; "where do digital games fit within a taxonomy of software visualisation techniques?".

2.4.5 Why Games Can be a Good Pedagogical Medium

There is much criticism of educational and serious games; many researchers believe that the entertainment aspects of the games are nullified by the educational message, and that the game elements devalue the educational process [3, 35, 36]. However, most researchers agree that games and games-based learning should not be seen as a silver bullet or a replacement for traditional lectures and tuition [3]. However, many researchers believe that games may be used to augment existing learning methods and address many of the underlying issues of the declining interest in science and engineering – such as student apathy [3]. A review of the literature reveals unique benefits that games may offer over other pedagogical tools. Games:

- can reach an incredibly wide demographic of people [3];
- afford inquiry-based learning;
- provide visual, interactive analogues and mirror real-world systems [36];
- challenge players at the edge of their growing realm of competence [51];
- reward failure and repetition [35];
- are fun, engaging and self-motivational [36];
- stimulate interest.

Games have the ability to bypass the – often obstructive – nature of traditional teaching institutions and reach an incredibly wide demography of people. Especially with the rise of app-based and mobile games, many more people – dubbed "native speakers in the language of digital media" – are becoming increasingly familiar with this medium [35]. Games also allow

for interaction; indeed, they require active participation from the players, and thus provide a perfect platform for exploratory, goal-based learning [36, 52]. Whereas with traditional teaching methods, the student is a passive receptor of information, exercises such as games force a student to take a more active approach in engaging with the material. Games thus afford inquiry-based (as opposed to instructional-based) learning. Games allow not only for interaction, but visualisation as well. Games are ideal as education tools because they are simulations or mirrors of real-world systems [36]. They are thus inherently good at portraying the dynamics of any given system – be it economic, physical or software – and showing the student the cause-and-effect relationships within this system [36]. When examining games as a medium for software education, the interactivity in tools such as *Alice* and *RoboMind* is analogous to the visual programming aspects of these tools, and the visualisation is analogous to the software visualisation techniques employed. Games also challenge players at "the outer and growing edge" of a player's competence [51]. In other words, they challenge players but never overwhelm them, and this is an incredibly powerful model for educational tools. In traditional teaching, failure is often perceived to be punished. However, in games failure is a part of the learning process: a powerful shift in paradigm in pedagogy [35]. Games, by definition, have the capability to be more fun than other activities [52]. Good games have innate reward systems, and these can be used to provide engaging and self-motivating ways of stimulating interest around a topic such as computer programming.

2.5 Summary

This chapter provides a brief review of relevant literature surrounding the research areas in this investigation. The salient points of this chapter are:

- software visualisation is the rendering of data about a software system in order to augment human understanding and efficient use of computer software;
- existing taxonomies of software visualisation tools define six main taxa: task, audience, target, representation, medium and effectiveness;
- the taxa of representation and medium are of particular importance when considering digital games as software visualisation tools;
- many pedagogical software visualisation tools bare resemblance to digital games;
- such tools often combine software visualisation and visual programming techniques;
- serious games are games with a purpose beyond that of mere entertainment;

- educational games offer simulation and training;
- games provide good tools for pedagogy because they provide a simulation of their problem domain, afford inquiry-based, exploratory learning, and are self-motivating.

This chapter provides the reader with background information necessary to understand the design and implementation methods used to develop the game. These methods are described in detail in the following chapter.

CHAPTER 3

Implementation Methodology

3.1 Introduction

This chapter focuses on the design decisions surrounding the iterative development of the educational tool. The taxonomy of software visualisation tools discussed in Section 2.3.2 is used as a basis to structure this discussion. Specifically, this chapter provides:

- a discussion of the tool's intended task and audience (in Section 3.2);
- an explanation of the target – the particular aspect of computer programming that is visualised (in Section 3.3);
- an explanation of the medium and representation of the tool in terms of the game design and features (in Section 3.4).

3.2 Task and Audience

The developed game is intended to augment the teaching of a small scope of the Software Development I course offered to students of Biomedical, Information and Electrical Engineering at the University of the Witwatersrand. Its task is to introduce a few programming concepts (the scope of the game is discussed further in Section 3.3) and to serve as a first exposure to computer programming for second-year university students. It is intended to lessen the learning curve of the covered concepts. However, the game is equally suitable for younger individuals such as primary and high-school students. Since the game is a Windows and

Mac-based PC game, it is assumed that the audience has basic computer literacy, but no knowledge of computer programming; as is discussed in Chapter 4, the game is not well suited to individuals who already have basic computer programming skills. Furthermore, it is assumed that the audience is English-speaking since the instructive elements of the game were written in English (see Section 3.4.3). The game is intended to replace a laboratory exercise, and take place during a tutorial session. The intended time on task is thus roughly 45 minutes.

3.3 Target

As mentioned, the game is designed to replace a laboratory exercise in the Software Development I course. Specifically, the second laboratory exercise was used to decide the skills and concepts that the game would teach. In this exercise, students are required to write a simple Body Mass Index (BMI) calculator. The programme allows a user to insert their height and weight, calculates their BMI and outputs a message corresponding to different value ranges. An example of a solution to the lab may be seen in Listing 3.1 below. As may be seen, the laboratory covers the concept of code flow control, the *if, else* and *else if* statements, and the *&&* logical operator. These concepts are thus used as the learning goals of the game. The game aims to provide a highly visual, interactive equivalent of this laboratory. The manner in which this was done is described in the following section.

Listing 3.1: An example solution to the laboratory exercise

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     double w;
8     double h;
9     double BMI;
10
11    cout << "Insert your height in metres: " << endl;
12    cin >> h;
```

```
13    cout << "Insert your weight in kgs: " << endl;
14    cin >> w;
15    BMI = w / (h * h);
16
17    cout << BMI << endl;
18
19    if (BMI < 18.5) {
20    cout << "Underweight: Health risks increased" << endl;
21    }
22    else if (BMI >= 18.5 && BMI <= 24.9) {
23      cout << "Normal weight: No health risks" << endl;
24    }
25    else if (BMI >= 25.0 && BMI <= 29.9) {
26      cout << "Overweight: Health risks increased" << endl;
27    }
28    else if (BMI >= 30.0 && BMI <= 34.9) {
29      cout << "Obese: Health risks high" << endl;
30    }
31    else if (BMI >= 35.0 && BMI <= 39.9) {
32      cout << "Obese: Health risks very high" << endl;
33    }
34    else {
35      cout << "Severe obesity: Health risks extremely high" << endl;
36    }
37    return 0;
38 }
```

3.4 Representation and Medium

The game is a PC internet browser-based game, developed with the *Unity3D* engine, that works on both Windows and Mac machines. The genre of the game is puzzle-solving and the game requires players to type code in order to solve various puzzles, each presented in a different game level. Players have to complete a level in order to proceed to the next, and

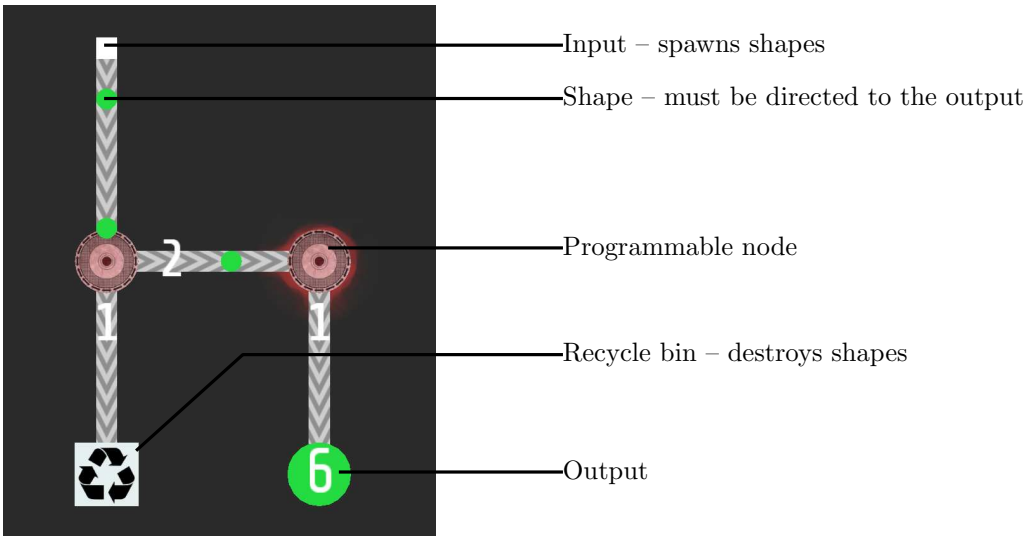
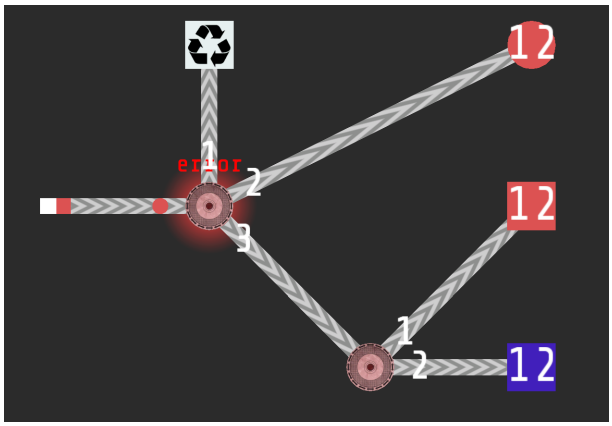


Figure 3.1: An annotated screenshot of the first iteration, showing important game elements

levels become progressively more difficult. The typing of code was used as opposed to a drag-and-drop interface (such as in *LightBot* and *Robozzle*) so as to better afford transference of skills into actual text-based programming. This is discussed in more detail in Chapter 4. To describe the basis of the game, the first iteration is examined in the following subsection.

3.4.1 Iteration 1: *Conveyor*

The first iteration of the game is called *Conveyor*. To visualise the branching nature of conditional statements, the visual metaphor of a branching conveyor belt system is used – see Figure 3.1, which shows an annotated example of a level. Players are required to direct objects, that move along the conveyor belts, to the correct output. The objects can follow different paths, and this is an abstraction of conditional statements causing different outcomes based on input criteria (such as the BMI in the above example). Players achieve this by writing the code to programme the nodes in the virtual conveyor belt system. The objects (abstractions of variables or data in code) have different properties such as shape – square, circular, or triangular – and colour – red, green or blue. A level may have multiple outputs which each expect a different number of objects of differing shape and colour. The player is able to click on nodes to type their code to query the colour and shape of the object and direct them to the correct outputs (see Figure 3.2 for an example of a level and code used to solve it). The player essentially forms a rudimentary sorting algorithm, which is visualised in real-time. Objects are represented through both different shapes and colours, and animation is used to visualise the sorting algorithm that the player creates. The game may thus be seen



(a) Screenshot

Node 1 (top left)

```
if(shapeIsSquare())
     outputTo(3);
else if (shapeIsRed())
     outputTo(2);
else
     outputTo(1);
```

Node 2 (bottom right)

```
if(shapeIsRed())
     outputTo(1);
else
     outputTo(2);
```

(b) Example solution

Figure 3.2: A level of the game during iteration 1, and an example of a possible solution to the level

as a form of algorithm visualisation and a basic visual programming environment.

As may be seen in Figure 3.1, the representation is deliberately very minimalist and abstract. A red 'error' message is displayed and a sound played to indicate errors in the code. The mere presence of an error is shown, but not any useful information that could lead to its resolution. This was improved in the second iteration, as discussed in the following section. Sound is also played when a shape reaches the correct output. In terms of navigation, levels are shown in their entirety on the screen; players are not required to navigate the space. This conscious decision allows players to focus on learning to programme rather than spending time with game interface and navigation. When a player clicks on a programmable node, a window opens that allows a player to use the keyboard to type his/her code. The code that a player types is evaluated at runtime. Simple on-screen text hints guide players and give examples of the available functions and flow control statements (refer to Section 3.4.3 for more information). Each time the game instructs a player of a new syntax or function, a reference to the player's 'codebook' is added. The 'codebook' is accessible at any time and serves as a quick reference guide for players. As discussed further in Section 4.7, it was observed that this quick reference system was vital for the game as players would often forget syntax or keywords. Iteration 1 consisted of 9 levels.

3.4.2 Iterations 2 and 3: *if(traffic)else{win}*

Following testing and feedback on iteration 1, the game was refined. Although the main idea of iteration 1 persisted, the game – renamed *if(traffic)else{win}* – was made less abstract

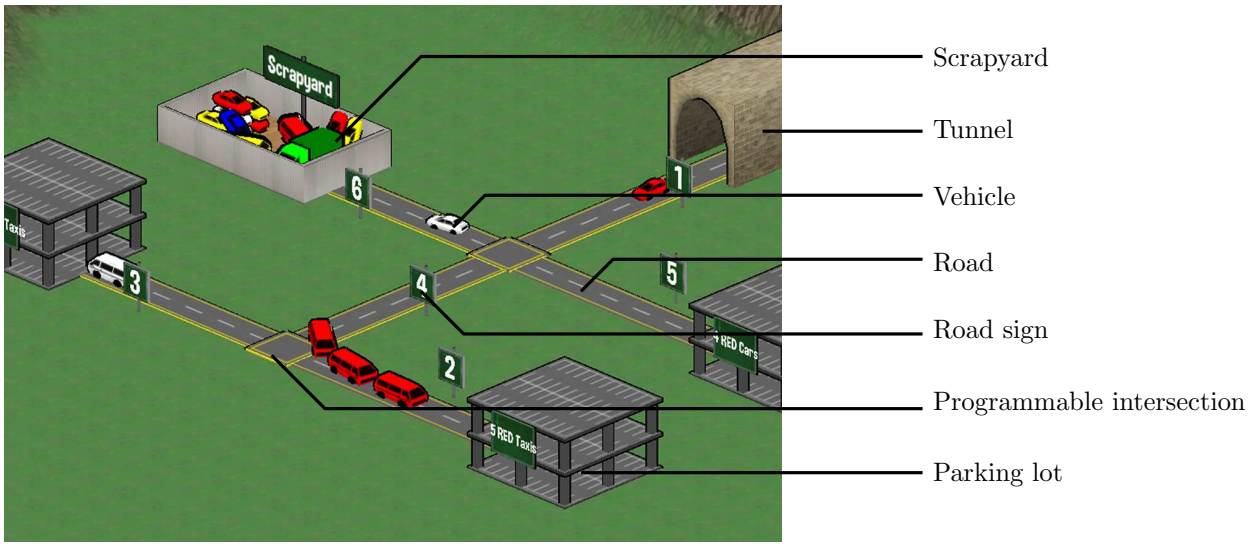
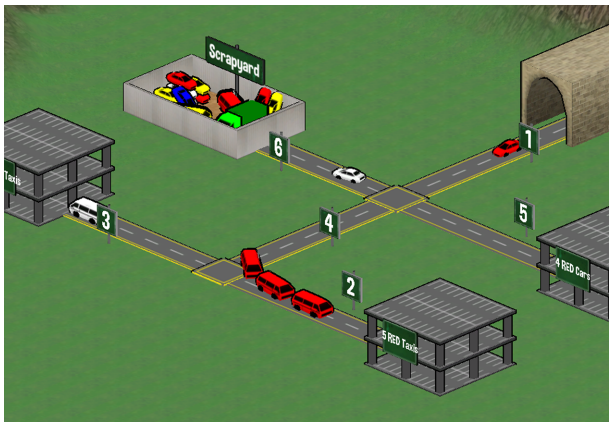


Figure 3.3: An annotated screenshot of iteration 2 and 3, showing the main game elements

by the introduction of a theme: traffic. Figure 3.3 annotates the main game elements. As may be seen, the conveyor belt system of iteration 1 is replaced by a road network; the abstract objects by vehicles; and the output bins by parking lots. This theme was chosen for three main reasons. Firstly, a concrete theme provides a less abstract game, and thus a more engaging and relatable experience to players. Secondly, it is understandable by a wider demographic than a more esoteric theme. Thirdly, it makes diegetic (narrative) sense for players to programme traffic lights to direct traffic. In place of the objects, players are now required to direct vehicles of different colour and type (cars, taxis and trucks). As may be seen in Figure 3.3 the graphics were improved. The game is presented in an isometric 3D view. This was chosen as it is more interesting than the top-down view of iteration 1, but is not as complicated as full 3D for people unfamiliar with games. It also minimises the interface overhead and allows players to concentrate on the programming aspect of the game. Levels are now able to stretch beyond one screen, and the mouse is used to navigate through the level. The instructive aspect of the game was also revised. At the beginning of each level, a help screen interface is shown that provides hints and example code that can be used to solve the level. The help screens persist as players progress, so they are still able to refer back to previously learnt knowledge if they forget something. As may be seen in Figure 3.4, which shows the equivalent level to Figure 3.2, the programming syntax and language was greatly simplified in this iteration. Iteration 2 was made longer than the first with 15 levels. Following critical feedback about the length of the game (see Chapter 4), iteration 3 consisted of 12 levels, and had only minor improvements from iteration 2.



(a) Screenshot

Intersection 1 (top right)

```
if(taxi())
     drive(4);
else if (red())
     drive(5);
else
     drive(6);
```

Intersection 2 (bottom left)

```
if(red())
     drive(2);
else
     drive(3);
```

(b) Example solution

Figure 3.4: A level of the game during iterations 2 and 3, and an example of a possible solution to the level

3.4.3 Instructive Game Elements

The game presents a mixture of instructive and exploratory learning techniques. By its nature, the game allows a player freedom to experiment with different code structures, and so affords free-play and exploratory learning. However, the game also has to be instructive to give novice programmers certain information required to complete the games puzzles. Such information is imparted to the player by the use of help screens, an example of which is shown in Figure 3.5. These are displayed before certain levels and give example code, alternative ways of completing certain outcomes, and English descriptions of source code and hints. They also allow a player

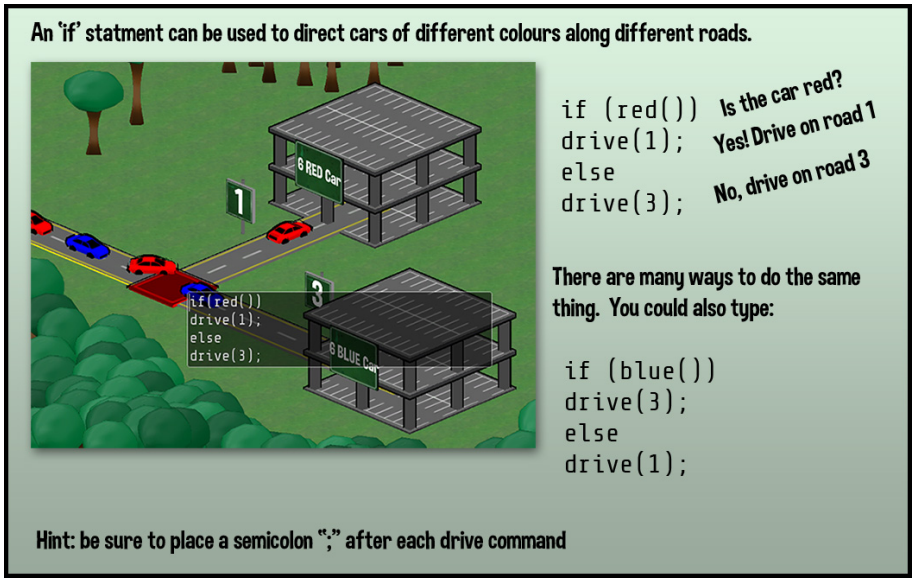


Figure 3.5: An example of an instructive help screen in the game

to see the correct syntax for various control statements. During prototyping, it was noted that it is imperative to allow players to refer to this knowledge as they please; often, players would forget how to write certain statements in later levels. Thus, players are able to click on a "Help" button to display all the help screens that they have seen so far. It is through these help screens that an instructive element of the game is portrayed, and through experimentation with typing code and solving puzzles that the players engage with the material.

3.5 Summary

This chapter has provided a discussion of the implementation details of the developed game. In summary:

- the game aims at reducing the learning curve of a student's exposure to conditional statements and learning programming syntax;
- the game's intended audience is first- and second-year university students, but could be used effectively by any novice programmer;
- the final iteration of the game is a puzzle game that requires players to direct traffic to the correct parking lots;
- graphics, animation and sound are used to visualise basic software conditional statements;
- the game provides a visual programming environment in which students type code that attempts to solve a designed puzzle;
- the game consists of levels of increasing difficulty and introduces concepts slowly;
- the game allows for instructive learning by providing help screens and a quick reference guide.

This chapter has provided a description of the task, audience, target, representation and medium taxa of the developed tool. The final taxon, effectiveness – in terms of the design, development and results collected from various measurement systems and experiments performed – is discussed in the following chapter.

CHAPTER 4

Testing Methodology and Analysis of Results

4.1 Introduction

In Chapter 2, a taxonomy of software visualisation tools was described. The taxon of effectiveness attempts to empirically evaluate the degree to which the requirements of the software visualisation tool have been fulfilled. A few empirical techniques and experiments have been performed on the developed game, as discussed in this chapter. Specifically, this chapter provides:

- an overview of the experimental techniques and measurement systems developed (in Section 4.2);
- a description of the in-game analytics system designed (in Section 4.3);
- a discussion of the before-and-after software skills quiz that was used to judge the educational value of the game (in Section 4.4);
- details about the survey that was used to collect qualitative user experience data (in Section 4.5);
- the presentation and analysis of results collected (in Section 4.6);
- a discussion of the lessons learnt and observations made during development and testing (in Section 4.7).

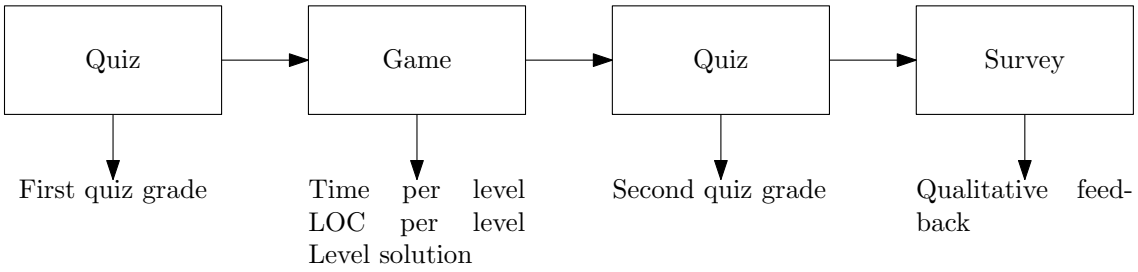


Figure 4.1: The testing process. The quizzes measured increase in student marks and thus the educational value of the game. The survey was used to collect qualitative user experience data.

4.2 Overview of Experimental Methodology

At the end of each development iteration, the game was tested on a group of university students. Ethical clearance (protocol number H120407) for these tests was granted through the Ethics Committee (Non-Medical Subjects) at the University of the Witwatersrand, and participants signed consent forms before taking part. The participant group was different in each iteration, depending on student availability. Iteration 1, for instance, was tested on a group of 39 students who ranged from first-year to fourth-year Electrical, Information and Biomedical engineers. However, first-year Game Design students – from both engineering and arts streams – comprised the majority of this group. Iteration 2 was tested on a group of 14 Applied Computing students, and the third iteration on a group of 84 first-year Electrical engineers. The tests involved the collection of qualitative and quantitative data from three sources: an integrated analytics system; a survey; and a before-and-after software skills quiz. Participants gathered in a computer laboratory at the University to take part in the experiments. The total time that students spent on the process was around 45 minutes to an hour. This time included time spent on the tests and the survey. The testing process is illustrated in Figure 4.1. The participants first completed the software skills quiz. The quiz is an online multiple-choice test, and the students' marks were recorded and captured on a server (see Section 4.4). Participants then played an online version of the game. While playing, the designed analytics system collected data which was uploaded to a server database (see Section 4.3). Participants then completed the software skills test again, and their second marks were stored. The difference in student marks is assumed to be an indication of the educational value of the game. Finally, participants completed the survey. Because of iterative development, not all tests were used in each iteration, as shown in Table 4.1. Each of the testing procedures is described in more detail in the following subsections, and the results obtained are presented and analysed.

Table 4.1: The testing methods used per iteration of the game

	Analytics collected											
Iteration	**Time per level**	**Number reattempts per level**	**Total number of plays**	**Total play time**	**Player solutions**	**Programming experience**	**Gaming experience**	**Gender**	**Intended degree path**	**Gaming experience**	**Survey**	**Before-and-after test**
1	✓	✓	✓	✓	×	×	×	×	×	×	✓	×
2	✓	✓	✓	✓	✓	×	×	×	×	×	✓	✓
3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

4.3 Analytics System

An in-game, integrated analytics system was used to collect data about how participants played the game. The system (shown in Figure 4.2) consists of the *Unity* game, acting as a client, and a server with a database. The *Unity* analytics module (written in `JavaScript`) passes `HTTP POST` messages to the server-side analytics system (written in `PHP`). This uses a designed database abstraction layer to access and store the information in the database. As shown in Table 4.1, in the first iteration, the analytics system was used to collect and store the times it took players to complete levels, and how many times they reprogrammed a node. The total number of plays and the total play time of the game is also collected. In the second iteration, the number of lines of code used in the solution of a level was stored for each player. The players' actual solutions are also stored, since some levels are solvable in a variety of ways. The third iteration analytics system also collected demographic data about the players such as their gender, their perceived level of programming and gaming experience, and their intended degree path (either electrical engineering, or information engineering). The results of this system are presented and analysed in Section 4.6.

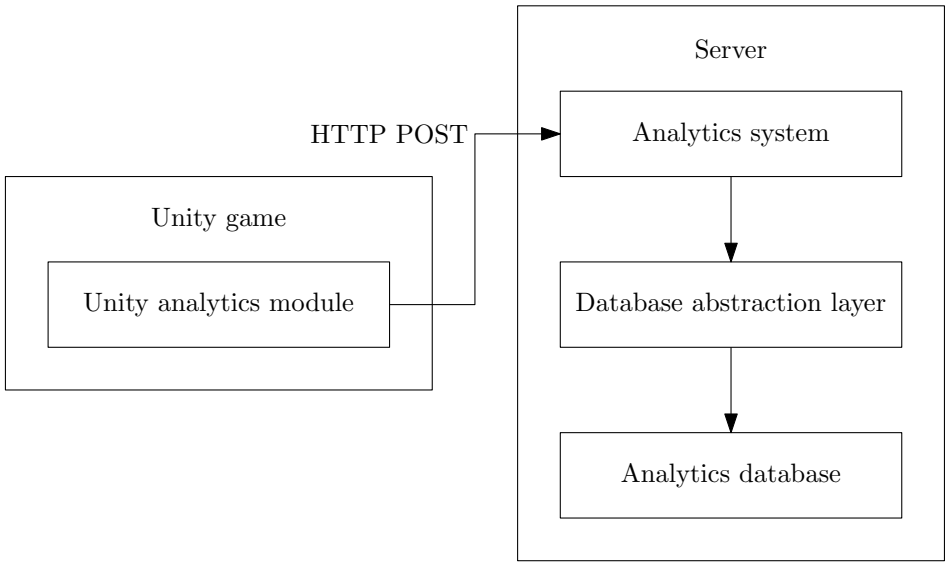


Figure 4.2: High-level design of the analytics system

4.4 Software Skills Quiz

The software skills quiz was designed as a before-and-after test in order to gauge the educational value of the game in teaching syntax and conditional statements. It is vital for an educator to be able to measure how useful the game is in teaching the intended material beyond the scope of the game. In other words, that students do not merely learn the game system, but also learn the intended subject matter (target) in the scope of their traditional education.

The test was created using an existing survey tool, *LimeSurvey*. This allowed the data to be automatically captured and stored in the server database for further analysis. The test consists of 12 multiple-choice questions covering concepts taught in the game. In order to reduce bias in either test (due to differences in question difficulty), a single test was used. However, to prevent participants learning the solutions, and to provide more meaningful results, the 12 questions were randomly selected from a pool of 24. Questions were also asked in a number of categories corresponding to different difficulty levels. The full software skills test may be found in Appendix D. The marks of students before and after playing the game were calculated and the results are presented and discussed in Section 4.6.

4.5 Survey

A survey was administered to participants to judge the more qualitative aspects of the developed game, and to profile the audience. Demographic information such as the participants' perceived level of programming ability and their past exposure to gaming was collected.

Participants were also asked whether or not they feel games are useful in educational contexts, and whether or not they would recommend this game to people learning how to programme. A four-point Likert scale was used to quantify the responses of participants when asked how useful they felt the game was at teaching programming, how easy it was to learn the tool (learnability) and how easy it is to recall what they learnt (memorability) [18]. The scale also quantifies the entertainment value of the game, the players' perception of graphical and aural quality and the quality of the user interface. Participants were also given the opportunity to give open-ended feedback in terms of what they learnt, what they felt was well executed, what they felt was poorly executed, and any additional comments or suggestions. The full survey can be found in Appendix E, and the results of the survey are presented in the following section.

4.6 Results and Discussion

4.6.1 Software Quiz Results

As mentioned, the software skills test was only administered in the later two iterations. The students' average mark and standard deviation in marks are shown in Table 4.2. As may be seen, the game only improved the applied computing students' marks by a statistically insignificant 1%. However, this increase was on an already good mark of 83%. The applied computing students had, at the time of testing, already completed a university-level course in computer programming. The electrical engineering students, on the other hand, had no previous exposure to programming at university, and their results increased by 8% on an average mark of 54%. It may thus be seen that the game is better suited to less experienced participants. The variability of marks (standard deviation) decreased by 4% for Iteration 2, and remained constant in Iteration 3. Due to the larger sample size in Iteration 3, this change is more statistically relevant. Therefore, the game had no notable impact on the variability of student marks. The standard deviation is also high (25%), since some students did markedly better – and others, markedly worse – than the mean mark. As mentioned, the students only had a time-on-task of under 45 minutes. The benefits of playing the game multiple times, and the effect of a longer time-on-task is recommended for future investigation (see Section 5.2). This test reveals the educational value of the game in teaching the intended target. This is important for educators since the game can be compared to – and quantified against – other media of delivery, such as a lecture or tutorial.

Table 4.2: Results of before-and-after software skills testing

Iteration Number	**Student group**	**Mean mark before playing game (and standard deviation)**	**Mean Mark after playing game (and standard deviation)**	**Increase in mark**
Iteration 2	Applied Computing	83% ($\sigma = 13\%$)	84% ($\sigma = 8\%$)	1%
Iteration 3	Electrical Engineering	54% ($\sigma = 25\%$)	62% ($\sigma = 25\%$)	8%

4.6.2 Level Times and Learning Curve

When a player completes a level, the analytics system stores the time taken to solve it. The average time taken for each level over all three iterations is shown in Figure 4.3. As may be seen, the applied computing students (those with greater skill in programming) completed the game the fastest, with an average level time of 1 minute, 30 seconds. The game design and electrical engineers completed the game with average level times of 2 minutes, 27 seconds and 2 minutes, 36 seconds, respectively. As previously mentioned, each iteration of the game had a different number of levels. The x-axis of Figure 4.3 is thus indicative of the level number in iteration 2 – the longest of the three games. The other two iteration data points on this graph are shown in relation to the equivalent levels in iteration 2. For example, level 9 in iteration 2 introduces the *else if* control, and the data points shown in Figure 4.3 are in relation to this, even if these levels were not ninth in their respective iterations. With this in mind, it is worth noting the short time taken by the game design students in solving level 8 as compared to the other groups. This level was actually placed second in the game design students' iteration, and was solved much faster than the other two participant groups, for whom it appeared later. This may indicate that players necessarily expect later levels to be more complicated, and may indicate poor problem analysis by the students. In future, the game could be redesigned so that the levels do not necessarily get progressively more difficult to see if students attempt complicated solutions to easy problems.

For this research, it may be assumed that the time spent on a level is an indication of its perceived level of difficulty. Figure 4.3 may thus be seen as a representation of the learning

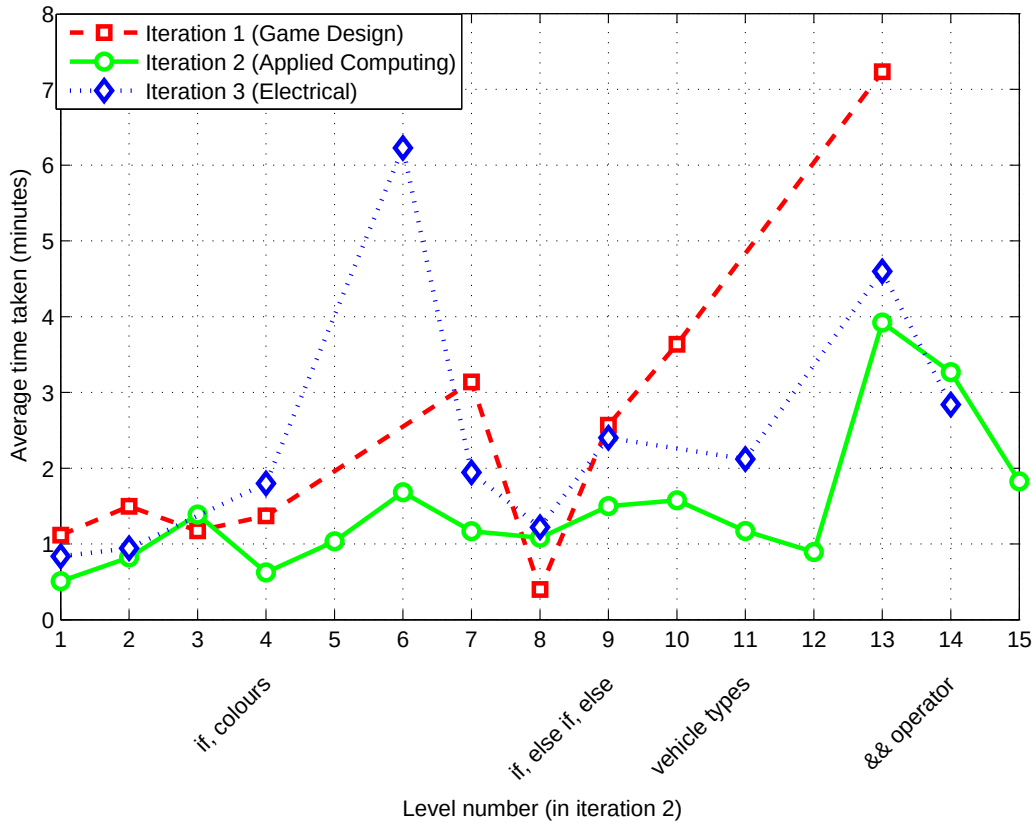


Figure 4.3: Average level times over each iteration

curve of the game. Apart from level 8 (mentioned above), the learning curve of iteration 1 is fairly linear. This resulted in many students not completing the game as it introduced new, more complicated, puzzles too quickly. A longer game (such as iteration 2), allows the designer and educators better control over the learning curve. For iteration 2 in Figure 4.3, a rise and fall of level difficulty may be noted. This corresponds to the introduction of new concepts and the players' learning and adopting these concepts in future, similar problems. However, many students (50% of the audience) felt that this iteration was too lengthy, and became bored of its repetition (see Section 4.6.5). It is thus necessary for an educator or the game designer to obtain a balance between these two factors, especially when dealing with different audiences. Hence, the measurement of the game's learning curve (in this case, done through the proxy of level time) is also important from an educator's perspective.

For iteration 2, there is a sharp increase in the time taken by Electrical Engineering students on level 6. This is due to the nature of the puzzle presented, which required the writing of four cascaded *if, else* statements. The collection of this metric thus identifies this level as being poorly designed, and this illustrates how the collected metrics can be used to

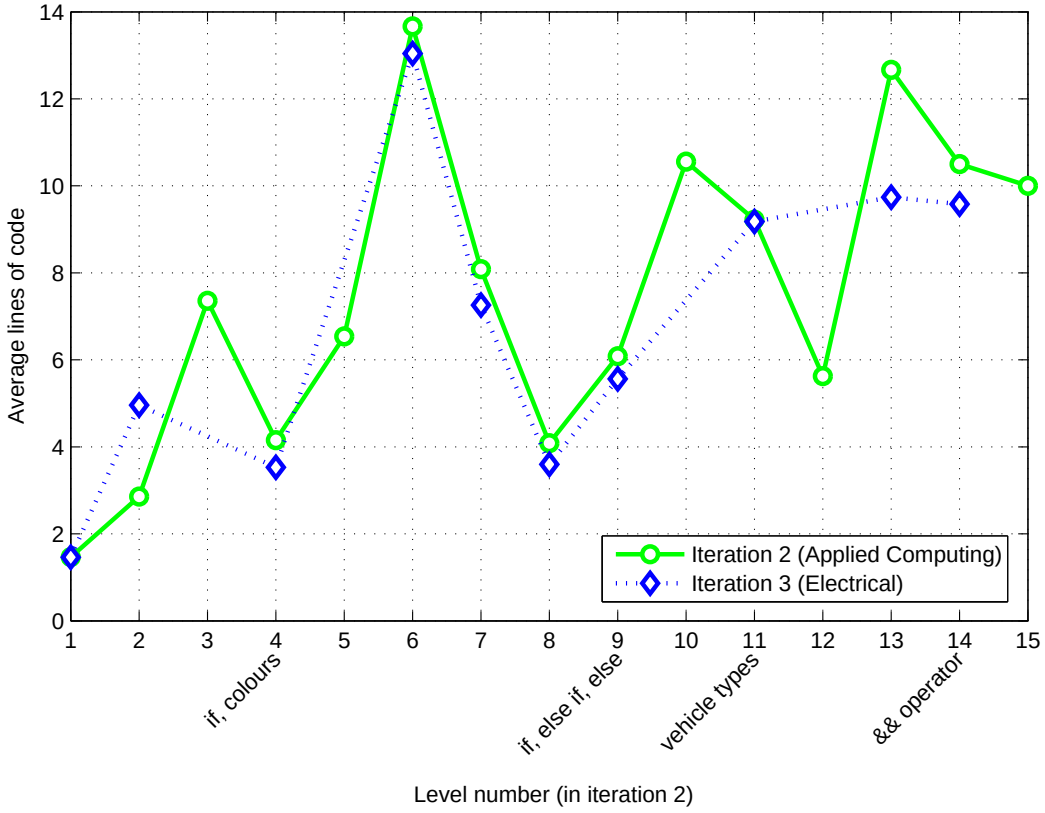


Figure 4.4: Average lines of code used per level over each iteration

inform further redesign of the game. Students also spent more time solving level 13. This level, however, was more difficult than previous levels. It required students to understand that the order of an *if, else if, else* control statement matters. This was never explicitly explained to players, and thus the level proved somewhat more challenging.

Figure 4.3 may also be used by educators to identify levels with which the majority of the class is having difficulty. For instance, the Electrical Engineering class spent significantly longer on level 6 than the other two groups, as shown in this figure. A lecturer or demonstrator could thus use this information to narrow the focus of a lecture or tutorial on the concepts introduced in this level (cascading *if, else* statements, in this case). This data is thus important for educators and affords possible personalised learning.

4.6.3 Lines of Code

Figure 4.4 shows the average lines of code used in the solution of each level in iterations 2 and 3. As may be seen, the number of lines of code used corresponds to the time taken to solve each level. Additionally, both participant groups used a similar number of lines of code

to solve each level. This could indicate that the levels themselves need to be redesigned to allow for a greater variability. However, within the limited scope of the game, this is difficult, but could naturally arise should the game be extended to allow for loops, variables and other programming concepts.

4.6.4 Survey: Likert Scale

The survey feedback was based on a four-point Likert scale that limited answers to very poor, poor, good and very good. This eliminated the option for a student to remain neutral. All responses for each iteration were then averaged, and the results are depicted in Figure 4.5. As may be expected, the applied computing students who had previous exposure to programming rated their programming experience the highest, and the game design students rated their gaming experience higher than the other groups. The applied computing students also rated the usefulness of the tool the lowest, and this corresponds to the 1% increase in their marks (see Section 4.6.1). The electrical and game design students rated the game between good and very good in terms of its usefulness. The applied computing students found the tool easier to learn and recall, possibly because they required less effort to learn the programming aspects of the game with which they were already familiar. The game design students found the game the most fun, and applied computing students the least fun. This is related to the students' relative skill in programming, and, interestingly, the length of the game; according to these results, a shorter game is more enjoyable. This is supported by the fact that 50% of applied computing students, compared to only 14% of the electrical engineering students, felt that the game was too long. Despite the improvement in game graphics from iteration 1 to 2 (as may be seen from Figures 3.3 and 3.1), all three participant groups rated the graphics as good. This strange result is possibly due to participant groups not seeing the other games, and thus not being able to judge the improvement in graphics. No participants listened to the game audio (the laboratory computers did not have speaker systems), but this result (very poor to poor) is used as a control question to indicate that participants were unbiased. 100% of participants said that they feel that games can be educational, and 98% said that they would hypothetically recommend this game to other people learning how to programme.

4.6.5 Survey: Open-Ended Feedback

As mentioned, the survey was also used to collect open-ended feedback from the audience (see Appendix E). The answers to these open-ended questions were examined and categorised into

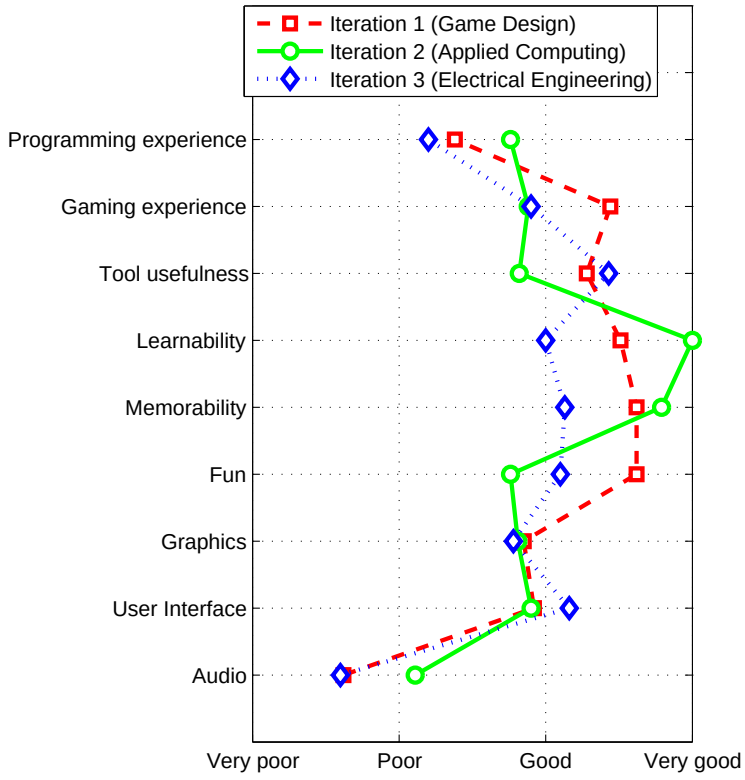
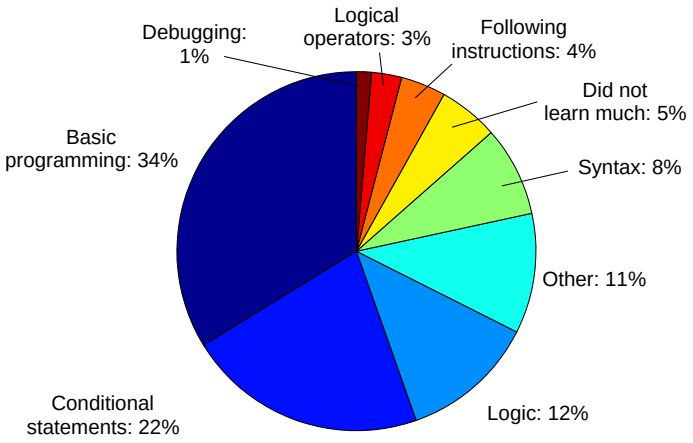


Figure 4.5: Likert scale of the qualitative evaluation

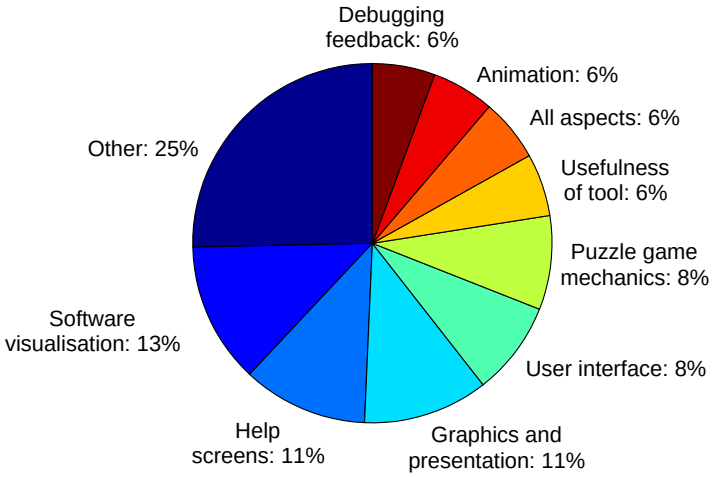
the most common responses for each question. The results of this are shown in Figure 4.6.

Skills learnt: As may be seen in Figure 4.6a, the main concepts that students were aware of learning were basic programming, the use of conditional statements and logic. This shows that the transferability of the tool is transparent – students are able to see that the skills they develop in the game may be transferred to real programming problems. This is discussed further in Section 4.7. The learning of syntax, the use of logical operators and debugging, however, were only mentioned by a minority of the audience. It is possible that these concepts are secondary to the primary learning goal, and are thus not identified by students as main learning outcomes. However, it is also possible that the game should place more emphasis on these areas (for instance, logical operators were only covered in the final two levels of the final game).

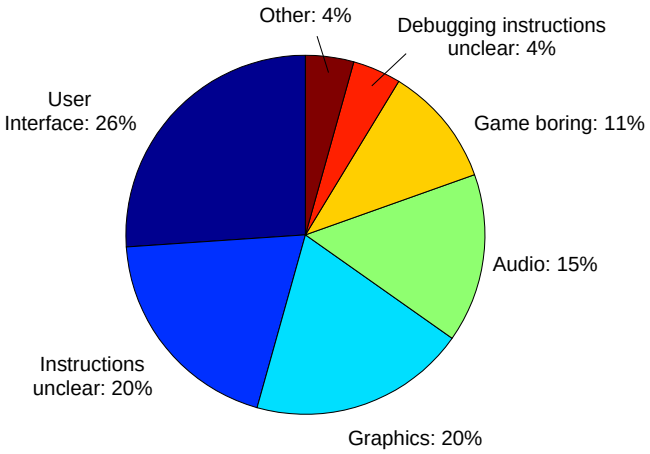
Favourable aspects of the game: Figure 4.6b shows what aspects of the game students felt were executed well. Of these, 25% were unclassifiable, but concerned aspects such as the execution of the game, the player's ability to command the cars, and game effects.



(a) *What did you learn playing this game, if anything?*

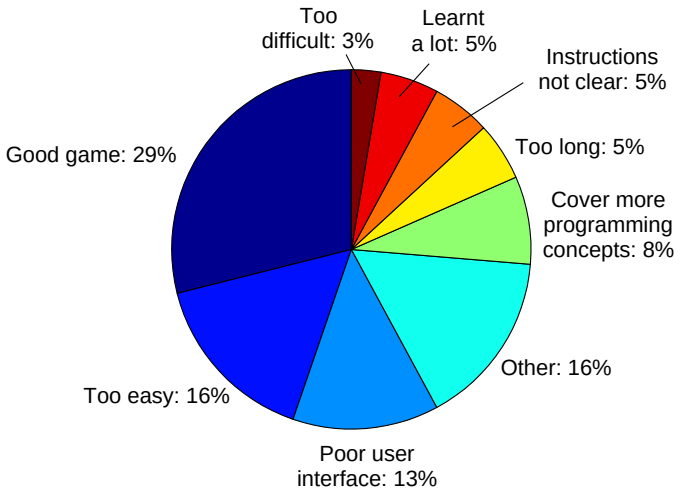


(b) *What features of the game do you think worked well or find useful and enjoyable?*



(c) *What features of the game do you think are poorly done or need improvement?*

Figure 4.6: The answers to the first three of the four open-ended feedback questions



(d) *I would really appreciate any other feedback or suggestions*

Figure 4.6: (Continued from previous page) The answers to the final open-ended feedback question

Software visualisation aspects (such as visualised code and the visual programming aspects of the game) were considered favourable by 13% of the students. The help screens were also identified as useful by 11% of players. The graphics, presentation and animation components were also highlighted (by 11% of players), as was the user interface (by 8%). A few students specifically mentioned the use of puzzle game mechanics, and a few that the game was useful and easy to learn. The following two quotes taken from the open-ended feedback indicate the importance of tool learnability which is afforded by both instructive game elements and the selection of appropriate theme:

> “ [The game] was definitely very easy to use. The code can be learnt easily with the tips at the beginning and it is a great tool to teach someone the basics of programming. ”

and:

> “ The graphics and game-play were cute and friendly. They provided a non-

> intimidating and easy way for beginners to be introduced to programming! ”

Unfavourable aspects of the game: When asked what features of the game were poorly done and need improvement, most students noted user interface issues (see Figure 4.6c). One student commented:

> “ The click and drag to move around was a bit annoying since I couldn't highlight and delete a word without the screen moving. ”

and another:

> “ When clicking on traffic intersections, the cursor did not automatically go in the 'code box'. A second click was required! ”

About a quarter of the students also felt that the instructions were unclear, and another quarter felt that the graphics were poor. 15% of students noted the lack of audio (though this was due to the computers in the test venue). Some students commented that the game was boring.

Other feedback and suggestions: Participants were also given the opportunity to give any feedback or suggestions that they desired. As shown in Figure 4.6d, about 30% of players expressed positive opinions about the game. 16% of students felt that the game was too easy, and 3% that it was too difficult. Again, comments on the quality of the user interface were made. About 8% of participants felt that the game should be extended to cover more computer programming concepts.

4.7 Lessons Learnt and Observations

This section highlights some of the observations and lessons learnt during the research and testing.

An early prototype of *Conveyor* did not have a continual stream of shapes as in the final iteration 1 game. Rather, each level had a predefined line-up of shapes, and players would programme all nodes before 'running' the level. Mistakes in players' code would only be visualised later, and it was thus more difficult to learn the programming concepts in the game. It was thus concluded that the concept of immediate visualisation is important to allow students to get immediate feedback on parts of their solutions.

Early prototypes also revealed the importance of having a quick reference system for players. Players often forget concepts that are taught in earlier levels. It is thus important not only to provide instructive elements in the game, but to allow students to refer back to these taught concepts. The help system of *if(traffic)else{win}* was thus created to allow for this. From observation it was noted that players made extensive use of this especially to reference language syntax.

As mentioned in Section 3.4.1, the levels of *Conveyor* were deliberately created to fit in a single screen. This decision was made to eliminate the need for camera control, as it was thought that this would confuse players and detract them from the programming aspects of the game. However, players were comfortable with the navigation required in iterations 2 and 3. All players observed found it understandable that they were only seeing a small view of the entire game level. This is believed to be because of a learned affordance from familiarity with other games, even casual games such as *Angry Birds*. However, movement of the camera was constrained to an isometric view with two degrees of freedom, and camera control (via mouse) separated from the programming controls (keyboard). It is thus recommended that the navigational aspects of such games be simplified as much as possible so as to maintain focus on the educational content.

It was observed that an identical puzzle (level 8 in Figure 4.3) was completed faster by less experienced students if it appeared earlier in the game. This is believed to be because students expect later levels to be necessarily more difficult, and are unable to adequately analyse the presented problems. There is insufficient data to confirm this, but could be confirmed by introducing easier levels later in the game.

The concept of transferability is important in educational game design [53]. Some educational games, such as *LightBot*, although covering the same concepts as this game, do so in a more removed manner. In *LightBot* for instance, players use a very simplified drag-and-drop interface to programme a virtual robot, whereas in the developed game, players type actual code. The designed game is thus much closer to and less abstracted from the target. As

shown in Figure 4.6a, the majority of students were able to identify the areas that were being taught, and the transferability of skills is thus transparent to players.

4.8 Summary

This chapter documents the experimental design of a few systems to measure the effectiveness of the game as a teaching tool. Also discussed is what factors of an educational game are important to both students and educators. As stated in this chapter:

- an analytics system was designed and used to collect data about how people played the game, such as the time taken and number of lines of code in the solution of each level;
- a software skills test revealed that experienced students only increased their mark by a statistically insignificant 1%, but less experienced students increased theirs by about 8%;
- a survey was used to collect qualitative data about the users' experiences of the tool;
- the measurement of the educational value of the game and the game's learning curve is important for game designers and educators;
- in this game, the transferability of skills is transparent to the player;
- the learnability of the game was afforded by the use of instructive help screens, which were made available on demand;
- the user interface was identified as an area that could use improvement.

The following chapter highlights some areas of improvement and some limitations of the developed tool.

CHAPTER 5

Recommendations and Future Work

5.1 Introduction

This chapter investigates possible areas of improvement for future research. In particular, this chapter:

- provides recommendations and improvements that could be made to the developed game in the immediate future (see Section 5.2);
- presents some alternative solutions that were considered but ultimately unimplemented (in Section 5.3);
- highlights the limitations of the developed tool (in Section 5.4).

5.2 Immediate Future

There are a number of immediate improvements that could be made to the game. For instance, the game's target and scope could be extended to include switch statements. Since the player's code is evaluated at run-time, no additional game programming would be required for this extension. A pre-level help screen could introduce the switch statement, and a number of levels dedicated to teaching players the correct syntax. The introduction of variables was designed for but not implemented in the game. This could be achieved by providing levels in which there are parking lots expecting vehicles of different colours (a single parking lot expecting 4 green cars, and 3 red trucks, for example). The player would then be required

to declare a variable at an intersection to count the number of times a green car passes, for instance. The player could then increment and query the variable in a conditional statement.

There are also a number of improvements that could be made to the testing methodology of the game. For instance, it would be highly beneficial to perform the before-and-after testing on students who perform different activities. For instance, one group could be given a lecture, one group given a textbook extract to read, and another group could be given a non-educational game to play. The time on task could remain at 45 minutes for all activities, and the increase in student marks as a result of these activities would be measured. This could further contextualise the results obtained for the increase of student results from the designed game. Since the research deals with human subjects, there is a lot of variability in the results obtained for the software skills test (see Table 4.2). To ensure more meaningful results, a larger sample size should be used. It would also be beneficial to test the effect of the game on students' marks over multiple sessions. This could indicate whether or not there is a diminishing return for total time on task, and see the lasting benefits of the game in increasing students' initial mark.

5.3 Alternatives not examined

During the course of the research, there were a number of design ideas that were not implemented. One such idea was to integrate the game with an existing course management system, *Moodle* [54]. This system would represent an intelligent tutoring system (ITS), that would consist of the game itself and some other support systems. For instance, a forum would be available to the students to discuss the game and ask questions about the puzzles; this is an example of a constructivist model of learning [54]. A competitive aspect of the game could be fostered by having a leaderboard for the players who perform the best in the game, although care would have to be taken that this does not alienate the players that perform more poorly.

The success of the sandbox open-world game *Minecraft* was also considered as the basis of an educational game [55]. In *Minecraft*, players are able to explore a voxel-based terrain and create physical structures with building-block-like cubes. *Minecraft* thus represents a mixture between games and toys (especially *Lego*), and facilitates free play and experimentation. This idea could be extended into the realm of programming. Instead of physical blocks, each cube could be a visualisation of a programmable code block. These blocks could then be placed adjacent to each other in order to interact (for instance, the output of one block could lead to

the input of another) and more complicated software systems created and visualised.

As mentioned in Section 3.3, the target taxon of this particular game was chosen as conditional statements. However, a number of other targets were considered during the early design stage. For instance, the visualisation of pointers and pointees was considered due to the difficulty students have in visualising these concepts. The visualisation of containers such as arrays and vectors was also considered for the same reasoning.

5.4 Limitations of the Developed Tool

There are a number of recommendations for software visualisation tools applied to software pedagogy [7]. The developed game does not adhere to these recommendations, as it cannot scale up to larger tasks, it focuses only on a very small aspect of computer programming. These limitations of this tool should be considered should future work be done in this area.

As mentioned in Chapter 4, the learning curve of the game is crucial to balance the educational and entertainment value of the game. It is also shown in Chapter 4 that the programming skill level of participants greatly affects their enjoyment of the game. It would thus be highly beneficial for the game to allow for customisability in order to target multiple skill levels. An educator could select levels to present to different audiences. For instance, the applied computing students could have been given only the more difficult levels of the game since they were already familiar with many of the concepts taught. Currently, the tool does not allow for customisability, and the audience is thus limited to individuals with no programming experience.

As mentioned, the statistical validity of the result obtained is also limited due to small sample sizes. This methodological limitation may be addressed in future research by broadening testing on more people and a wider audience, including school students or students from other faculties at the University of the Witwatersrand.

From observation, it was noted that some students approached the game in a competitive manner, and tried to finish levels faster than their peers. Currently, the game does not exploit this competitive spirit of its players. As noted above, however, care should be taken when doing this. It would be worthy to investigate this in future research.

5.5 Summary

This chapter has presented some recommendations for possible future work. Specifically:

- the scope of the game could be expanded to include switch statements and variables;
- the game's educational value (measured by before-and-after testing) could be measured relative to other activities such as a lecture or a reading of a textbook abstract on the same material;
- the game could be integrated with *Moodle*, the course management system used in the School;
- the idea behind an existing game, *Minecraft*, could be altered to provide a rich and enjoyable software visualisation tool;
- other targets (such as the visualisation of arrays, vectors and pointers) were considered but not implemented;
- the developed tool is very limited in scope and cannot scale to other tasks;
- the tool does not allow for customisability, and is thus limited in target audience;
- a natural competitive element could be introduced to provide a more enjoyable game.

The following chapter provides a recapitulation of work presented in this document, and highlights the answers to the research questions.

CHAPTER 6

Conclusion

6.1 Introduction

This chapter provides a recapitulation of the work that has been presented in this dissertation – see Section 6.2. Also provided is a summary of the answers to the research questions and a discussion of how the research findings support these answers – see Section 6.3. Section 6.4 makes some postulations of the future of serious games and the future of this research.

6.2 Recapitulation of Work Presented

This section provides a summary of the research dissertation.

Research overview: This dissertation has presented an investigation into the application of digital games in software development pedagogy. The research represents an overlap of the fields of digital games, educational games, software visualisation and visual programming (see Section 1.2). An educational game has been designed, developed and tested in three iterations. It is a puzzle game that requires players to programme simple sorting algorithms to direct traffic to the correct locations (see Chapter 3). Such a tool can help lessen the barrier to entry of learning basic computer programming concepts, and help stimulate interest and re-engage students in learning computer science. The reader is referred to Chapter 1 for more details of the research overview.

Research questions: The research aims to answer the following two research questions:

1. Where do digital games fit within a taxonomy of software visualisation techniques?
2. When designing an educational game for teaching novice computer programming skills:
   (a) what factors should be prioritised, from a student's perspective?
   (b) what factors should be prioritised, from an educator's perspective?

Literature review findings : There is a globally declining interest in computer science, especially amongst female students. Software visualisation, which is the rendering of intangible software systems into a more accessible visual format, may be used in an academic environment to re-engage learning in this field. Existing taxonomies of software visualisation techniques define six main taxa: task, audience, target, representation, medium and effectiveness. Many software visualisation tools aimed at novice computer programmers bare marked similarities to educational computer games. It is within the representation and medium taxa that these similarities are found, and where the fields of educational games and software visualisation overlap – see Section 6.3, below. Educational computer games fall within a broader field of serious games – games with a purpose transcending mere entertainment. Games are seen by many researchers as valuable tools to augment traditional education since they provide a simulation of their problem domain, stimulate interest in this domain, afford experimental and inquiry-based learning and are inherently self-motivating and rewarding experiences. The reader is referred to Chapter 2 for an in-depth review of relevant literature.

Research methodology: As a part of this research, an educational computer game has been developed. The game, *if(traffic)else{win}*, aims to reduce the learning curve of some basic computer programming concepts by providing a didactic software visualisation and visual programming environment. The intended audience is any novice computer programmer, such as first- and second-year engineering students or school students. The game is ill-suited to people who are already familiar with the covered concepts. The game introduces conditional statements and logical operators, primarily. Secondarily, players also learn programming syntax (`JavaScript`, in this case) and debugging. The game also presents both exploratory and instructive learning elements. Players are required to solve puzzles by themselves using logical thinking, and are also required to follow instructions. The instructive elements of the game are presented in help screens, which the player can refer to as a quick reference guide.

Testing methodology: The game was tested in a number of ways. Firstly, a before-and-after software skills quiz was administered. This multiple-choice test asked questions covering the concepts taught in the game. The increase in student marks from before and

after playing the game thus quantifies the game's educational value. More advanced students did not benefit much from the game (only an increase of 1% was measured after a time on task of 45 minutes), but novice students increase their marks by 8% in the same time. Secondly, an integrated analytics system collected data while students played the game. The time that students spend on each level, for instance, may be used as a measure of the learning curve of the game. This data can also be used to design lectures and tutorials that focus on concepts that students are battling with (see the following section). Thirdly, a survey was used to collect qualitative data about the audience, such as their demographics and user experience with and perceptions of the tool.

6.3 Answers to Research Questions

This section restates the research questions and summarises how the presented work leads to the answering of each.

6.3.1 Digital Games as Representations of Software

This section presents the findings relating to the research question: Where do digital games fit within a taxonomy of software visualisation techniques?

This research has shown that existing taxonomies of software visualisation specify six main areas of software visualisation tools: task, audience, target, representation, medium and effectiveness. As shown in Figure 6.1, each of these taxa attempts to answer a specific question about the tool. When examining the use of digital games within this taxonomy, the areas of representation and medium become the most important (see Section 2.3.2). That is, the task (the purpose of the tool), the desired target audience and the target (what aspect of software systems is visualised) could remain the same for any number of chosen applications. For instance, the task, audience and target of this developed tool (see Chapter 3) could have been presented in a non-game format such as a visual learning aid. However, the choice of using affects both the representation and medium of the tool.

The reader is referred back to the subcategories of these taxa, as presented in Figure 2.3. In the developed tool, the graphical vocabulary of traffic is used. The visual metaphor of branching road networks is used to represent the branching nature of code. Through the use of colour and animation, the flow control of traffic is used to mirror the flow control of a computer programme. Interaction is afforded by the use of the keyboard and mouse: the keyboard allowing the player to visually programme the rudimentary sorting algorithm that

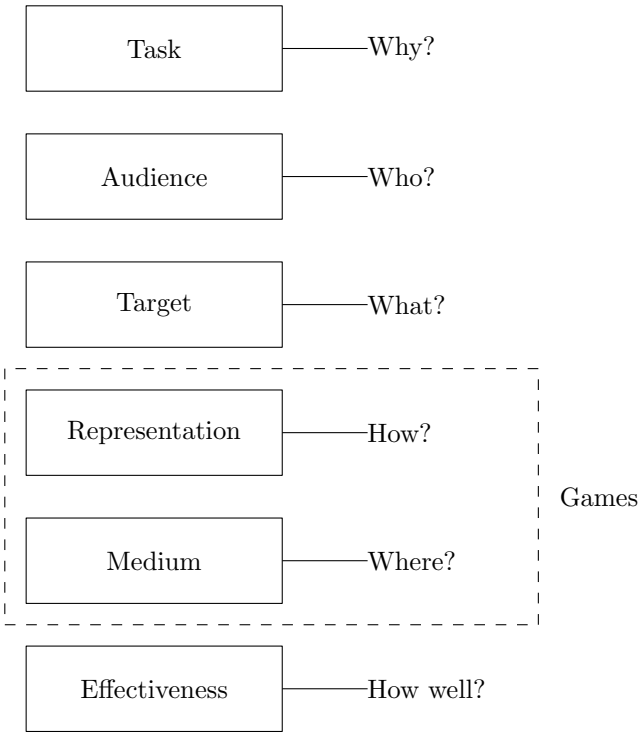


Figure 6.1: The areas of software visualisation that are most important when considering games

solves the game's puzzles; the mouse allowing for navigation. Many aspects of the game design are thus reflected in the representation and medium taxa of software visualisation techniques.

6.3.2 Factors Important to Students

This section presents the findings in relation to the research question: When designing an educational game for teaching novice computer programming skills, what factors should be prioritised, from a student's perspective?

Game enjoyment: When using games as a medium for education, it is vital to ensure the educational value of the game does not come at the expense of its entertainment. Indeed, such serious games have to balance the seriousness and the game elements (see Section 2.4.2). From a student's perspective, the entertainment value of the game is essential. The qualitative data collected for this indicates that the game was enjoyed by players, as shown by the Likert scale in Figure 4.5. The overall positive opinion of the the game – 30% of participants commented that it was a good game, see Figure 4.6b – indicates that this was fulfilled in this particular application.

Game length: The enjoyment that students have playing the game, according to the results in Figure 4.5, was found to be negatively impacted by the length of the game. The game should thus be as short as possible without compromising its educational intention, and

while still being accessible for a wide audience. In other words, the learning curve of the game has to be balanced with the overall game length. This is discussed later in this section.

User interface: The Likert scale in Figure 4.5 shows that the user interface was rated as good on average over all three iterations. However, in the open-ended feedback (see Figure 4.6c), roughly 26% of participants commented that the user interface was poorly executed. Reasons given for this were that separate clicks were required to select and programme an intersection, and that some menus and windows would occasionally overlap. There was also a navigation issue: clicking and dragging would navigate the game's camera, and so code could not be selected with the mouse. Such in-depth feedback indicates that the user interface is an important component from a student's perspective.

Tool learnability: It is vital for all games, especially educational games, to be learnable by players in a short amount of time. For the designed game, which was designed to replace a tutorial session and to have a gameplay time of only 45 minutes, it was important to be stand-alone, and not require a session to teach the game to students. The game thus used the instructive help screens – discussed in Section 3.4.3 – to allow players to learn the game easily. As shown in Section 4.6.5, students commented that the help screens were indeed useful, as was the selection of an approachable theme.

A concise, on-demand help system: From observation (see Section 4.7), it was found that the game should not only have instructive elements, but that these elements should be available on demand. Often, a player would forget something (such as syntax) that was taught in previous levels. These instructive elements thus need to be compiled into a quick reference that is available to students when they need them.

Game graphics: Students rated the graphics of both *Conveyor* and *if(traffic)else{win}* as good (see Figure 4.5), despite the improvement of the graphics between these iterations. This indicates that even simplistic graphics are perceived as adequate for such a tool. In open-ended feedback, roughly 20% of students, mentioned graphics as a poorly executed component, and only 11% thought that they were good. The graphics of educational games should thus be given some attention, but, according to this research, other components such as the level design and user interface should take priority.

6.3.3 Factors Important to Educators

This section presents the findings in relation to the research question: When designing an educational game for teaching novice computer programming skills, what factors should be

prioritised, from an educator's perspective?

Quantification of educational value beyond the game: As mentioned in Section 4.4 and Section 4.6.1, it is vital that educators can quantify the educational benefit of using the game. The developed before-and-after test, for instance, has presented a quantification of the usefulness of the tool to help different student groups (see Section 4.6.1). It is important that such tests relate back to the subject matter being taught (in this case, conditional statements and logical operators), and do not simply test the students' ability to learn the game itself. This transferability of skills learnt is another important aspect of educational game design.

Transparent transferability: As mentioned in Section 4.6.5 and Section 4.7, it is important that the game teaches concepts in such a way that they are transferable to more 'real-world' applications. The educator should ensure that the skill transferability is transparent to students. It was a conscious decision to require students to type actual code in this game to make the game as close to an actual laboratory exercise. The open-ended feedback shown in Figure 4.6a shows that the majority of students were able to identify the skills that were being taught: 34% identified 'basic programming' and 22% identified 'conditional statements'. A smaller percentage of students observed the secondary learning skills of logic, syntax and debugging.

Measurement of learning curve: The measurement of time spent per level is used in this research as an indication of the difficulty of levels, and thus a measurement of the game's learning curve (see Section 4.6.2). This is important from the perspective of a designer and educator for two reasons. Firstly, it allows a designer to refactor the game if, for instance, some levels are too difficult or too easy. Secondly, an educator is also able to see which levels (and thus, which programming concepts) students struggle with. This is an example of how the collected data could be used to inform traditional teaching components of the course.

Collection of data to inform traditional learning, on a personalised basis: As mentioned, the learning curve or measure of difficulty that students have with each level could be used to focus lectures or tutorials on these areas. Other data collected in this game (such as the players' solutions to levels) could also be used to identify common mistakes or poor programming practices. The collection of this data should also be on a personalised level. A lecturer thus has an indication of personal performance in the game, and this could be used to inform other administrative decisions. It also allows for possible personalised learning.

Accessibility by a large audience and tool customisability: In this research, it was found that this particular game is a lot more beneficial for a group of novice programmers

(Electrical Engineering students) than for a student group (Applied Computing students) who had already done a course in computer programming (see Section 4.6.1). For an educator, it is important to be able to use the same tool on as wide an audience as possible. It is thus important to be able to customise the tool for presentation to different audiences. For instance, more advanced students could be given a game with the easier level omitted. This was not implemented in the scope of this research, but should be included in future work.

Effort of game creation: The development of the game required a lot of effort. The required time to develop such a game should therefore be estimated and compared to its possible educational benefit prior to development. Only if the educational benefit outweighs the development effort should the game be created. However, the creation of the game could be a one time effort, but the game could be used over multiple years of teaching.

6.4 The Future

There is a globally declining interest in the fields of computer science and engineering. Software visualisation tools such as *Alice* and *Robomind* are used to reduce the learning curve of a student's first exposure to computer programming. These tools bear resemblance to many digital games. Within a taxonomy of software visualisation techniques, the areas of representation and medium are encompassed when using games to visualise software. Educational games may be used because of the popularity and approachability of games, their ability to afford experimental, exploratory learning and their innate reward systems which provide self-motivating environments for learning. Games are intended as a tool to augment, not replace, traditional education, and should be designed with care so as not to trivialise or devalue the educational experience. This research has identified how games may be used within a framework of software visualisation techniques, and factors that should be prioritised from both a student's and an educator's perspectives when designing educational games. This research may thus be used for the future development of educational games, especially for software pedagogy.

References

[1] K. Maillet and M. Porta. Consequences of the declining interest in computer science studies in Europe. In *Education Engineering (EDUCON)*, pages 71–76. IEEE, 2010.

[2] M. Porta, K. Maillet, and M. Gil. The Computer Science Declining Phenomenon. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 2, pages 1173–1178, 2010, [Available at: http://upcommons.upc.edu/e-prints/bitstream/2117/13904/1/Porta.pdf].

[3] M.J. Mayo. Games for science and engineering education. *Communications of the ACM*, 50(7):30–35, 2007.

[4] J. McGonigal. *Reality is broken: Why games make us better and how they can change the world*. Penguin Press HC, 2011.

[5] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

[6] J.I. Maletic, A. Marcus, and M.L. Collard. A task oriented view of software visualization. In *First International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE, 2002.

[7] M. Eisenstadt, B.A. Price, and J. Domingue. Software visualization as a pedagogical tool. *Instructional Science*, 21(5):335–364, 1992.

[8] R. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. pages 369–381. The MIT Press, Cambridge, MA, 1998.

[9] M. Petre and E. de Quincey. A gentle overview of software visualisation. *PPIG Newsletter*, 2006.

[10] B.A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *ACM SIGCHI Bulletin*, volume 17, pages 59–66. ACM, 1986.

[11] Kevin Webach. Coursera: online gamification course offered by Univeristy of Pennsylvania. https://www.coursera.org/course/gamification: [Last accessed 2nd November 2012].

[12] Unity - game development tool. http://unity3d.com/unity/ [Last accessed 16th May 2012].

[13] Limesurvey - the free and open source survey software tool. http://www.limesurvey.org/ [Last accessed 17th Novemeber 2012].

[14] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[15] T. Ball and S.G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[16] M. Tory and T. Moller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

[17] P. Young and M. Munro. Visualising software in virtual reality. In *6th International Workshop on Program Comprehension, 1998. IWPC'98*, pages 19–26. IEEE, 1998.

[18] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd, Chichester, England, 3rd edition, 2008.

[19] Humane assessment. http://humane-assessment.com/: [Last accessed 4th February 2012].

[20] Moose technology. http://www.moosetechnology.org/: [Last accessed 4th February 2012].

[21] Welcome to codecity. http://www.inf.usi.ch/phd/wettel/codecity.html: [Last accessed 4th February 2012].

[22] Gource: Software version control visualisation. http://code.google.com/p/gource/: [Last accessed 4th February 2012].

[23] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. *Software landscapes: Visualizing the structure of large software systems*. Bibliothek der Universität Konstanz, 2004.

[24] L. Feijs and R. De Jong. 3d visualization of software architectures. *Communications of the ACM*, 41(12):73–78, 1998.

[25] A. Marcus, L. Feng, and J.I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff. ACM, 2003.

[26] C++ and java source code analysis. http://www.imagix.com/products/source-code-analysis.html: [Last accessed 17th March 2012].

[27] DrJava. http://www.drjava.org/: [Last accessed 17th November 2012].

[28] Logo foundation. http://el.media.mit.edu/logo-foundation/ [Last accessed 17th November 2012].

[29] Karel J. Robot. `http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html` [Last accessed 17th November 2012].

[30] Alice.org. http://www.alice.org/: [Last accessed 4th February 2012].

[31] Robomind.net. http://www.robomind.net/en/index.html [Last accessed 15th March 2012].

[32] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493. ACM, 2000.

[33] Scratch. http://scratch.mit.edu/ [Last accessed 7th February 2013].

[34] Mitch Resnick. Let's teach kids to code. TED talk, 2004.

[35] T. Susi, M. Johannesson, and P. Backlund. Serious games–an overview. *Skövde: University of Skövde (Technical Report HS-IKI-TR-07-001)*, 2007.

[36] B. Brathwaite and I. Schreiber. *Challenges for game designers*. Course Technology, 2009.

[37] K. Squire and H. Jenkins. Harnessing the power of games in education. *Insight*, 3(1):5–33, 2003.

[38] B.S. Bloom, M.D. Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl. Taxonomy of educational objectives: Handbook I: Cognitive domain. *New York: David McKay*, 19:56, 1956.

[39] J. D. Bayliss. Teaching Game AI Through Minecraft Mods. *IEEE International Games Innovation Conference*, pages 17–20, 2012.

[40] .net terrarium 2.0. http://terrarium2.codeplex.com/ [Last accessed 15th March 2012].

[41] Robocode. http://robocode.sourceforge.net/ [Last accessed 15th March 2012].

[42] Droid battles. http://www.bluefire.nu/ [Last accessed 15th March 2012].

[43] Rails for zombies. http://railsforzombies.org/ [Last accessed 15th March 2012].

[44] Simse online. http://www.ics.uci.edu/~emilyo/SimSE/ [Last accessed 15th May 2012].

[45] Vim adventures. http://vim-adventures.com/ [Last accessed 10th May 2012].

[46] Computer science unplugged. http://csunplugged.org/ [Last accessed 15th March 2012].

[47] Zombies, run! https://www.zombiesrungame.com/: [Last accessed 2nd November 2012].

[48] The speed camera lottery. http://www.thefuntheory.com/speed-camera-lottery-0 [Last accessed 7th February 2013].

[49] L. Sheldon. *The Multiplayer Classroom: Designing Coursework as a Game*. Course Technology Press, 2011.

[50] K. Bierre. Implementing a game design course as a multiplayer game. *IEEE International Games Innovation Conference*, pages 137–140, 2012.

[51] J.P. Gee. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)*, 1(1):20–20, 2003.

[52] K. Salen and E. Zimmerman. *Rules of play: Game design fundamentals*. The MIT Press, 2003.

[53] A.L. Alexander, T. Brunyé, J. Sidman, and S.A. Weil. From gaming to training: A review of studies on fidelity, immersion, presence, and buy-in and their effects on transfer in PC-based simulations and games. In *The Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), NTSA, Orlando, Florida*, 2005.

[54] Moodle.org: open-source community-based tools for learning. http://moodle.org/ [Last accessed 3rd May 2012].

[55] Minecraft. https://minecraft.net/ [Last accessed 17th November 2012].

Bibliography

K. Maillet and M. Porta. Consequences of the declining interest in computer science studies in Europe. In *Education Engineering (EDUCON)*, pages 71–76. IEEE, 2010.

M. Porta, K. Maillet, and M. Gil. The Computer Science Declining Phenomenon. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 2, pages 1173–1178, 2010, [Available at: http://upcommons.upc.edu/e-prints/bitstream/2117/13904/1/Porta.pdf].

M. Eisenstadt, B.A. Price, and J. Domingue. Software visualization as a pedagogical tool. *Instructional Science*, 21(5):335–364, 1992.

M.J. Mayo. Games for science and engineering education. *Communications of the ACM*, 50(7):30–35, 2007.

Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

J.I. Maletic, A. Marcus, and M.L. Collard. A task oriented view of software visualization. In *First International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE, 2002.

R. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. pages 369–381. The MIT Press, Cambridge, MA, 1998.

J. McGonigal. *Reality is broken: Why games make us better and how they can change the world.* Penguin Press HC, 2011.

T. Ball and S.G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

B.A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *ACM SIGCHI Bulletin*, volume 17, pages 59–66. ACM, 1986.

M. Tory and T. Moller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

P. Young and M. Munro. Visualising software in virtual reality. In *6th International Workshop on Program Comprehension, 1998. IWPC'98*, pages 19–26. IEEE, 1998.

Peter C. Rigby, Davor Cubranic, Suzanne Thompson, Daniel German, and Margaret-Anne Storey. The challenges of creating open source educational software: the gild experience. *Open Educational Symposium of the 1st International Conference on Open Source Systems*, pages 338–340, 2005.

M. Petre and E. de Quincey. A gentle overview of software visualisation. *PPIG Newsletter*, 2006.

Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Ltd, Chichester, England, 3rd edition, 2008.

Gource: Software version control visualisation. http://code.google.com/p/gource/: [Last accessed 4th February 2012].

Moose technology. http://www.moosetechnology.org/: [Last accessed 4th February 2012].

Humane assessment. http://humane-assessment.com/: [Last accessed 4th February 2012].

Welcome to codecity. http://www.inf.usi.ch/phd/wettel/codecity.html: [Last accessed 4th February 2012].

Alice.org. http://www.alice.org/: [Last accessed 4th February 2012].

M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493. ACM, 2000.

H.M. Kienle and H.A. Muller. Requirements of software visualization tools: A literature survey. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 2–9. Ieee, 2007.

Visualisation toolkit. http://www.vtk.org/: [Last accessed 4th February 2012].

Welcome to sv3d. http://www.sdml.info/projects/sv3d/: [Last accessed 4th February 2012].

C++ and java source code analysis. http://www.imagix.com/products/source-code-analysis.html: [Last accessed 17th March 2012].

XNA Developer Center. http://msdn.microsoft.com/en-us/aa937791: [Last accessed 4th February 2012].

M.A.D. Storey. *A cognitive framework for describing and evaluating software exploration tools*. PhD thesis, Simon Fraser University, 1998.

Human-computer interaction. http://www.hci-class.org/: [Last accessed 4th February 2012].

P. Young and M. Munro. Visualising software in virtual reality. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 19–26. IEEE, 1998.

Simse online. http://www.ics.uci.edu/˜emilyo/SimSE/ [Last accessed 15th May 2012].

W. Dann, S. Cooper, and R. Pausch. Using visualization to teach novices recursion. *ACM SIGCSE Bulletin*, 33(3):109–112, 2001.

B. Brathwaite and I. Schreiber. *Challenges for game designers*. Course Technology, 2009.

Jane McGonigal. Gaming for a better world. TED talk, 2004.

K. Salen and E. Zimmerman. *Rules of play: Game design fundamentals*. The MIT Press, 2003.

T. Barnes, H. Richter, A. Chaffin, A. Godwin, E. Powell, T. Ralph, P. Matthews, and H. Jordan. Game2learn: A study of games as tools for learning introductory programming concepts. *Proceedings of the ACM SIGCSE*, 7, 2007.

Microsoft Research. Transforming computer science in the gaming age. Microsoft External Research, 2008.

Lightbot. http://www.newgrounds.com/ [Last accessed 15th March 2012].

Robozzle online puzzle game. http://robozzle.com/ [Last accessed 15th March 2012].

Robomind.net. http://www.robomind.net/en/index.html [Last accessed 15th March 2012].

.net terrarium 2.0. http://terrarium2.codeplex.com/ [Last accessed 15th March 2012].

Robocode. http://robocode.sourceforge.net/ [Last accessed 15th March 2012].

Droid battles. http://www.bluefire.nu/ [Last accessed 15th March 2012].

Rails for zombies. http://railsforzombies.org/ [Last accessed 15th March 2012].

Computer science unplugged. http://csunplugged.org/ [Last accessed 15th March 2012].

L. Feijs and R. De Jong. 3d visualization of software architectures. *Communications of the ACM*, 41(12):73–78, 1998.

A. Marcus, L. Feng, and J.I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff. ACM, 2003.

M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. *Software landscapes: Visualizing the structure of large software systems*. Bibliothek der Universität Konstanz, 2004.

Wolfgang Kramer. What makes a game good? http://www.thegamesjournal.com/articles [Last accessed 21st March 2012].

Pete Collier. Gaming reality: What makes a good game? http://www.petecollier.com/?p=243 [Last accessed 21st March 2012].

J.P. Gee. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)*, 1(1):20–20, 2003.

Moodle.org: open-source community-based tools for learning. http://moodle.org/ [Last accessed 3rd May 2012].

Pedagogy - moodledocs. http://docs.moodle.org/22/en/Pedagogy [Last accessed 6th May 2012].

Kodu - Microsoft Research. http://research.microsoft.com/en-us/projects/kodu/ [Last accessed 10th May 2012].

Vim adventures. http://vim-adventures.com/ [Last accessed 10th May 2012].

Unity - game development tool. http://unity3d.com/unity/ [Last accessed 16th May 2012].

Limesurvey - the free and open source survey software tool. http://www.limesurvey.org/ [Last accessed 17th Novemeber 2012].

T. Susi, M. Johannesson, and P. Backlund. Serious games–an overview. *Skövde: University of Skövde (Technical Report HS-IKI-TR-07-001)*, 2007.

A. Mitchell and C. Savill-Smith. The use of computer and video games for learning: A review of the literature. 2004.

Julie Sarama and Douglas Clements. Building blocks and cognitive building blocks: Playing to know the world mathematically. *Americal Journal of Play, Volume 1, Number 3*, pages 1–25, 2009.

K. Squire and H. Jenkins. Harnessing the power of games in education. *Insight*, 3(1):5–33, 2003.

R. van Eck. Digital game-based learning: It's not just the digital natives who are restless. *EDUCAUSE review*, 41(2):16–30, 2006.

J. D. Bayliss. Teaching Game AI Through Minecraft Mods. *IEEE International Games Innovation Conference*, pages 17–20, 2012.

A. Decker D. Simkins, C. Egert. Evaluating Martha Madison: Developing analytical tools for gauging the breadth of learning facilitated by STEM games. *IEEE International Games Innovation Conference*, pages 137–140, 2012.

Kevin Webach. Coursera: online gamification course offered by Univeristy of Pennsylvania. https://www.coursera.org/course/gamification: [Last accessed 2nd November 2012].

Zombies, run! https://www.zombiesrungame.com/: [Last accessed 2nd November 2012].

K. Bierre. Implementing a game design course as a multiplayer game. *IEEE International Games Innovation Conference*, pages 137–140, 2012.

L. Sheldon. *The Multiplayer Classroom: Designing Coursework as a Game*. Course Technology Press, 2011.

R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

DrJava. http://www.drjava.org/: [Last accessed 17th November 2012].

Logo foundation. http://el.media.mit.edu/logo-foundation/ [Last accessed 17th November 2012].

Karel J. Robot. `http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html` [Last accessed 17th November 2012].

Minecraft. https://minecraft.net/ [Last accessed 17th November 2012].

B.S. Bloom, MD Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl. Taxonomy of educational objectives: Handbook i: Cognitive domain. *New York: David McKay*, 19:56, 1956.

A.L. Alexander, T. Brunyé, J. Sidman, and S.A. Weil. From gaming to training: A review of studies on fidelity, immersion, presence, and buy-in and their effects on transfer in PC-based simulations and games. In *The Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), NTSA, Orlando, Florida*, 2005.

Scratch. http://scratch.mit.edu/ [Last accessed 7th February 2013].

Mitch Resnick. Let's teach kids to code. TED talk, 2004.

APPENDIX A

Gantt Chart

This appendix presents a Gantt chart showing the breakdown of time spent on the research.

Task	Task Name	Started	Completed
1	Literature review draft	2012/02/01	2012/02/22
2	PEIES	2012/04/03	2012/04/03
3	Iteration 1	2012/04/03	2012/05/09
4	SAICSIT paper	2012/05/14	2012/05/27
5	Iteration 1 testing (Game Design students)	2012/05/23	2012/05/23
6	Iteration 2	2012/05/09	2012/06/16
7	Iteration 2 testing (Applied Computing students)	2012/06/16	2012/06/16
8	Iteration 3	2012/06/16	2012/09/03
9	Iteration 3 testing (Electrical Engineering students)	2012/08/03	2012/08/03
10	IGIC Paper	2012/08/01	2012/09/06
11	Amaze festival presentation	2012/08/31	2012/08/31
12	IGIC	2012/09/07	2012/09/07
13	Dissertation draft write-up	2012/09/17	2012/10/08
14	SAICSIT	2012/10/01	2012/10/01
15	Wits Cross-Faculty Symposium	2012/10/19	2012/10/19
16	Dissertation write-up	2012/10/08	2012/11/14

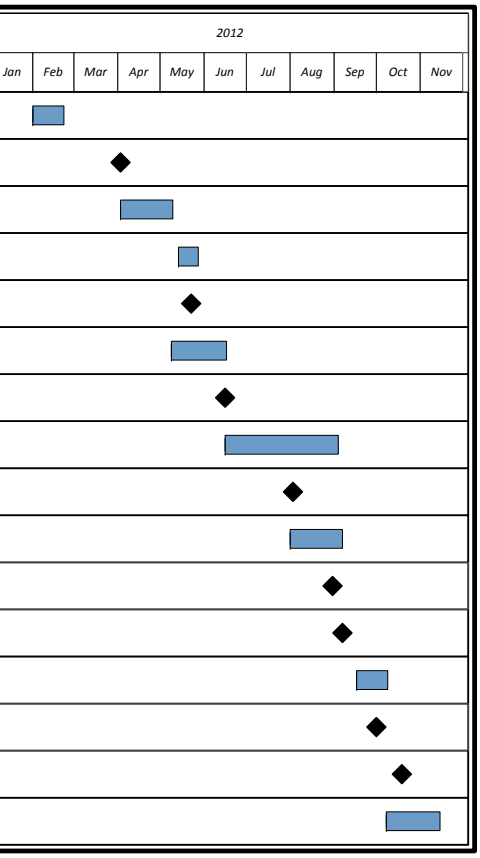


Figure A.1: Gantt chart showing the research activities

APPENDIX B

SAICSIT 2012 Conference Paper

This appendix contains the paper that was presented as a part of this research at the SAICSIT (South African Institute of Computer Scientists and Information Technologists) conference in Centurion on the 3rd of October, 2012. At the time of writing, only the first iteration of the research had been completed, and the paper only discusses the first game, *Conveyor*. The paper also focuses on the taxonomy of software visualisation and how the game fits within this.

The Application of Video Games to Visualise and Teach Computer Programming

Bradley R C Marques
School of Electrical and Information Engineering
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg, South Africa
bradley.marques@students.wits.ac.za

Stephen P Levitt
School of Electrical and Information Engineering
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg, South Africa
stephen.levitt@wits.ac.za

Ken J Nixon
School of Electrical and Information Engineering
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg, South Africa
ken.nixon@wits.ac.za

ABSTRACT

A educational computer game, *Conveyor*, was designed and developed as part of an investigation into the use of games to teach software development at a tertiary level. The puzzle game requires players to sort random input objects of particular shapes and colours into the correct outputs by programming the solution. An integrated analytics system collects data about how players engage with the game, such as the time taken and number of reattempts at each level. The game was tested on a group of 39 electrical engineering and game design students, many of whom had not programmed before. 80% of participants completed all 9 levels of the game. The minimum average time for a level was around 20 seconds with an average of 2 attempts, whilst the most difficult level took an average of 7 minutes over 3 attempts to complete. These results indicate that the game provided a good, but not overwhelming, challenge for the desired audience. A survey was used to collect qualitative feedback from the audience. The game was judged to be both fun and rewarding, but is too instructive and the user interface was deemed confusing. Further work conducted includes a simple test before and after the game is played, to judge the effect of the game on students' understanding.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*; K.8.0 [**Personal Computing**]: General—*Games*

General Terms

Design, Human Factors

SAICSIT '12, October 1-3, Tshwane, South Africa

Keywords

Computer games, Educational games, Game design, Software education, Software visualisation

1. INTRODUCTION

Fewer students are choosing careers in computer science and engineering [22, 15]. Meanwhile, the global video games industry continues to grow and reaches a wider demographic [18]. Researchers and educators are examining the use of educational and serious games to re-engage learning and support traditional teaching methods. Software visualisation is the rendition of data about a software system into a graphical format for the primary purpose of enhancing code comprehension [21, 23, 16, 13]. Software visualisation tools are thus widely used in both the professional software development industry, and in software pedagogy [13, 9, 12, 19]. Students are able to understand software systems better if they are able to visualise what is happening [13, 9]. Many software visualisation and visual programming tools exist that allow students to engage with and learn computer programming in a highly visual, interactive and enjoyable way. Such tools not only stimulate an interest in the field, but can lessen the learning curve in a student's exposure to computer science.

This paper investigates the use of computer games as a medium for software visualisation and software development pedagogy. An educational computer game, *Conveyor* has been designed, developed and tested as part of this investigation. The literature review of Section 2 contextualises the research within the field of software visualisation and serious games. Section 2.2 examines the process of visualisation. The structure of software visualisation tools is presented in Section 2.3 in terms of three existing taxonomies. Section 2.4 provides a brief discussion of software visualisation in computer science pedagogy. The design, implementation and results obtained from a game-based learning environment are presented in Section 3. The overall design of the game and supporting systems is presented in Section 3.1. How the game fits into the defined taxonomy is discussed in Section 3.2. The testing framework used to validate the tool is discussed in Section 3.3.1, and the results obtained from testing the system on a group of participants are presented in Section 3.3.2. General lessons learnt from informal interviews and observation are presented in Section 3.4. Section 3.5 briefly highlights some subsequent work and future plans. Section 4 provides a brief conclusion.

2. LITERATURE REVIEW

2.1 Software Visualisation and its Applications

Software visualisation is the use of graphical techniques to render raw data about software systems into a more perceptible format to aid code comprehension [23, 21, 10]. The data rendered is often the source code, but may be other code artefacts such as logging and tracing information, file dependencies, version control statistics, network traffic, or any other data [21, 10]. The visualisation techniques include graphic design, animation and, recently, virtual reality [23, 26, 17, 14]. Even simple typographic techniques such as indenting and syntax highlighting – common in most integrated development environments – are considered a rudimentary form of software visualisation [23]. Cinematographic methods have also been used to visualise software, thus illustrating that senses other than sight – such as sound and haptics – may also be leveraged to portray software data. [9].

As in the wider field of data visualisation, the visualisation of software data typically adds useful information, or presents the data in such a way that facilitates the derivation of some secondary information: for instance, the visualisation of a very large dataset or an inaccessible legacy system into a more tangible visualistaion. The process thus needs to take into account numerous cognitive and human-computer interaction factors such as colour, depth and space perception, short-term memory, spatial memory and spatial reasoning [21, 25]. Ultimately, software visualisation tools need to consider the so-called cognitive economy of data presentation [21]. Software visualisations may help software developers by facilitating:

- comprehension of large software architectures or legacy systems for re-engineering purposes;
- design reasoning through the visualisation of static code structure and class design;
- debugging, profiling and performance tuning of code through the visualisation of dynamic run-time behaviour;
- the software engineering process and management of software projects;
- the understanding of code-base evolution, development life cycle and developer contributions.

Software visualisation techniques may also be used to visualise very specific concepts in software or algorithm design, in which case the term algorithm visualisation is sometimes used [23, 12, 9]. An example of algorithm visualisation is the film *Sorting out Sorting* (see Section 2.4). Whereas software visualisation involves the illustration of existing computer programs, visual programming tools involve the generation or specification of software systems by a graphical means [23]. As discussed in Section 3.1.2, the developed game is a visual programming environment and a form of algorithm visualisation.

2.2 The Software Visualisation Process

The process of software visualisation, as shown in Figure 1, involves a dialogue between the human user of the system and the underlying raw data [16, 25]. This raw data undergoes a process of data derivation to produce a secondary dataset. For instance, the source code of a project could be analysed and the number of functions or classes could be derived. The derived data are then mapped into a visual structure. The entire visual structure is rarely shown; only a subset, or specific view, is rendered on screen for the user to perceive. Most software visualisation tools utilise a number of view transformations to produce views that offer different levels of granularity, filter and present different information, or offer different zoom levels [21, 16]. It is vital that software visualisation tools allow the user to interact with this process [21, 23, 16, 26]. The user should thus be able to alter the processes of data derivation, visual mapping and view transformations.

2.3 A Taxonomy of Software Visualisation Tools

Numerous taxonomies exist that define and categorise software visualisation tools [23, 16, 26]. These specify distinct areas, or taxa, of software visualisation techniques. This section presents such a taxonomy based on those found in [16], [23] and [26]. The taxonomy defines six taxa of software visualisation tools: the task, audience, target, representation, medium and effectiveness. Each taxon attempts to answer a specific question about the tool.

2.3.1 Task:

The Task taxon, answers *why* the software visualisation exists. It describes the specific problem domain the tool is targeting, and describes its purpose [23, 16]. All aspects of design depend on what specific task the tool attempts to facilitate.

2.3.2 Audience:

The Audience taxon specifies *who* will use the tool [16]. It specifies the intended human user of Figure 1. The design of the tool will differ greatly depending on the intended audience. For instance, visualisation tools aimed at professional software developers and high school children will have very different allowable levels of subject matter ignorance, levels of abstraction, institutional culture and user interface design [16, 13].

2.3.3 Target:

The Target addresses *what* aspect of the software data is visualised [16]. It may be further sub-categorised into scope and content. Scope addresses the subset of software systems that may be visualised and aspects of the tool's generality in terms of hardware, operating system and programming languages supported [23]. The scope also encompasses the scalability of the tool in terms of supported programs and datasets [23]. The content is the collection of programs and datasets that may be visualised, and thus represents the raw data in Figure 1.

2.3.4 Representation:

Representation specifies *how* the data is visualised. The processes of visual mapping, visual structure and view transformations, in Figure 1, are specified by the representation of the tool. Representation is further sub-categorised into method and interaction taxa. The method specifies the level of granularity and abstraction, the degree of automation, and whether the data gathering occurs at compile-time or run-time [23, 26]. Method also encompasses the correlation between the visualisation representations and the underlying data [23, 26]. Many software visualisation tools are lacking in this respect, leading to

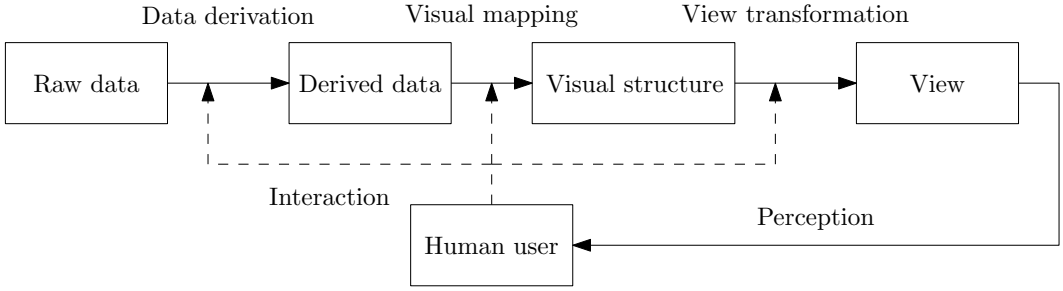


Figure 1: The software visualisation process, adapted from [16]

it being referred to as the 'missing link' of software visualisation [21]. The interaction taxon specifies the human computer interaction aspects of the tools. It describes the user interface design and the users' temporal and elision control.

2.3.5 Medium:

The taxonomy of Medium describes *where* the visualisation is presented and its presentation style: the use of graphical vocabulary, appropriate visual metaphor, and the use of animation or sound [23]. In the process of software visualisation, in Figure 1, it would encompass the concept of views.

Computer games and virtual environments that teach computer programming (as discussed more in Section 2.5) have to consider how the software system is represented, and what methods and media are used. Section 3.2 discusses how the elements of the designed game correspond to this taxonomy.

2.3.6 Effectiveness:

The Effectiveness taxon attempts to answer *how well* the software visualisation tool completes its task. Fidelity and completeness is required to ensure that the visual representations fully and truthfully portray the underlying software [23]. Appropriateness and clarity ensures that such portrayals are suitable and unambiguous [23]. Finally, the effectiveness of a software visualisation tool may be measured by empirical evaluation. Research has shown that proper empirical analysis is a rarity in this field [23, 13]. Part of this research will attempt to formalise a means of determining the effectiveness of an educational game used as a medium for software visualisation – see Section 3.3. The taxonomy is used to specify the game-based learning environment as in Section 3.2.

2.4 Software Visualisation in Software Pedagogy

Since the visualisation of software systems facilitates code comprehension, software visualisation tools may also be used in computer science pedagogy [13, 12, 9, 11]. Students are able to grasp and remember concepts more readily if they are able to visualise what is happening in their source code [13]. Typically, chalkboard sketches have been used, but software visualisation tools provide a medium for visualisation that is partly automated, scalable and interactive [13].

The 1978 motion capture film, *Sorting out Sorting*, for instance, has been used with great success at universities around the world [9]. The film represents a collection of algorithm visualisations that cover the concepts and compares the performance of nine sorting algorithms [9]. Through the medium of cinematography and the effective use of the taxa of colour, shape, animation and sound (see Section 2.3), the film portrays in thirty minutes what would take much longer to cover by traditional lecturing [9]. This example also illustrates that a software visualisation tool need not be a software system itself, and that other media may be used.

It has been shown that interactivity allows for exploratory learning: a vital aspect of software education [13]. Thus, numerous tools such as *Dr Java*, *LOGO*, and *Karel J Robot* are used at tertiary institutions to augment the traditional education of computer science [13]. Such tools provide open-ended learning environments which encourage self-motivated study [13]. Another such tool is *Alice*, developed at Carnegie Mellon University [1, 12]. Students are given a highly visual environment with which to interact. They are able to construct scenes in the virtual world by adding pre-built 3D objects, and program them using an intuitive drag-and-drop programming interface. Thus, students are able to engage with the concepts of object-oriented programming, functions and parameters, variables, conditional statements and iteration structures. The graphical programming interface mirrors what actual code looks like, thus lessening the learning curve for a novice programmer's exposure to computer programming. *RoboMind* offers a similar virtual environment in which students are able to program a robot with a simple scripting language [4]. These tools offer a highly graphical and interactive means of teaching basic computer programming skills. This presentation and interaction style is similar to that of digital games, as is examined in the following section.

2.5 Serious Games and Software Education

The term 'serious game' is used to describe a game that is used for any purpose other than mere entertainment [24]. Whilst they can provide socio-economic or socio-political commentary, they are more often associated with the learning or development of skills [24]. These skills may be in a variety of areas, including spatial reasoning, analytical and problem-solving skills, learning and recollection ability, and skills in mathematics, science and language [24, 20]. With the rise of massively-multiplayer online games, social skills such as collaboration are also learnt [24]. In 2002, the U.S. Military released a game entitled *America's Army*, which is considered the first serious game [24]. It has served both as a recruitment drive, and a combat training and simulation environment. Indeed, it is the addition of pedagogy into a game that makes a game 'serious'. Serious games can be similarly used to create environments for the visualisation and learning of computer programming. The games *LightBot* and *RoboZZle*, for instance, allows players to program virtual robots to solve logical puzzles [3, 5]. Similar to *Alice*, players drag-and-

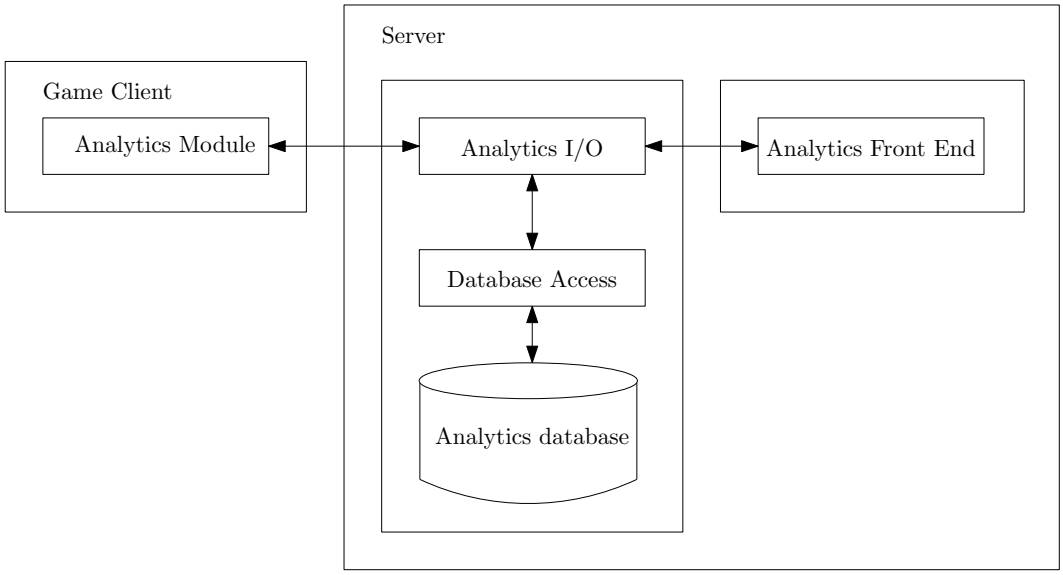


Figure 2: The system architecture

drop commands and are exposed to basic logical puzzle solving, conditional statements, and functions. The games progress in levels of increasing difficulty, and require players to make use of functional decomposition, iteration and recursion. *VIMAdventures* is an adventure role-playing game which teaches players the keyboard shortcut commands of the VIM text editor, an important tool for professional software developers [8]. *SimSE* is a management game in which players assume the role of a project manager in a software development company [6]. Players have to manage budget, time and team members and are exposed to different software engineering life cycles.

These few examples show that games can teach a variety of software and computer science concepts. They are able to present these concepts in a fun and enjoyable way, thus stimulating player engagement with the material. They are highly interactive, and thus allow for exploratory learning. The games set realistic challenges that target the players' growing realm of competence, a good model for an educational tool. This research aims at investigating the use of a game-based learning system in the teaching of software development at a university level. A system designed for testing is examined in the following section.

3. A GAME-BASED LEARNING ENVIRONMENT

3.1 Design Overview and Justification

This section examines and justifies the design of a game-based learning environment aimed at teaching basic computer programming concepts.

3.1.1 The Game-Based Learning Environment

The architecture and high-level design of the game-based learning environment is shown in Figure 2. The system consists of three main components, separated into a client-server architecture. The game itself runs in a client web browser utilising the Unity3D webplayer [7]. The game is the primary learning and software visualisation tool. A server-side analytics system is used to collect data about how players interact with the game. The data collected may be used to continually monitor and improve the game and to judge its usefulness as a software visualisation tool. A web-based analytics front end was developed to calculate and visualise the statistics from the data.

3.1.2 The Game

Conveyor is a educational computer game designed to teach novice computer programming skills. It is built using the Unity3D game framework, and aims at teaching the basic programming skills of conditional statements, and logical operators. The player's goal within the game is to solve a series of logical and programming puzzles, each contained within a game level, to sort random input objects through a series of conveyor belts and into the desired output bins. The input objects are represented as shapes (circles, squares and triangles) of various colour (red, green or blue). These must be sorted into the output bins which expect a certain number of shapes with a particular colour and shape. The player is thus required to program each node in a pre-defined conveyor belt system (a new system for each level of the game) to perform the logical steps in the sorting process.

As mentioned, the game represents both a visual programming environment and a form of algorithm visualisation; players generate very simple sorting algorithms using control statements, and the effects of their algorithm is visualised and animated. Each of the nine levels presents a different predefined puzzle (some are shown in Figures 3 - 6), for which the player must code the solution. Through instructive text hints, the player learns how to sort these objects by creating *if* statements and output the shapes to a particular node output. Figure 3 shows an annotated example of this, and a correct solution to the level. Figure 4(a) shows a higher level in the game, in which players have to satisfy two outputs. To do this, an *if* statement may be used. As may be seen, one logical decision is represented by a programmable node. Figure 5(a) shows an even more complicated case in which three outputs (red, green and blue) must be satisfied. The player thus needs to make use of an *if*, *else if*, *else* structure. Text hints are used to instruct players the first time a new construct is required. Figure 6(a) shows the last level of the game.

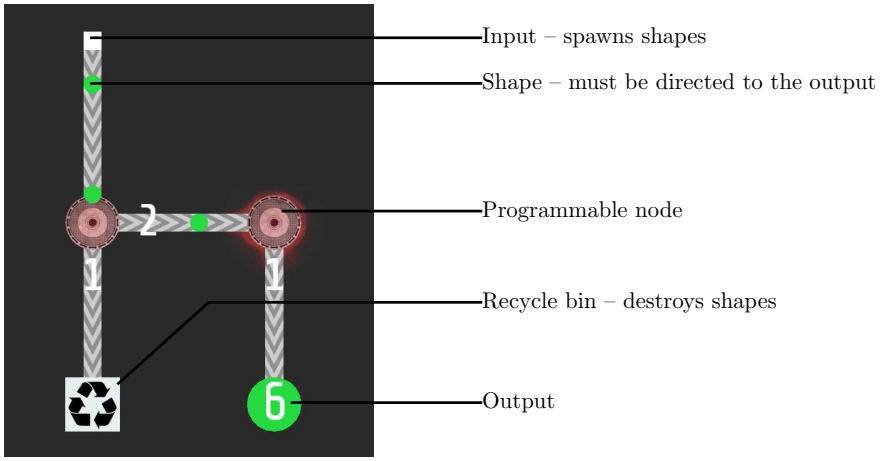
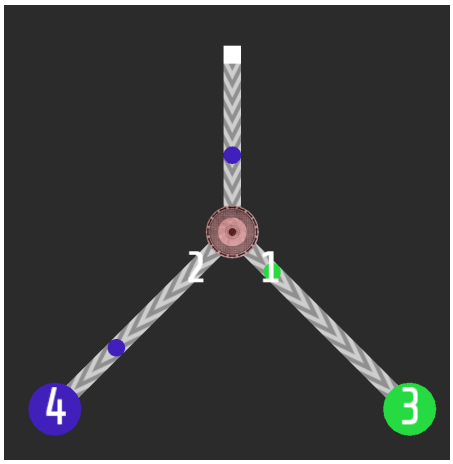


Figure 3: An annotated screenshot of a level in the game

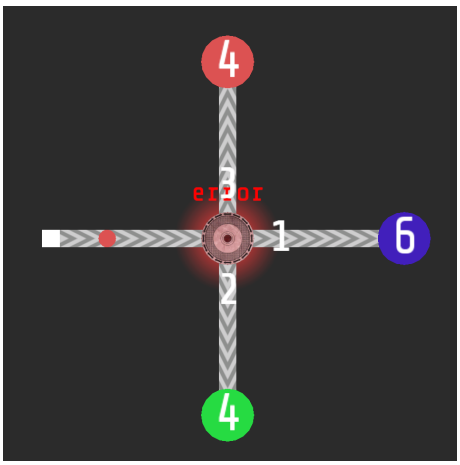


(a) A level with a simple *if* statement

```
if(shapeIsGreen())
    outputTo(1);
else
    outputTo(2);
```

(b) Solution to the level in Figure 4(a)

Figure 4: A level of the game representing a simple *if* statement and its corresponding solution

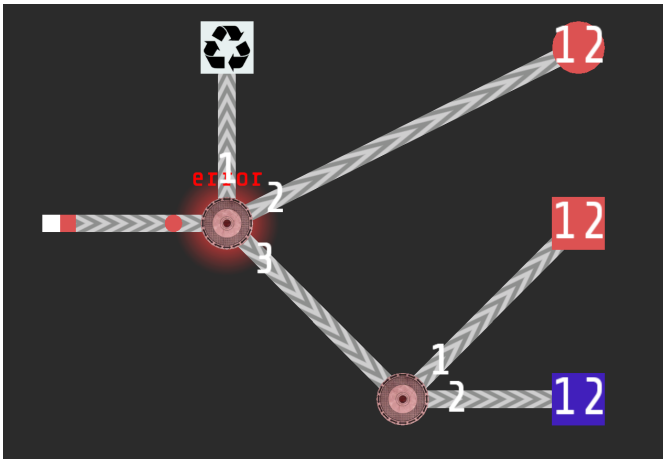


(a) A level with an *if*, *else if*, *else* structure

```
if(shapeIsBlue())
    outputTo(1);
else if (shapeIsGreen())
    outputTo(2);
else
    outputTo(3);
```

(b) Solution to the level in Figure 5(a)

Figure 5: A level of the game representing an *if, else if, else* structure



(a) The final level of the game that is solvable in a variety of ways

Node 1

```
if(shapeIsSquare())
     outputTo(3);
else if (shapeIsRed())
     outputTo(2);
else
     outputTo(1);
```

Node 2

```
if(shapeIsRed())
     outputTo(1);
else
     outputTo(2);
```

(b) Solution to the level in Figure 6(a)

Figure 6: A level in which a combination of shapes and colours are used

As seen during testing, it is solvable in a variety of ways; this indicates that the storing of players' solutions is required in order to provide meaningful feedback. Levels increase in difficulty as fewer instructive hints are given, and the logic becomes more complex. As discussed further in Section 3.2, the game provides a visual representation of code branches, and provides a goal-based exploratory environment for basic software education.

3.1.3 *The Analytics System*

The game analytics system serves to collect quantitative data about how players interact with the game. Such data may be used to measure the degree to which the game meets its requirements (see Section 3.3) and could be used to iteratively improve the game. The data may be viewed by course lecturers and mentors so that feedback can be given to the students. The analytics system is a server-side PHP-based application consisting of a logic and data layer. The logic layer allows for analytics input and output, and communicates to a database abstraction layer. This, in turn, allows for operations on the analytics database, implemented in mySQL. A small Unity analytics module was implemented to allow for communication from the game to the analytics server by using URL query strings. The actual metrics that are collected are discussed in Section 3.3.

3.2 Its Place in the Taxonomy

The taxonomy of software visualisation tools described in Section 2.3 may be used to form the framework of the game.

3.2.1 *Task*

The task of the game is to provide an engaging learning environment to lessen the learning curve of a student's first exposure to computer programming. The game thus aims to teach the basics of programming syntax, conditional statements and logical operators. The task of the learning tool is also to provide a collaborative learning environment to encourage self-motivated exploratory learning. The game has a very similar task to applications such as *Alice* and *Robomind*.

3.2.2 *Audience*

The intended audience is any person who wishes to begin learning computer programming, particularly, first-, second- and third-year engineering students studying software development and game design. The game thus assumes a minimal level of knowledge and would not be very useful for a proficient audience. In comparison to *Alice* and *Robomind* – targeted at primary and high-school children – this game is intended for a slightly older audience.

3.2.3 *Target*

The scope of the tool is very limited, as it consists of predefined virtual environments that merely respond to code written by the player. The content of the visualisation includes conditional statements, logical operators, basic functions and JavaScript syntax. *Alice*, for example, has a much wider target since it allows students to create their own functions and variables in an open-ended environment.

3.2.4 *Representation*

The game represents code segments as programmable nodes. For instance, the level in Figure 4(a) shows an *if, else* structure, and that of Figure 5(a) shows an *if, else if, else* structure. The game offers a view of the underlying code with interactive windows. Such windows allow for interaction with keyboard – to type the code – and mouse – for repositioning. Code correlation is visualised through the relationship of what a player types and what effect this has on the game-world. The player receives immediate visual and audio feedback of the cause and effect relationships of the code that he/she types; in this case the JavaScript is evaluated and visualised at runtime. When a syntactical error occurs, a red error message is displayed above the corresponding node, and a particular sound is played. This is analogous to a unit test failing with a particular input. When a player makes a logical error – for instance, sends a shape into the incorrect output – an error message is displayed and a different sound is played. When the player has no syntactical errors and the shapes are directed to the correct outputs, a success sound is played, analogous to a unit test passing all inputs.

From initial testing, it became clear that players also re-

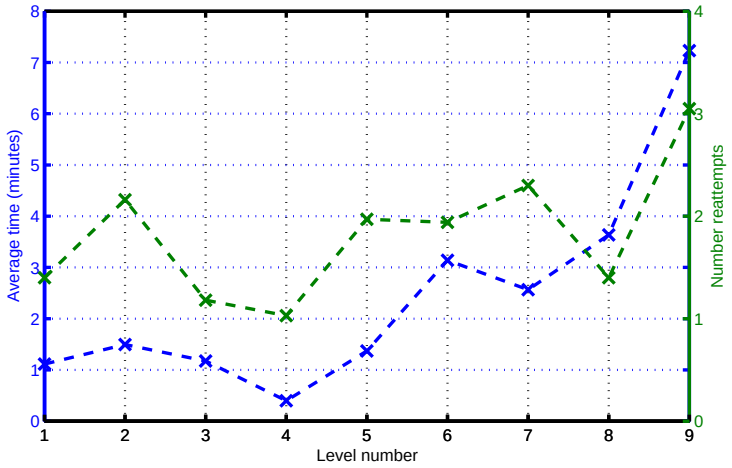


Figure 7: Average time spent on and number of attempts at each level

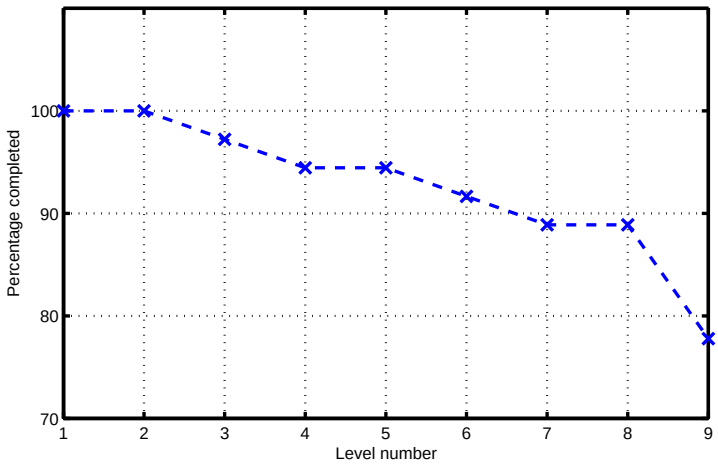


Figure 8: Percentage of players to complete each level

quire a quick reference system to refer to something that they may have been taught, but cannot recall. Thus, the 'codebook' feature was introduced as a vital interface option. This feature provides a quick reference for students to view commands and syntax that has been taught in previous levels. In this particular visualisation tool, automation does not apply since it is not visualising an existing code base but, rather, provides an environment for learning programming with rich visual and audio feedback.

3.2.5 Medium

Each level offers its own static camera view of all the elements in the level. The user may control the view to some extent by opening, closing or moving the windows that allow for code input. The 'inputs' to the simulated software system are represented by objects of different colour and shape, and each logical step in the sorting process is represented by a node. Distinct primary colours are used to represent the game objects. The desired outputs are shown as boxes with a particular shape, colour and numerical value describing what types of shapes need to be brought to it. The game uses animation to show the process of sorting, and the result of each node's code as the shapes move around the virtual code branches. Distinct sounds are used to indicate when a player has made a syntactical error, a mistake in logic, or has produced the required output of the system.

3.2.6 Effectiveness

The effectiveness of the system is to be determined by both quantitative and qualitative analysis as described in Section 3.3.

3.3 Testing and Analysis

This section describes how the data was collected by examining the testing framework in Section 3.3.1. The game was tested on a group of 39 voluntary participants. The gameplay analytics and qualitative data collected is analysed and discussed in Section 3.3.2.

3.3.1 Testing Framework

As mentioned in Section 3.1.3, a server-side analytics system collects data regarding how players engage with the game. The time taken for players to complete each level is stored (to ensure participant anonymity, players are identified by a randomly-generated identification number). The number of times a player reprogrammes each

node in each level is also stored, effectively tracking the number of reattempts at a level.

Questionnaires were administered to the 39 participants. These evaluated their perception of their programming and gaming backgrounds, and their opinions of the game. The game is judged in terms of its usefulness, the ease of learning the tool and the memorability of the material covered. Participant's answers to a semantic differential are used to judge the game in terms of its entertainment and reward value, its facilitation of creative play and its graphical, audio and user interface quality. The results of both the quantitative and qualitative metrics are presented and discussed in the following section.

3.3.2 Analysis of Results

Quantitative Results

The participant group consisted of second-, third-, and fourth-year electrical, information and biomedical engineering students, all of whom had previous programming experience. Also present were a class of first-year game design students consisting of both engineers and artists; these students had no formal exposure to computer programming. The participants played the game in a 45 minute session, and were given about 15 minutes to complete the survey.

Figure 7 shows the average time spent on each level, and the average number of attempts at each. As may be seen, these two datasets follow each other; thus, the plots show the effective learning curve of the game over the nine levels. There is an initial learning curve in the first and second levels, with Level 3 and Level 4 presenting less of a challenge as players begin to understand the game. The introduction of *if* statements in Level 5, the *if, else if* in Level 7, and the introduction of shapes and colours in Level 8 present more of a challenge to players. As may be seen from Figure 8, these levels caused a drop in the number of successful students to about 90%. The final level (see Figure 6(a)) was purposefully designed to be more difficult, and this is represented in Figure 7 by the steep increase in both time and number of attempts. On average, a time of about seven minutes was required to complete the level, and it was attempted on average three times. As evidenced in Figure 8, only 80% of the participants were able to finish the game. Considering that this was many of the participants' first exposure to programming, this figure is quite good and indicates a high audience and task suitability. However, the test group was small, and consisted of many people who play games often. A larger test group with a more divergent background is thus required for more a more accurate analysis of audience suitability. From Figure 7, it is possible to see that the initial learning curve was followed by a dip in perceived difficulty, whereas Level 5 and onward rose in difficulty. From this, and from open-ended feedback obtained, it is possible that these levels introduce too many new concepts too quickly. Ideally, the game should contain increases in difficulty followed by decreases as the players fully understand any one programming concept. There should, however, be a general trend of increase in the game's difficulty as players have to combine and synthesise multiple learnt concepts.

Qualitative Results

Figures 9 and 10 show the qualitative results from the questionnaires. As Figure 9 shows, the audience was average (poor to good) in terms of their programming experience, but had a lot of gaming experience (good to very good). This is important to note, since an audience with a lower interest in games would be expected to result in a steeper initial learning curve. Overall, the game was judged useful, easy to learn, and easy to remember. Figure 10 shows the results of a semantic differential to judge the quality of the game. The results from the questionnaires are averaged and presented on a symmetric scale from -5 (being poor) to 5 (being good). The game was deemed +3.7 in terms of fun, and +4.2 in terms of how rewarding it is. From open-ended feedback, some participants found that the game presents too many on-screen hints. Thus, the score for how creative players felt in the game is only +2.2. The graphical quality was determined to be +1.2. The sound was rated the poorest element of the game, at -3.0; however, it is noteworthy that none of the participants used headphones or speakers. This negative result thus indicates that the feedback obtained is reliable. The user interface was rated at +1.4 since some participants found it cluttered and confusing.

3.4 Lessons Learnt

It is vital to give players a sense of their progress within the game's levels. Otherwise, the player can feel disoriented and is unsure of how much longer they are required to play. The actual visualisation process should allow for immediate visualisations, so that there is minimal delay in a player's seeing the effect related to the cause of their programming. These visualisations should also be distinct and unambiguous. A software visualisation tool aimed at novice programmers should also allow for a high level of subject matter ignorance. However, it is impossible to cater to the entire range of possible students. Thus, there will always be a portion of the group that feels the game is boring and too instructive, and another that feels overwhelmed and confused. In terms of educational psychology, such games should make use of spaced repetition with added constraints. For instance, by replaying an older level with an added time limit, the player is forced to recall learnt knowledge, thus testing understanding and aiding retention. Also, the player requires a quick reference system to learnt knowledge such as the syntax of a particular programming construct.

3.5 Further and Future Work

Subsequent work has been performed on this research. The game has been developed further by improving the graphics and introducing the *and* operator. A familiar theme of traffic has also been applied to the game – players are now required to direct vehicles of different colour and types to the desired parking lots. The analytics system has also been developed to capture participant backgrounds (such as their level of education and gender), and the number of lines of code that are used to solve each level. The game was also made longer to account for the steep learning curve discussed in Section 3.3.2. A multiple choice test was designed and completed by the participants both before and after playing the game. Thus, the influence of the game on students' understanding of software concepts was measured. When tested on a group of 14 applied computing students (who were already proficient at computer programming), no increase in marks was noted. When tested on a group of 32 first-year electrical engineering students (who had no previous exposure to programming at university), an average increase in mark of 10% was measured. Of the 32 students, 20 increased

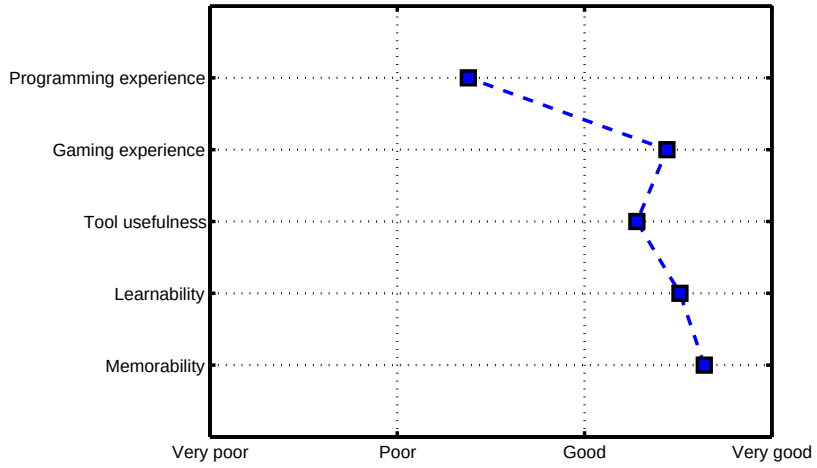


Figure 9: Likert scale of the audience and their perception of the game

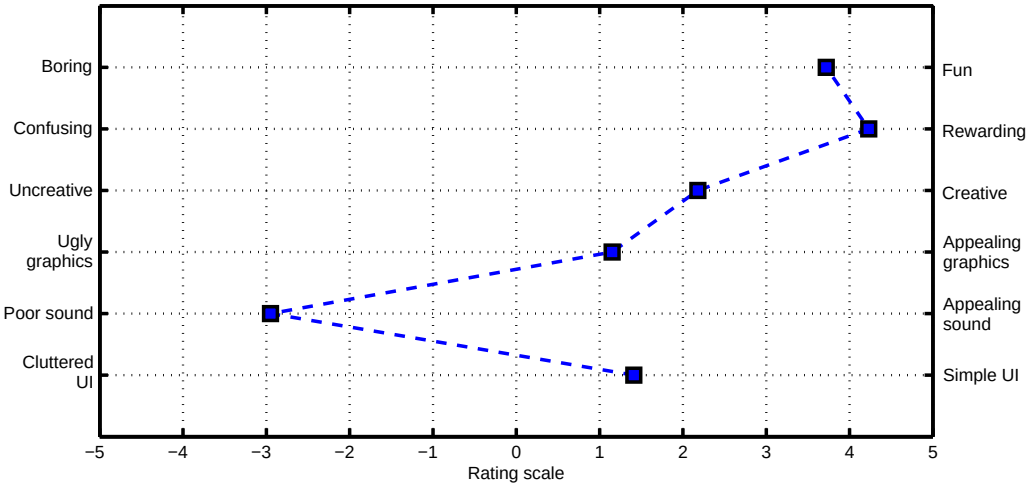


Figure 10: Semantic differential of the game

their marks, 6 remained the same, and 6 did poorer after playing the game. The game is also to be tested on school-level students in the future. This educational game is to be integrated into the course management system *Moodle* to provide a game-based learning environment.

4. CONCLUSION

Software visualisation is the rendering of data about a software system into a visual format. Software visualisation techniques may be used to enhance code comprehension by showing the static design of software systems or their dynamic runtime behaviour. Such techniques are thus employed in the professional software development industry, but can also be applied to the academic teaching of computer science. Many applications such as *Alice* and *RoboMind* are used to teach novice programmers at both high school and university levels. Educational games like *RoboZZle* and *VIMAdventures* are also used to visualise software and augment traditional computer science education. Such games encompass the areas of representation and medium in a taxonomy of software visualisation tools; they are highly reliant on graphical and audio representation and need to consider interactivity with the material they represent. An educational computer game, *Conveyor*, has been designed and developed. The system consists of an online game and an integrated analytics system. The game consists of nine levels of increasing difficulty in which players need to program the solution to sort random inputs into output bins. The analytics system collects data about how players engage with the game. The time taken per level and the number of reattempts are collected and stored.

The game was tested on a group of 39 participants consisting of first-year game design students and second-, third-, and fourth-year electrical, information and biomedical engineering students. The group represented an overall average exposure to computer programming, but a high exposure to and interest in computer games. 80% of participants were able to complete the entire game. The time spent on and number of attempts at each level indicates a initially slow learning curve, with too many new concepts being introduced in higher levels. Qualitative feedback shows that the game is fun and rewarding, but has

poor graphical and audio presentation. The user interface was also deemed to be quite cluttered and confusing. This feedback was used to develop the game further. In subsequent work, the analytics system was expanded to capture the number of lines of code that players use in their solution of a level. The game was also made longer, and displayed a better learning model. In order to judge the effect of the game on students' marks in a traditional test environment, a simple software quiz was administered to subsequent test groups before and after the game was played. When tested on a group of 14 applied computing students, who already had previous exposure to computer programming, no increase in marks was noted. However, in a group of 32 electrical engineering students, 63% of students increased their marks by an average of 10%. This investigation shows that games do have the potential of supporting traditional education and providing an enjoyable learning tool that can stimulate student interest in computer programming.

5. ACKNOWLEDGMENTS

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to the NRF. The authors also acknowledge the participants that took part in the research testing.

6. REFERENCES

[1] Alice.org. http://www.alice.org/: [Last accessed 4th February 2012].

[2] Kodu - Microsoft Research. http://research.microsoft.com/en-us/projects/kodu/ [Last accessed 10th May 2012].

[3] Lightbot. http://www.newgrounds.com/ [Last accessed 15th March 2012].

[4] Robomind.net. http://www.robomind.net/en/index.html [Last accessed 15th March 2012].

[5] Robozzle online puzzle game. http://robozzle.com/ [Last accessed 15th March 2012].

[6] Simse online. http://www.ics.uci.edu/~emilyo/SimSE/ [Last accessed 15th May 2012].

[7] Unity - game development tool. http://unity3d.com/unity/ [Last accessed 16th May 2012.

[8] Vim adventures. http://vim-adventures.com/ [Last accessed 10th May 2012].

[9] R. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. *Software Visualization: Programming as a Multimedia Experience*, pages 369–381, 1998.

[10] T. Ball and S. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[11] T. Barnes, H. Richter, A. Chaffin, A. Godwin, E. Powell, T. Ralph, P. Matthews, and H. Jordan. Game2learn: A study of games as tools for learning introductory programming concepts. *Proceedings of the ACM SIGCSE*, 7, 2007.

[12] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493. ACM, 2000.

[13] M. Eisenstadt, B. Price, and J. Domingue. Software visualization as a pedagogical tool. *Instructional Science*, 21(5):335–364, 1992.

[14] L. Feijs and R. De Jong. 3d visualization of software architectures. *Communications of the ACM*, 41(12):73–78, 1998.

[15] K. Maillet and M. Porta. Consequences of the declining interest in computer science studies in europe. In *Education Engineering (EDUCON), 2010 IEEE*, pages 71–76. IEEE, 2010.

[16] J. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40. IEEE, 2002.

[17] A. Marcus, L. Feng, and J. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff. ACM, 2003.

[18] M. Mayo. Games for science and engineering education. *Communications of the ACM*, 50(7):30–35, 2007.

[19] Microsoft Research. Transforming computer science in the gaming age. Microsoft External Research, 2008.

[20] A. Mitchell and C. Savill-Smith. The use of computer and video games for learning: A review of the literature. 2004.

[21] M. Pettre and E. de Quincey. A gentle overview of software visualisation. *The Computer Society of India Communications & Autumn 2006 Psychology of Programming Interest Group Newsletter*, 2006.

[22] M. Porta, K. Maillet, and M. Gil. Dec-cs: The computer science declining phenomenon. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 2, 2010.

[23] B. A. Price, I. S. Small, and R. M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1998.

[24] T. Susi, M. Johannesson, and P. Backlund. Serious games–an overview. *Skövde: University of Skövde (Technical Report HS-IKI-TR-07-001)*, 2007.

[25] M. Tory and T. Möller. Human factors in visualization research. *IEEE Transactions on Visualisation and Computer Graphics*, 10:72–84, 2004.

[26] P. Young and M. Munro. Visualising software in virtual reality. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 19–26. IEEE, 1998.

APPENDIX C

IGIC 2012 Conference Paper

This appendix contains the paper that was presented at the IEEE IGIC (International Games Innovation Conference) in Rochester, New York on the 7th of September, 2012. The paper was peer-reviewed and accepted to the conference, which had an acceptance rate of 69%. All three iterations of the research had been completed when the paper was written. The paper focuses on the design of the educational game, the design of the testing framework and the presentation of the results obtained.

Video Games as a Medium for Software Education

Bradley R C Marques
School of Electrical and
Information Engineering
The University of the Witwatersrand
Johannesburg, South Africa
Email: bradley.marques@students.wits.ac.za

Stephen P Levitt
School of Electrical and
Information Engineering
The University of the Witwatersrand
Johannesburg, South Africa
Email: stephen.levitt@wits.ac.za

Ken J Nixon
School of Electrical and
Information Engineering
The University of the Witwatersrand
Johannesburg, South Africa
Email: ken.nixon@wits.ac.za

***Abstract*—Educational video games may be used as a medium for software visualisation and visual programming to provide highly enjoyable, self-motivating and inquiry-based pedagogical tools. An educational game has been developed and tested on university-level students in three iterations. Players are required to solve puzzles by programming the solutions; the game introduces syntax, conditional statements and logical operators. An integrated analytics system is used to store the time taken, the number of lines of code, and players' solutions to each level. Qualitative feedback indicates that the tool is very easy to learn because of the help system and user interface. A software quiz was administered before and after participants played the game. When tested on 14 applied computing students (who had formal exposure to programming), there was no increase in the average grade. In a group of 32 electrical engineering students (who had no exposure to programming at university), the game helped about 60% of participants increase their grade, by an average of 11%.**

I. Introduction

There is a globally declining interest in computer science and engineering, as fewer young people are being attracted to study within these fields [1]–[3]. However, concurrent with this is a continual rise in the popularity and consumption of video games amongst a growing demographic of gamers [1]. Many students are visual learners [4]. Software visualisation – which is the rendering of intangible data about a software system into a visual format – can thus aid a student's understanding of software [4], [5]. As is discussed, educational games may be used to provide visual programming environments to teach software development in a way that is engaging, fun, and relatable to students' experiences [6], [7].

This research examines how digital games can be used as a medium to support traditional software pedagogy and provides a simple validation framework. Digital games are understood from a software visualisation perspective, and their application in this field is contextualised within an existing taxonomy (see Section II). A game that teaches a few basic concepts of computer programming was iteratively developed and tested on university-level students. This puzzle game, shown in Figure 1, requires players to programme traffic lights to direct traffic to the correct locations (see Section III). Section IV examines the research methodology and testing framework, and the results are presented and analysed in Section V.

II. Literature Review

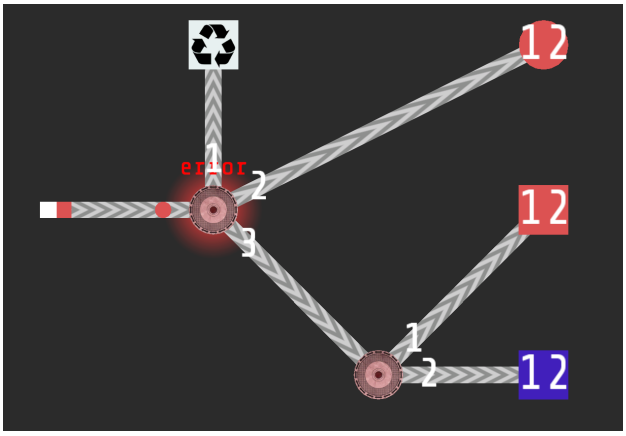
A. Software Visualisation and Visual Programming

Software visualisation is the abstraction of data about an existing software system into a visual format [6], [7]. Visual programming is the creation of software systems through a visual medium [6]. The visualisation of software facilitates rapid code comprehension. Software visualisation tools are thus vital in the professional software development industry, but can also be used for software pedagogy. A task-oriented taxonomy identifies five areas of software visualisation techniques: task, audience, target, representation and medium [7]. The areas of representation and medium define aspects such as the interactivity of the data, the use of sound and graphics, the method of abstraction, and visual metaphor. Thus, an interactive educational computer game – with the primary purpose of facilitating code comprehension – focuses on these areas of software visualisation.

B. Educational Games for Software Pedagogy

Pedagogical software visualisation tools range from instructional films – such as *Sorting out Sorting* – to more interactive, open-ended learning environments [5]. *Alice*, for instance, allows students to create and animate simple scenes using drag-and-drop commands and introduces basic programming concepts such as variables, looping and functions [8]. *RoboMind* presents an open-ended virtual world in which students program a robot to perform certain tasks [9]. The games *RoboZZle* and *LightBot* present similar environments, with a heavier emphasis on puzzle game mechanics [10], [11]. These didactic games support logical thinking and good design skills, two vital components of software development and engineering [12]. *SimSE* is a role-playing game in which players learn various software engineering life cycle models in a simulated office environment [13]. *VIMAdventures* is an adventure game that teaches players the controls to the popular text-editor, VIM [14].

Whilst most researchers agree that games cannot *replace* traditional lecturing and tuition, many believe that games can be used to address many of the underlying issues of the declining interest in science and engineering – such as student apathy – and may thus be used to *augment* traditional teaching of software development [1]. Games offer some unique



(a) Screenshot of the game during Iteration 1

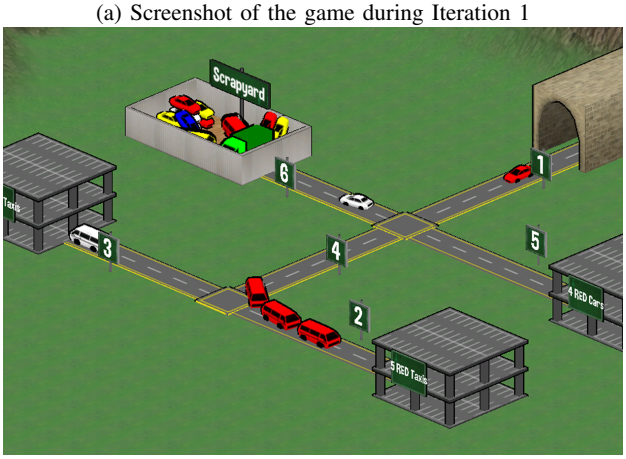
Node 1 (top left)

```
if(shapeIsSquare())
     outputTo(3);
else if (shapeIsRed())
     outputTo(2);
else
     outputTo(1);
```

Node 2 (bottom right)

```
if(shapeIsRed())
     outputTo(1);
else
     outputTo(2);
```

(b) Solution to the level in Figure 1a



(c) Screenshot of the game during Iterations 2 and 3

Intersection 1 (top right)

```
if(taxi())
     drive(4);
else if (red())
     drive(5);
else
     drive(6);
```

Intersection 2 (bottom left)

```
if(red())
     drive(2);
else
     drive(3);
```

(d) Solution to the level in Figure 1c

Fig. 1: Screenshots of an equivalent level in Iteration 1 and 2, and example code that would solve the levels

benefits over other means of tuition: they have the ability to reach many students whilst bypassing the – often obstructive – nature of educational systems; because of their reward value, they provide effective learning paradigms of experiential and inquiry-based learning. Whilst providing concrete goals, they challenge players at the edge of their growing realm of capability, which is a good model for an educational tool [1].

III. Overview of the Game Design

The designed game is both a visual programming environment and a software visualisation tool since it allows players to create code and visualise its effect. *Conveyor*, the first iteration of the game is shown in Figure 1a. Players are required to solve puzzles by sorting input objects of different shape and colour into the correct output bins. They do this by programming the nodes in a conveyor belt system with various conditional statements, such as an *if, else if, else* structure. Figure 1b shows an example of code that can be used to solve a level of the game. Players thus learn basic programming syntax, code flow control and logical thinking. In-game help texts guide the player through 9 levels of increasing difficulty and a quick reference guide is available to check syntax of each programming statement.

Similarly, the second iteration, *if(traffic)else{win}*, shown in Figure 1c, requires players to direct vehicles of different type and colour to the correct parking lots by programming the traffic lights at each intersection of a simple road network. Apart from introducing a theme and improved graphics, the second iteration is longer (with 15 levels) and introduces the *&&* operator. The game of Iteration 3 was simply a shorter version of *if(traffic)else{win}*, consisting of 10 levels. As may be seen in Figure 1, the programming style was simplified from the first iteration. Players are guided by both in-game text hints and help screens that provide example code and hints. In both iterations of the game, players receive immediate visual feedback indicating the effect of their typed code. Both games also indicate through sound and graphics where syntax errors occur. Iterations 2 and 3 provide natural language descriptions of these errors. When a player completes a level of *if(traffic)else{win}*, they receive a summary of their performance in terms of the time taken and the number of lines of code (LOC) used. The data is collected and stored alongside other useful data by the in-game analytics system. This system and the research testing framework are discussed in the following section.

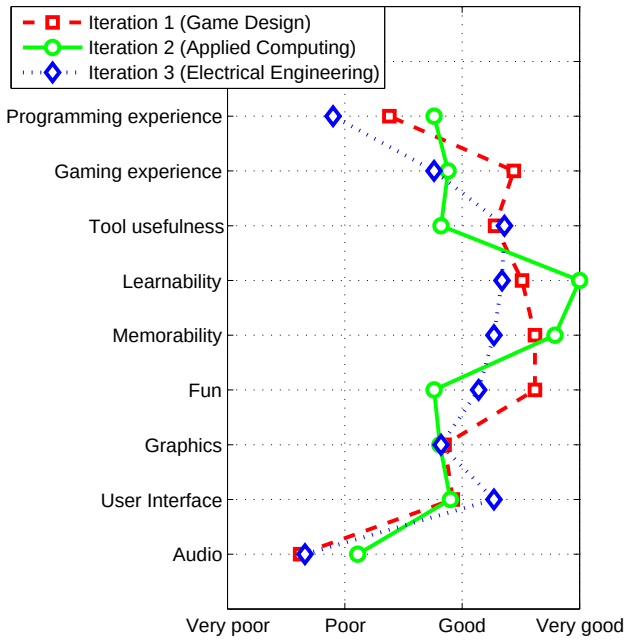


Fig. 2: Likert scale of the qualitative feedback obtained from all three iterations

IV. Research Methodology and Testing Framework

Iteration 1 was tested on a group of 39 participants consisting of first- to fourth-year electrical, information and biomedical engineering students and first-year game design students from both engineering and arts disciplines. For this iteration, the analytics system kept track of level times, but not LOC. Qualitative feedback was obtained through a questionnaire that allowed participants to judge the game's entertainment factor, its perceived usefulness, the quality of presentation and the learnability and retention of the material covered. Iteration 2 was tested on a group of 14 first-year applied computing students, who had previous exposure to programming. A simple online software skills quiz was administered both before and after the game (see Section V). These quizzes were randomly generated from a pool of questions of various difficulty levels. The analytics system was improved to capture level times, LOC and participants' solutions to each level. The same survey was used for qualitative feedback. Iteration 3 was tested on a group of 32 first-year electrical engineering students – with no formal exposure to programming – by the same method as Iteration 2.

V. Results and Analysis

Figure 2 shows the survey results. As may be expected, the applied computing students rated their programming experience higher than the other groups, and the game design students had the highest gaming experience. The tool was rated less useful by the applied computing group. All participants said that they would recommend the game to people who were learning to programme. Notable is the decrease in the rating of the entertainment value in Iteration 2. It is believed that this is because of the length of the second game (50% of the audience felt that it was too long), and because the audience was more familiar with programming. As mentioned, this was remedied in the third iteration, in which only 1 participant felt that the game was too long. The graphics and user interface received similar ratings in all iterations. This was unexpected, since these factors improved (see Figure 1). Participants did not listen to the audio, but the result is included to show that the audience was unbiased.

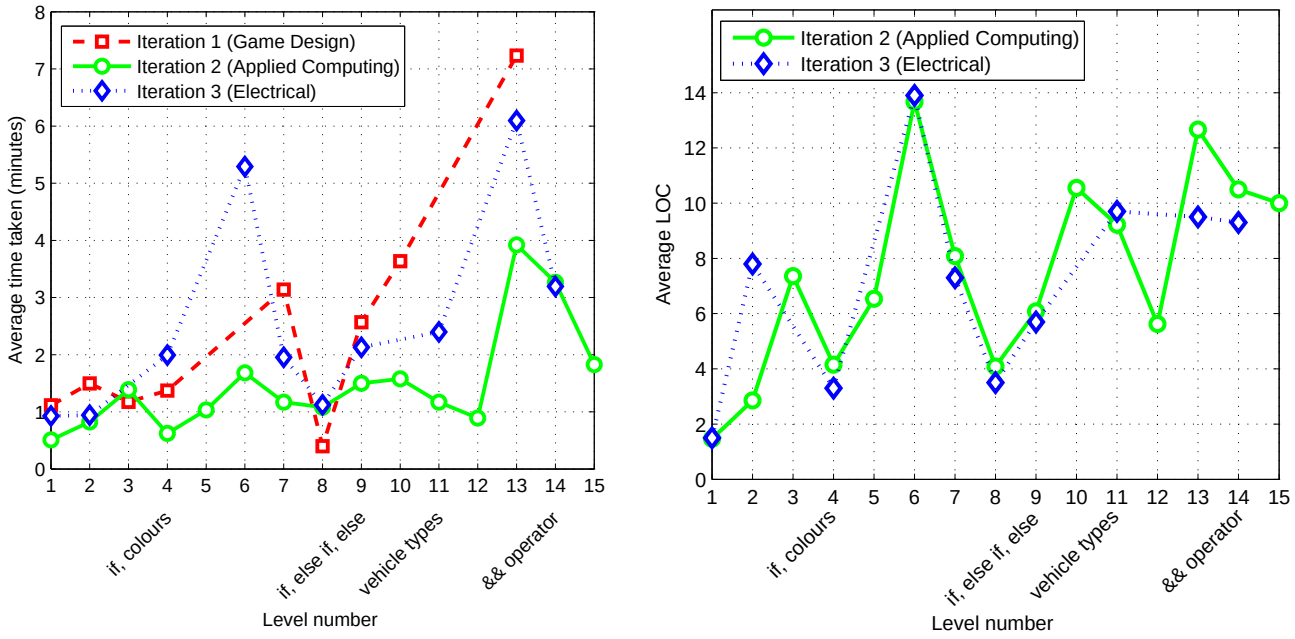
Figure 3a shows the times taken for each level in both iterations. As mentioned, each iteration had a different number of levels; each point in Figure 3a thus identifies equivalent levels in the different iterations. The average level time for the game design students was 2 minutes 27 seconds; for the applied computing group, 1 minute 30 seconds; and for the electrical engineers, 2 minutes 36 seconds. As expected, the students with more programming experience completed levels faster. Gaming experience had no effect on the level times. By assuming that the time taken per level is an indication of the game's difficulty, it may also be seen that Iteration 2 has a better learning model. Iteration 1 has a fairly linear increase in the time taken per level; players were not given adequate time between the introduction of new concepts to fully integrate them into their skill set. Iteration 2, on the other hand, shows a rising and falling of level difficulty, corresponding to the introduction of new concepts and the player learning and adopting these concepts in future, similar problems.

Figure 3b shows the average LOC used for each level in Iterations 2 and 3. As may be seen, this is highly correlated to the time taken. Level 6 shows a sharp increase in the time required, but only a slight increase in the LOC. This is due to the nature of the level: many cascaded *if, else* statements. This may be an indication of a boring level, but one participant mentioned that it was her favourite. Level 13, which is depicted in Figure 1, does not explicitly introduce a new concept, but requires a little more logical thought than other levels. As shown in Figure 1, players are required to understand that the order in which an *if, else if, else* statement is written is crucial. This concept is never explained, and thus the level presents a challenge.

As mentioned, participants of the two latter iterations completed a software quiz before and after playing the game. The applied computing students showed no increase of their average grade of 84%. The electrical engineering students showed an increase of 11% above their original average grade of 50%. Of the 32 participants, 20 improved their grades. Thus, it may be concluded that this game is best suited for novice students, and can provide a fun, augmentative lesson in novice computer programming skills.

VI. Conclusion

An educational puzzle game, *if(traffic)else*{*win*}, which teaches basic programming syntax, logical thinking and conditional statements has been iteratively designed and tested. The



(a) Average time taken per level in all three iterations

(b) Average LOC used per level in Iterations 2 and 3

Fig. 3: Time taken and LOC used per level

game was tested on a group of 39 engineering and game design students, 14 applied computing students, and 32 electrical engineering students. Participants of the latter 2 iterations completed a graded software quiz before and after playing the game. The applied computing students, who already had programming experience, showed no increase in their average grade of 84%. 62% of the inexperienced electrical engineering students increased their grade of 50% by 10.9%. It was found that a longer game provided a better learning curve, but was deemed boring. The LOC required to solve levels was highly correlated to the time taken. The portrayal of game rules and features was improved in the second and third iterations, and participants rated this game easier to learn. All participants said that they would recommend the game to novice programmers. From informal feedback, it was also noted that the majority of students were excited with the game. The game thus provided a fun, motivating and exploratory means of learning a few basic concepts of software development.

ACKNOWLEDGMENT

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

REFERENCES

[1] M. Mayo, "Games for science and engineering education," *Communications of the ACM*, vol. 50, no. 7, pp. 30–35, 2007.

[2] K. Maillet and M. Porta, "Consequences of the declining interest in computer science studies in europe," in *Education Engineering (EDUCON)*. IEEE, 2010, pp. 71–76.

[3] M. Porta, K. Maillet, and M. Gil, "Dec-cs: The computer science declining phenomenon," in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 2, 2010, pp. 1173–1178.

[4] M. Eisenstadt, B. Price, and J. Domingue, "Software visualization as a pedagogical tool," *Instructional Science*, vol. 21, no. 5, pp. 335–364, 1992.

[5] R. Baecker, "Sorting out sorting: A case study of software visualization for teaching computer science." The MIT Press, Cambridge, MA, 1998, pp. 369–381.

[6] B. A. Price, I. S. Small, and R. M. Baecker, "A principled taxonomy of software visualization," *Journal of Visual Languages and Computing*, vol. 4, pp. 211–266, 1998.

[7] J. Maletic, A. Marcus, and M. Collard, "A task oriented view of software visualization," in *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*. IEEE, 2002, pp. 32–40.

[8] "Alice.org," http://www.alice.org/: [Last accessed 4th February 2012].

[9] "Robomind.net," http://www.robomind.net/en/index.html [Last accessed 15th March 2012].

[10] "Robozzle online puzzle game," http://robozzle.com/ [Last accessed 15th March 2012].

[11] "Lightbot," http://www.newgrounds.com/ [Last accessed 15th March 2012].

[12] J. Sarama and D. Clements, "Building blocks and cognitive building blocks: Playing to know the world mathematically," *Americal Journal of Play, Volume 1, Number 3*, pp. 1–25, 2009.

[13] "Simse online," http://www.ics.uci.edu/ emilyo/SimSE/ [Last accessed 1st May 2012.

[14] "Vim adventures," http://vim-adventures.com/ [Last accessed 1st May 2012.

APPENDIX D

Software Skills Quiz

D.1 Introduction

To judge the usefulness of an educational game designed to teach some basic computer programming concepts, a software skills quiz was developed. The test was administered to research participants before and after they had played the designed game (the reader is referred to the main dissertation text for more details). The quiz was developed using the online survey tool, `LimeSurvey`. It consists of 24 possible multiple-choice questions that cover concepts taught by the game, but was randomised for each student and asked only 12 of these questions. This randomisation eliminated bias in the before-and-after tests' difficulty levels. Questions are organised into different categories based on their difficulty. The questions are written in pseudo-code, and focus more on the concepts covered (such as conditional statements and logical operators) than any language-specific syntax. This appendix presents the quiz questions and answers.

D.2 Software Skills Quiz

This section lists each of the 24 multiple-choice questions, and gives the possible answers. The correct answer is shown in bold.

1. In the following code, there is an error. On which line is the error found?

```
1 if (age >= 65);
```

```
2 print "Discount";
3 else
4 print "No discount";
```

(a) **Line 1**

(b) Line 2

(c) Line 3

(d) Line 4

2. In the following code, there is an error. On which line is the error found?

```
1 if (gender == "Male")
2 print "It's a boy!";
3 else print "It's a girl!"
```

(a) Line 1

(b) Line 2

(c) **Line 3**

3. On which line of the following code is there an error?

```
1 if (x == 10)
2 print "ten";
3 else if (x < 10)
4 "smaller";
5 else print "larger";
```

(a) Line 1

(b) Line 2

(c) Line 3

(d) **Line 4**

(e) Line 5

4. On which line of the following code is there an error?

```
1 if (x < y)
```

```
2 print = x less than y;
3 else if (x == y)
4 x = x + 1;
5 y = y + x;
```

(a) Line 1

(b) **Line 2**

(c) Line 3

(d) Line 4

(e) Line 5

5. What is the output of the following code?

```
1 x = 8;
2 if (x > 10)
3 print "big";
4 else print "small";
```

(a) big

(b) **small**

(c) There is no output

6. What is the output of the following code?

```
1 x = 10;
2 if (x > 10)
3 print "big";
4 else print "small";
```

(a) big

(b) **small**

(c) There is no output

7. What is the output of the following code?

```
1 x = 20;
```

```
2 if (x > 10)
3 print "big";
4 else print "small";
```

(a) **big**

(b) small

(c) There is no output

8. What is the output of the following code?

```
1 x = 5;
2 y = 9;
3 if (x > y)
4 print "one";
5 else
6 print "two";
```

(a) one

(b) **two**

(c) There is no output

9. What is the output of the following code?

```
1 x = -7;
2 y = -7;
3 if (x > y)
4 print "two";
5 else
6 print "one";
```

(a) **one**

(b) two

(c) There is no output

10. What is the output of the following code?

```
1 x = 55;
2 y = 22;
3 if (x < y)
4 print "Hello";
5 else print "World";
```

(a) Hello

(b) **World**

(c) Hello World

(d) There is no output

11. What is the output of the following code?

```
1 x = 40;
2 y = 70;
3 if (x < 20)
4 print "one";
5 else print "two";
6 if (x < y)
7 print "three";
8 else print "four";
```

(a) one three

(b) one four

(c) **two three**

(d) two four

(e) There is no output

12. What is the output of the following code?

```
1 x = -8;
2 y = -10;
3 if (x < -10)
4 print "one";
5 else print "two";
```

```
6 if (x > y)
7 print "three";
8 else print "four";
```

(a) one three

(b) one four

(c) **two three**

(d) two four

(e) There is no output

13. What is the output of the following code?

```
1 x = 19;
2 y = 24;
3 if (x >= 50)
4 print "Hello";
5 if (y >= 50)
6 print "World";
```

(a) Hello

(b) World

(c) Hello World

(d) **There is no output**

14. What is the output of the following code?

```
1 x = 19;
2 y = 20;
3 if (x >= 20)
4 print "Hello";
5 if (y >= 20)
6 print "World";
```

(a) Hello

(b) **World**

(c) Hello World

(d) There is no output

15. What is the output of the following code?

```
x = 15;
y = 34;
if ((x > 15) && (y > 30))
print "one";
else if (y < 50)
print "two";
else print "three";
```

(a) one

(b) **two**

(c) three

(d) one two

(e) two three

(f) one three

16. What is the output of the following code?

```
bob = 15;
mary = 34;
if((bob > 15) && (mary > 50))
print "Hi";
else if (bob > 15)
print "Bob and";
else if (mary > 34)
print "Mary";
```

(a) Hi

(b) Hi Bob and

(c) Hi Mary

(d) Hi Bob and Mary

(e) **There is no output**

17. What is the output of the following code?

```
1  a = 10;
2  b = 6;
3  if ((a > b) && (b > 3))
4  print "one";
5  else if (a < b)
6  print "two"
7  if (a > b)
8  print "three";
9  else if (b >= 10)
10 print "four";
11 else print "five";
```

(a) one four five

(b) one four

(c) **one three**

(d) two three four

(e) two five

(f) two four five

18. What is the output of the following code?

```
1  a = 9;
2  b = 9;
3  if ((a > b) && (b > 3))
4  print "one";
5  else if (a < b)
6  print "two"
7  if (a > b)
8  print "three";
```

```
9  else if (b >= 10)
10 print "four";
11 else print "five";
```

(a) one three

(b) two three

(c) one three five

(d) two four five

(e) four

(f) **five**

(g) There is no output

19. What is the output of the following code?

```
1  a = 90;
2  b = 10;
3  if ((a > b) && (b > 30))
4  print "one";
5  else if (a < b)
6  print "two"
7  if (a > b)
8  print "three";
9  else if (b >= 10);
10 else print "five";
```

(a) four

(b) **three**

(c) five

(d) two three

(e) two four

(f) two five

20. What is the output of the following code?

```
1 bob = 10;
2 jane = 6;
3 if ((bob < jane) && (jane < 3))
4 print "Happy";
5 else if (bob < jane)
6 print "Birthday";
7 if (bob > jane)
8 print "To";
9 print "You";
```

(a) Happy To You

(b) Happy To

(c) Birthday To

(d) Birthday You

(e) Birthday To You

(f) **To You**

21. What is the output of the following code?

```
1 x = 20;
2 if (x >= 20)
3 {
4   if (x < 20)
5   print "one";
6   else print "two";
7 }
8 print "three";
```

(a) **two three**

(b) three

(c) one three

(d) one two

(e) one

(f) There is no output

22. What is the output of the following code?

```
x = 10;
if (x >= 20)
{
  if (x < 20)
  print "one";
  else print "two";
}
print "three";
```

(a) two three

(b) **three**

(c) one three

(d) one two

(e) one

(f) There is no output

23. What is the output of the following code?

```
x = 12;
if (x > 12)
{
  if (x < 15)
    print "one";
}
else
{
  print "two";
  print "three";
}
```

(a) one two three

(b) one two

(c) one three

(d) two

(e) **two three**

(f) three

24. What is the output of the following code?

```
1  x = 12;
2  if (x >= 12)
3  {
4    if (x < 15)
5      print "one";
6  }
7  else
8  {
9    print "two";
10 }
11 print "three";
```

(a) one

(b) one two

(c) **one three**

(d) one two three

(e) two three

(f) three

(g) There is no output

D.3 Conclusion

A software skills test was developed as a means of judging the pedagogical value of an educational computer game. The reader is referred to Chapter 4 of the main dissertation for more details.

APPENDIX E

Survey

E.1 Introduction

An educational computer game that teaches some basic concepts of computer programming was designed and developed as part of Masters-level research. In order to collect qualitative data about the game, a survey was administered after research participants played the game. This appendix documents each survey question and their possible answers. The survey was designed to be administered to a wider audience (such as school children and working people) than was ultimately tested. The survey was created with a survey creation tool, `LimeSurvey`. The purpose of the survey was to collect user experience data – what players felt about the game – and to profile the audience. This data is used to judge the quality of the developed game – the reader is referred to Chapter 4 of the main dissertation text for more details.

E.2 Survey

E.2.1 Personal Background

These questions were asked to profile the audience:

1. Which of the following describes you best?
   (a) School student
   (b) Undergraduate university student

   (c) Postgraduate university students

   (d) Employed

   (e) Other

2. What is your student number? (Only asked if postgraduate or undergraduate student)

3. What grade are you in? (Only asked if school student)

4. Do you do IT or computer studies at school? (Only asked if school student)

   (a) Yes

   (b) No

5. Under what faculty are you registered in your university? (Only asked if postgraduate or undergraduate student)

   (a) Engineering and Architecture

   (b) Science

   (c) Humanities

   (d) Health Sciences

   (e) Commerce Law and Management

6. What degree are you doing? (Only asked if Engineering and Architecture student)

   (a) Electrical, Information or Biomedical Engineering

   (b) Mechanical, Industrial, Aeronautical, Mining or Chemical Engineering

   (c) Computer Science or Applied Computing

   (d) Game Design (Engineering Branch)

   (e) Game Design (Performing and Visual Arts Branch)

   (f) Other

7. To what level have you successfully completed Software Development? (Only asked if Engineering and Architecture student)

   (a) Software Development I

   (b) Software Development II

   (c) Software Development III

8. What year are you in? (Only asked if undergraduate student)

9. What is your job title? (Only asked if employed)

E.2.2 General Questions

These questions profile the audience in terms of the programming and gaming experience and also judges the audience's perception of educational games:

1. How would you rate your experience in computer programming?
   (a) Excellent - I am a professional programmer
   (b) Good - I have a fair amount of experience in programming
   (c) Alright - I am competent, but not great
   (d) Poor - I am a novice programmer
   (e) Terrible - I have never written a computer programme
2. How would you rate your experience in computer or video games?
   (a) Excellent - I play a lot of games
   (b) Good - I play games quite frequently
   (c) Alright - I play games casually
   (d) Poor - I hardly play games
   (e) Terrible - I never play games
3. Do you think that computer and video games can be educational?
   (a) Yes
   (b) No

E.2.3 Questions About the Game

These questions were asked to asses the user's experience with and perception of the developed game:

1. How useful was the tool in teaching you about computer programming and conditional statements (if, else if, else)?
   (a) Very useful - I learnt a lot
   (b) Useful - I learnt a few things
   (c) Useless - I did not learn much
   (d) Completely useless - I did not learn anything
2. Did you understand the goal of the game?
   (a) Yes

   (b) No

3. How easy was it to learn how the game worked (how to move around, what to click on, etc.)?
   (a) Very easy
   (b) Easy
   (c) Difficult
   (d) Very difficult
4. How easily can you recall what programming statements and logic you learnt through game?
   (a) Very easily
   (b) Easily
   (c) Poorly
   (d) Not at all
5. Do you think the game was of a suitable length?
   (a) Yes
   (b) No
6. Please rate the following aspects of the game (10 being the best, 1 being the worst):
   (a) Fun you had playing the game
   (b) Graphics
   (c) Sound
   (d) Music
   (e) User interface (menus, buttons, etc.)
7. Would you recommend this game to somebody who was learning computer programming?
   (a) Yes
   (b) No

E.2.4 Open Feedback

These open-ended questions allow players to give any suggestions:

1. What did you learn playing this game, if anything?

2. What features of the game do you think worked well or find useful and enjoyable?
3. What features of the game do you think are poorly done or need improvement?
4. I would really appreciate any other feedback or suggestions!

E.3 Conclusion

This appendix documents a survey that was administered to research participants after they had played an educational computer game. The survey is intended to both profile the audience (in terms of their educational level, gaming and programming experience) and collect qualitative user experience data. This data is used to judge the quality of the produced game – the reader is referred to Chapter 4 of the main dissertation text for more details.