

# Automated Testing and Validation of Computer Graphics Implementations for Cross-Platform Game Development

A thesis submitted to Auckland University of Technology in  
fulfilment of the requirements for the degree of Master of Philosophy (MPhil)

Supervisors

Dr. Stefan Marks

Anne Philpott

2017

By

Steffan Derek Hooper

Colab: Creative Technologies

# Abstract

Commercially released cross-platform video games often feature graphical defects, which negatively impact on the reputations of developers and publishers, as well as on the experience of the players. Game industry testing practices often rely on human testers to assure the quality of a game prior to release.

This thesis investigates the question of whether the testing and validation of computer graphics implementations for cross-platform game development can be automated to reduce the burden on human testers, accelerate the testing phase, and improve the quality of games. Using Design Science Research methods and patterns, iterative development and evaluation is undertaken to construct artefacts, drawing upon prior research and industrial works in related fields such as film and television, as well as proprietary game development insight. Elements of existing automated testing systems and image comparison techniques are combined with industry standard cross-platform development tools and methods to create a reusable and generalisable model of an automated test system, featuring image comparison methods, and record and playback techniques for cross-platform game implementations.

In conclusion, this thesis shows that it is possible to create a novel system that is able to detect graphical defects in output from a cross-platform game implementation.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Attestation of Authorship</b>	<b>viii</b>
<b>Acknowledgements</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	1
1.2 Research Motivation . . . . .	5
1.2.1 Graphical Bugs . . . . .	6
1.3 Research Questions . . . . .	8
1.4 Research Method . . . . .	9
1.5 A Brief History of Computer Graphics and Video Game Technology . . . . .	9
1.5.1 Computer Graphics APIs . . . . .	10
1.5.2 Middleware and Game Engines . . . . .	13
1.6 Summary . . . . .	15
<b>2 Literature Review</b>	<b>16</b>
2.1 Structure of the Literature Review . . . . .	16
2.2 Video Game Graphical Defects . . . . .	17
2.2.1 Examples of Graphical Defects in Commercial Games . . . . .	17
2.3 Automated Testing in Games . . . . .	33
2.3.1 Industry Insight: Unity Technologies . . . . .	34
2.3.2 Industry Insight: Electronic Arts . . . . .	41
2.3.3 Games-related Testing Automation Research . . . . .	44
2.4 Automated Testing in Related Fields . . . . .	47
2.5 Findings of Literature Review . . . . .	56
<b>3 Methodological Approach</b>	<b>58</b>
3.1 Introduction . . . . .	58
3.2 Methodology: Design Science Research . . . . .	59
3.2.1 Patterns . . . . .	63
3.3 Method: Research Project Design . . . . .	63
3.3.1 Awareness of Problem . . . . .	63
3.3.2 Suggestion . . . . .	64
3.3.3 Development . . . . .	65
3.3.4 Evaluation . . . . .	66
3.3.5 Conclusion . . . . .	68
3.4 Tools: Artefact Construction . . . . .	68
3.4.1 Industry Tools and Techniques for Cross-Platform Game Development . . . . .	68
3.4.2 Cross-Platform Development: The Layers of a Target Platform . . . . .	71
3.4.3 Image Comparison . . . . .	71
3.5 Limitations . . . . .	71
3.6 Summary . . . . .	72

<b>4</b>	<b>Artefact Development and Evaluation</b>	<b>74</b>
4.1	Introduction . . . . .	74
4.2	Awareness of Problem . . . . .	75
4.3	Suggestion . . . . .	75
4.4	Development . . . . .	76
4.4.1	Iteration 1 . . . . .	78
4.4.2	Iteration 2 . . . . .	83
4.4.3	Iteration 3 . . . . .	87
4.4.4	Iteration 4 . . . . .	93
4.4.5	Iteration 5 . . . . .	95
4.4.6	Iteration 6 . . . . .	101
4.4.7	Iteration 7 . . . . .	104
4.4.8	Iteration 8 . . . . .	113
4.5	Summary of Artefact Development and Evaluation . . . . .	115
<b>5</b>	<b>Discussion</b>	<b>116</b>
5.1	Overview . . . . .	116
5.2	Summary of Results . . . . .	116
5.3	Limitations . . . . .	119
<b>6</b>	<b>Conclusion</b>	<b>120</b>
6.1	Future Work . . . . .	121
	<b>References</b>	<b>122</b>
	<b>Glossary</b>	<b>132</b>
	<b>Appendices</b>	<b>135</b>
<b>A</b>	<b>Game Industry Testing</b>	<b>136</b>
A.1	Types of Testing . . . . .	137
A.2	Compliance Testing for Certification . . . . .	138
A.3	Industry Insight: Testing . . . . .	139
<b>B</b>	<b>Iteration 6: RobotWorld Assets</b>	<b>141</b>
B.1	Graphical Assets . . . . .	141
<b>C</b>	<b>Software Engineering Research Laboratory (SERL) Seminar 2014</b>	<b>142</b>
C.1	Seminar Presentation Slides . . . . .	142
<b>D</b>	<b>PikPok Developers' Conference 2015</b>	<b>154</b>
D.1	Conference Presentation Slides . . . . .	154

# List of Figures

1.1	Target platform block diagram. Adapted from “The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR, and AR” by J. Peddie, Copyright 2013. . . . .	3
1.2	<i>Assassin’s Creed Unity</i> (Ubisoft Montreal, 2014a) screen capture featuring a graphical defect where the character’s face is incorrectly rendered ([Assassin’s Creed Unity image screen capture], n.d.). . . . .	6
2.1	<i>NBA 2K13</i> (Visual Concepts, 2012) screen capture featuring a clipping issue where one character’s forearm penetrates the face of another character ([NBA 2K13 image screen capture], n.d.). . . . .	19
2.2	<i>Garry’s Mod</i> (Facepunch Studios, 2004) screen capture of a video demonstrating a z-fighting issue, where the wall in the centre of the frame flickers from the dark grey texture to a different lighter grey texture as the player moves (Exho, 2014). . . . .	20
2.3	<i>Battlefield 4</i> (EA Digital Illusions CE, 2013) screen capture featuring a rendering error where red spots are rendered ([Battlefield 4 image screen capture], n.d.). . . . .	21
2.4	<i>Batman: Arkham Knight</i> (Rocksteady Studios, 2015) screen capture featuring a rendering error where the in-game character has spherical items rendered around its eyes, as well as artifacts across the lower third of the frame ([Batman: Arkham Knight image screen capture], n.d.). . . . .	22
2.5	<i>Mafia II</i> (2K Czech, 2010) screen capture featuring a rendering error where shadowing is incorrectly rendered across half the frame ([Mafia II image screen capture], n.d.). . . . .	23
2.6	<i>Grand Theft Auto: San Andreas</i> (Rockstar North, 2004) screen capture featuring a rendering error where the in-game character on the right is missing its head ([Grand Theft Auto: San Andreas image screen capture], n.d.). . . . .	24
2.7	<i>Call of Duty: Modern Warfare 3</i> (Infinity Ward, 2011) screen capture featuring a rendering error where the in-game character is missing its head ([Call of Duty: Modern Warfare 3 image screen capture], n.d.). . . . .	25
2.8	<i>Rocky</i> (Rage Software, 2002) screen captures featuring graphical errors. . . . .	25
2.9	<i>Battlefield 3</i> (EA Digital Illusions CE, 2011) screen capture featuring a rendering error where the in-game character features an elongated neck ([Battlefield 3 image screen capture], n.d.). . . . .	26
2.10	<i>Fallout 4</i> (Bethesda Game Studios, 2015) screen capture featuring a rendering error where the player can see through the building on the left of the frame ([Fallout 4 image screen capture], n.d.). . . . .	27
2.11	<i>Age of Wonders III</i> (Triumph Studios, 2014) screen capture featuring a rendering error where red tooltip text is incorrectly rendered at the top-centre of the frame, revealing the secret location of an enemy army obscured behind clouds ([Age of Wonders III image screen capture], n.d.). . . . .	28
2.12	<i>Overwatch</i> (Blizzard Entertainment, 2016) screenshot comparison of the same scene rendered on two different graphics settings, showing that a player using a lower graphics setting can see more of the game environment than a player using a higher graphics setting, due to missing foliage. . . . .	28
2.13	<i>Rage</i> (id Software, 2011) screen capture featuring a texturing issue where the game world exhibits a checkerboard pattern ([Rage image screen capture], n.d.). . . . .	29

2.14	<i>Assassin's Creed IV: Black Flag</i> (Ubisoft Montreal, 2013) screen capture featuring blurred text in the heads-up-display of the user interface ([Assassin's Creed IV: Black Flag image screen capture], n.d.). . . . .	30
2.15	Unity graphics test result example ([Unity reference image, test result image, difference image], n.d.). . . . .	39
2.16	Unity <i>Graphics Tests Tool</i> screenshot ([Unity Graphics Tests Tool image], n.d.). . . . .	40
3.1	Activity framework for Design Science Research. Adapted from "On theory development in design science research: Anatomy of a research project" by B. Kuechler and V. Vaishnavi, European Journal of Information Systems, Volume 17, Issue 5, pp. 489-504. Copyright 2008 by Palgrave Macmillan. . . . .	60
3.2	Design Science Research Process Model. Adapted from "Design science research in information sciences" by V. Vaishnavi and B. Kuechler, Copyright 2013. . . . .	61
4.1	DSR artefact iteration overview. . . . .	77
4.2	Iteration 1: Test scene generation UML design. . . . .	79
4.3	Iteration 1: Test scene outputs: Coloured primitive rendering. . . . .	79
4.4	Iteration 1: Test scene outputs: Alpha blended coloured primitive rendering. . . . .	80
4.5	Iteration 1: Comparison tool UML design. . . . .	80
4.6	Iteration 1: Pipeline overview. . . . .	81
4.7	Iteration 1: Comparison tool difference image example. . . . .	82
4.8	Iteration 2: Comparison tool UML design. . . . .	83
4.9	Iteration 2: Comparison tool output example. . . . .	84
4.10	Input data: <i>Watch Dogs</i> (Ubisoft Montreal, 2014b), rendered scene at various graphical quality settings (Burnes, 2014). . . . .	85
4.11	Iteration 2: Comparison tool output example where the selected Compared tab shows the RGB difference between the High and Ultra graphical quality settings. . . . .	86
4.12	Iteration 3: Test scene generation: SDL multi-platform. . . . .	88
4.13	Iteration 3: Test scene created with C++ and SDL. . . . .	88
4.14	Screen captures of <i>FTL: Faster Than Light</i> (Subset Games, 2012) showing two different graphical bugs. . . . .	89
4.15	Screen captures of <i>N++</i> (Metanet Software, 2015) contrasting a correctly rendered game level with an incorrectly rendered game level. . . . .	90
4.16	Iteration 3: Test scene executing on four of five target platforms. . . . .	92
4.17	Iteration 4: Test scene generation: 3D mesh with procedural animation. . . . .	93
4.18	Iteration 4: Test scene 3D procedural animation capture, frame 200 example results. . . . .	94
4.19	Iteration 4: Test scene 3D procedural animation capture, frame 300 example results. . . . .	94
4.20	Iteration 5: Example of test results for cross-platform output generated by the Iteration 3 artefact. . . . .	97
4.21	Iteration 5: Example of test results for cross-platform output generated by the Iteration 4 artefact. . . . .	98
4.22	Iteration 5: Example of the difference image for cross-platform output generated by the Iteration 4 artefact. . . . .	99
4.23	Iteration 5: Example of the Overlay/Flip feature for cross-platform output generated by the Iteration 4 artefact. . . . .	100
4.24	Iteration 6: Test scene generation: TurtleWorld and RobotWorld. . . . .	102
4.25	Iteration 6: Example of cross-platform output from the line-based TurtleWorld, with state tracing text rendering. . . . .	103
4.26	Iteration 6: Example of cross-platform output from the sprite-based RobotWorld, with state tracing text rendering. . . . .	103
4.27	Iteration 7: Game clone with replay system. . . . .	105
4.28	Iteration 7: Example replay captures, showing every third, one-second framebuffer capture from the <i>Generic PC</i> build target. . . . .	106
4.29	Iteration 7: Example of cross-platform output featuring detection of graphical defect. . . . .	108
4.30	Iteration 7: Example of cross-platform output difference image, highlighting the difference between star placements in the two target platforms. . . . .	110
4.31	Iteration 7: Example of corrected cross-platform output, showing 100% SSIM result. . . . .	111

4.32	Iteration 7: Example of cross-platform output difference image for test scene 11. . . . .	112
4.33	Iteration 7: Example of cross-platform output Overlay/Flip comparison for test scene 11.	113
4.34	Iteration 8: Model: Automated testing and validation of computer graphics implemen- tations for cross-platform game development. . . . .	114
B.1	Iteration 6: RobotWorld sprite assets (Skene, 2014). . . . .	141
C.1	SERL Seminar (slide set 1 of 11). . . . .	143
C.2	SERL Seminar (slide set 2 of 11). . . . .	144
C.3	SERL Seminar (slide set 3 of 11). . . . .	145
C.4	SERL Seminar (slide set 4 of 11). . . . .	146
C.5	SERL Seminar (slide set 5 of 11). . . . .	147
C.6	SERL Seminar (slide set 6 of 11). . . . .	148
C.7	SERL Seminar (slide set 7 of 11). . . . .	149
C.8	SERL Seminar (slide set 8 of 11). . . . .	150
C.9	SERL Seminar (slide set 9 of 11). . . . .	151
C.10	SERL Seminar (slide set 10 of 11). . . . .	152
C.11	SERL Seminar (slide set 11 of 11). . . . .	153
D.1	PikPok Developers' Conference (slide set 1 of 21). . . . .	155
D.2	PikPok Developers' Conference (slide set 2 of 21). . . . .	156
D.3	PikPok Developers' Conference (slide set 3 of 21). . . . .	157
D.4	PikPok Developers' Conference (slide set 4 of 21). . . . .	158
D.5	PikPok Developers' Conference (slide set 5 of 21). . . . .	159
D.6	PikPok Developers' Conference (slide set 6 of 21). . . . .	160
D.7	PikPok Developers' Conference (slide set 7 of 21). . . . .	161
D.8	PikPok Developers' Conference (slide set 8 of 21). . . . .	162
D.9	PikPok Developers' Conference (slide set 9 of 21). . . . .	163
D.10	PikPok Developers' Conference (slide set 10 of 21). . . . .	164
D.11	PikPok Developers' Conference (slide set 11 of 21). . . . .	165
D.12	PikPok Developers' Conference (slide set 12 of 21). . . . .	166
D.13	PikPok Developers' Conference (slide set 13 of 21). . . . .	167
D.14	PikPok Developers' Conference (slide set 14 of 21). . . . .	168
D.15	PikPok Developers' Conference (slide set 15 of 21). . . . .	169
D.16	PikPok Developers' Conference (slide set 16 of 21). . . . .	170
D.17	PikPok Developers' Conference (slide set 17 of 21). . . . .	171
D.18	PikPok Developers' Conference (slide set 18 of 21). . . . .	172
D.19	PikPok Developers' Conference (slide set 19 of 21). . . . .	173
D.20	PikPok Developers' Conference (slide set 20 of 21). . . . .	174
D.21	PikPok Developers' Conference (slide set 21 of 21). . . . .	175

# List of Tables

2.1	Summary of example video games with rendering errors, their associated game engine and target platforms. . . . .	32
3.1	A target platform’s layered architecture. . . . .	71
4.1	Iteration 1: Target platform matrix. . . . .	78
4.2	Iteration 2: <i>Watch Dogs</i> test scenes RGB per-pixel comparison. . . . .	87
4.3	Iteration 2: <i>Watch Dogs</i> test scenes SSIM comparison. . . . .	87
4.4	Iteration 3: Target platform matrix. . . . .	91
4.5	Iteration 4: Example comparison output results. . . . .	95
4.6	Iteration 7: Automated comparison tool results. . . . .	109
4.7	Iteration 7: Automated comparison tool results, following defect repair. . . . .	112



# Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

A handwritten signature in black ink, consisting of stylized, overlapping loops and strokes, positioned above a horizontal line.

---

Signature of candidate

# Acknowledgements

My sincere thanks to my supervisors, Dr. Stefan Marks and Anne Philpott, for your support and advice throughout this research project. Your insights and experience have been invaluable.

To the School of Engineering, Computer and Mathematical Sciences, thank you for supporting my postgraduate studies by creating an environment that allows such research to occur.

To Dr. Stephen Thorpe and the late Graham Bidois, thank you for convincing me to join Auckland University of Technology, and for giving me the opportunity to pursue further study.

To Dr. James Skene, friend and colleague, provider of Robots and Pis, thank you for your honest insights into research and shared passion for games.

Thank you to Mario Wynands and Tyrone McAuley of PikPok for your continued support of game development education and for the opportunity to share my research at the PikPok Developers' Conference.

Finally, to Tanya Carter, thank you for your tireless efforts in supporting me to undertake post-graduate study while working full-time. Thanks for keeping me alive.

# Chapter 1

## Introduction

“No game is 100% bug free.”

---

John Hight and Jeannie Novak,  
Game Development Essentials:  
Game Project Management  
(Hight & Novak, 2008, p. 171)

### 1.1 Overview

The game development industry has been described as operating at the “bleeding edge of technical complexity” (Lewis & Whitehead, 2011b, p. 2). It is a global industry comprised of large, multi-billion dollar companies, as well as small, independent studios (Scacchi & Cooper, 2015). Modern video game development is creative, often team-based and multidisciplinary, where specialised game development roles collaboratively contribute to the creation of a video game (Kanode & Haddad, 2009). The development team is primarily composed of programmers, designers, artists and producers, but can include other supporting roles. These roles are dependent on each other to create the overall game experience (Murphy-Hill, Zimmermann, & Nagappan, 2014) and work in parallel during development (Kanode & Haddad, 2009). There are unique challenges in game development due to its multidisciplinary nature. For example, there can be division within a team which is often reduced to *programmers vs. artists* (Kanode & Haddad, 2009; Petrillo, Pimenta, Trindade, & Dietrich, 2008). When contrasted with general software, games have only one main requirement — that they must be *fun* (Petrillo et al., 2008). The concept of fun is subjective and is an artistic achievement (Murphy-Hill et al., 2014). Game play is about *feel* — developers aim to design an emotional experience that engages a player over time (Murphy-Hill et al., 2014).

The overall development process is known as *game production*, which is generally broken down into

*Alpha*, *Beta* and *Gold* stages, each with milestone goals which measure the output of the stage (Bethke, 2003). The target of the Alpha milestone is to have a game feature-complete, but not necessarily bug-free. At the Beta stage, all assets should be permanently in place, and all severe bugs should have been repaired. The Gold milestone is the finished game that is shipped to the customer (Levy & Novak, 2010). The production phase is where developers create the majority of the game assets and source code, but it can also include further pre-production-related activities, as well as testing activities (Kanode & Haddad, 2009).

An example of a large-scale, big-budget game — commonly known as a Triple-A (AAA) game — is the commercially released video game, *BioShock* (2K Boston and 2K Australia, 2007), which was developed over a three-year period across three continents. The development studio employed 93 in-house developers and 30 contractors. The code base comprised of 758903 lines of C++ code, and 187114 lines of script code. The game’s publisher provided eight on-site testers (Kanode & Haddad, 2009).

A single video game can target multiple hardware platforms, including home game consoles, mobile devices, such as smartphones and tablets, and personal computers (PC) and laptops (Peddie, 2013; Scacchi & Cooper, 2015). Platforms have a general purpose central processing unit (CPU), as well as a graphics processing unit (GPU) (Shirley & Marschner, 2009), and also include an operating system (OS), such as Microsoft Windows, Apple Mac OS X, or Linux. Shirley and Marschner (2009) define a platform as the combination of hardware, operating system and application programming interface (API). The computer graphics API is responsible for providing a set of rendering functionality (Shirley & Marschner, 2009) and communicates with the graphics device driver, which is software that exposes the hardware’s capabilities to the API (Peddie, 2013), as can be seen in Figure 1.1. *Cross-platform* software is executable on multiple operating systems (Thorn, 2008) and must behave the same way on different hardware (Goodwin, 2005). In game production terms, each target platform version of a game is known as a SKU (stock keeping unit) (Irish, 2005).

Current video game platforms range from home consoles, such as the Sony *PlayStation 4*, Microsoft *Xbox One*, and Nintendo *Wii U*, to mobile handheld devices such as the *Nintendo 3DS*, Sony *PlayStation Vita*, and modern smartphones and tablets, such as the Apple *iPhone* and *iPad* and the various Google Android-capable devices (B. Sutherland, 2013). The PC is also a major game platform.

Game development is often driven by the desire to ship a product in time for Christmas, as this is the annual period of highest customer demand (Murphy-Hill et al., 2014). The opportunity to have a commercially successful game relates to when the game is delivered to market and what competing game products may also be on the market at the same time (Petrillo et al., 2008). For example, if a game is released two months after a competitor’s, the likelihood of success is minimised, but if a

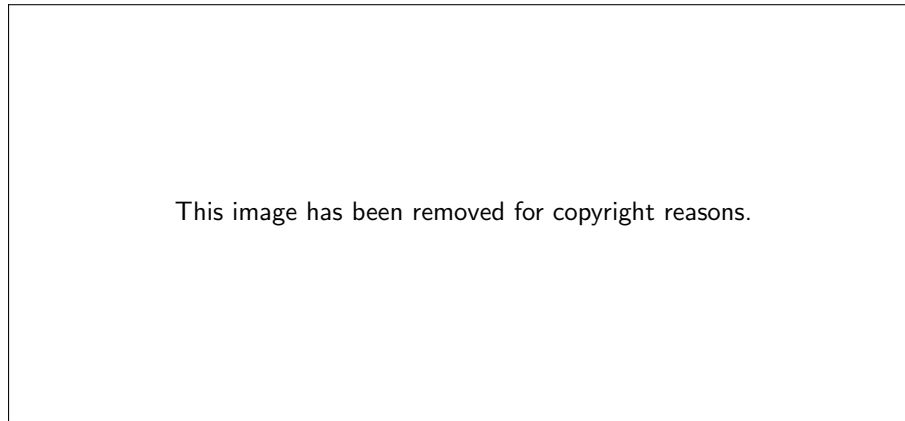


Figure 1.1: Target platform block diagram. Adapted from “The History of Visual Magic in Computers: How Beautiful Images are Made in CAD, 3D, VR, and AR” by J. Peddie, Copyright 2013.

game is released before a competitor’s and contains defects, there could also be a negative commercial impact (Petrillo et al., 2008). In order to ship games on time and meet their production schedule, game developers endure *crunch time* periods. A team could spend months in the lead up to a release working twelve-hour days, seven days a week (Petrillo et al., 2008) — a practice that is considered normal within the game development industry.

Advances in video game technology help to drive computer graphics hardware innovation, research and development (Lewis & Whitehead, 2011b; Petrillo et al., 2008). As hardware has increased in power, game development times and costs have also increased. The game development industry has to accommodate new technologies on a regular basis. When a new graphics card is released developers need to take advantage of the new hardware features, and when new platforms are released developers need to build their game for the new platform (Kanode & Haddad, 2009). There are risks involved with utilising new technology and an associated learning curve with the arrival of new hardware (Kanode & Haddad, 2009; Petrillo et al., 2008). Developers can be unaware of potential problems with new hardware and a new platform can contain problems that are only discovered by developers creating launch title game software (Petrillo et al., 2008).

Game developers are often required by publishers to target as many platforms as possible to increase potential profit (Fahy & Krewer, 2012; Reimer, 2005). Cross-platform games have a wider possible consumer base (Gregory, 2009; Irish, 2005; Phillips, 2002) and marketing effort and budgets can be reduced if companies can launch all target platforms at the same time (Demerjian, 2012). This can be achieved with only a small increase to the overall cost of developing a game (Shirley & Marschner, 2009). A game could be released on multiple platforms timed to coincide with the release of a major film (Irish, 2005), further increasing potential exposure and, therefore, profit. Cross-platform portability can also benefit developers, as they can make use of additional debugging strategies provided by

different platforms to potentially detect bugs earlier in development. It also gives developers the freedom to change platforms as technology changes (Hook, 2005). Reduced financial risk to game developers is seen as an advantage of releasing a game on multiple platforms (DeLoura, 2011).

The AAA games *Assassin's Creed IV: Black Flag* (Ubisoft Montreal, 2013), *Call of Duty: Ghosts* (Infinity Ward, 2013) and *Watch Dogs* (Ubisoft Montreal, 2014b) each released six SKUs, targeting *Xbox One*, *Xbox 360*, *PlayStation 4*, *PlayStation 3*, and *Wii U*, as well as Windows PC. *Lego Marvel Super Heroes* (Traveller's Tales, 2013) targeted seven SKUs — all of the above, plus Mac OS X. The game was also released with mobile SKUs, under the title *Lego Marvel Super Heroes: Universe in Peril* (TT Fusion, 2013), targeting five platforms — *PlayStation Vita*, *Nintendo 3DS*, *Nintendo DS*, iOS, and Android; a total of twelve SKUs overall. As well as being cross-platform, these games can also be considered cross-generational, since they targeted not only the latest iteration of console hardware, but also the previous generation.

To create video game software for multiple platforms, developers must take account of the variations between platforms. These differences can include the media the game is stored on, the physical hardware that executes the game software, the operating system, device driver capabilities, the style of input system and sound capabilities (B. Sutherland, 2013). There is also a variety of computer graphics systems available to game developers, with a range of hardware devices, GPUs, graphics drivers and APIs (Zioma & Pranckevicius, 2012). Specialised low-level systems must be developed for each platform and this hinders developers wishing to target multiple platforms (Fahy & Krewer, 2012).

Reliable computer graphics implementations across all target platforms are important to ensure customers experience a video game the same way on each platform. A game needs to look good to the consumer on low-end, as well as high-end hardware, and perform as expected (Hardwidge, 2009; Petrillo, Pimenta, Trindade, & Dietrich, 2009). Games can achieve fame or notoriety based upon the foundations of their software and consumer expectations have risen as video games have become more complex (Kanode & Haddad, 2009). A game must be able to run on different configurations of PCs and the number of PC hardware combinations is a hindrance to game developers testing multi-platform games (Hardwidge, 2009). Large game companies have testing centres that contain a wide variety of hardware combinations that can be used, allowing for combinations of CPU, RAM, graphics cards and drivers to be tested to ensure games are fault-free on a variety of hardware setups (Murphy-Hill et al., 2014).

During development, game developers use different *build configurations* to control the build process. Different configurations have different purposes for development. For example, the *Release* build configuration generates an executable intended to be the consumer version of a game; the *Debug* build configuration helps with the development and debugging of a game, but is not as efficient as the Release

build due to the inclusion of the additional debugging features; and, the *Profile* build configuration is used to measure the runtime performance of the game software (Gregory, 2009; McShaffry & Graham, 2013). Each build configuration results in a different executable and hence creates additional versions of the game that need testing for bugs (Rabin, 2010). To further complicate development, each target platform will have different build configurations. Even a game with only two target platforms, can have many build configurations to test, as seen with *Thief: Deadly Shadows* (Ion Storm Austin, 2004), a game built for Windows PC and *Xbox*. With the inclusion of an *Editor* build, and a Debug, Profile and Release configuration for this build, as well as the two target platforms, the game had a total of nine executable versions that potentially required testing (McShaffry & Graham, 2013). In the examples cited previously, where games were developed for up to 12 target platforms, there could have been up to 36 executable versions to test, with the potential for bugs to exist in any one version.

## 1.2 Research Motivation

Detailed knowledge of internal processes and development is not readily shared between rival game developers (Petrillo et al., 2008; Scacchi & Cooper, 2015). It is therefore difficult to pinpoint what valuable areas for research exist in games due to the *blackbox* nature of the industry and the prevalence of non-disclosure agreements (NDA), as well as the existence of proprietary source code (Lewis & Whitehead, 2011b).

Kanode and Haddad (2009) suggest that the majority of problems occur during the production phase. Problems are often caused when new functional requirements are added to the game during this phase, a phenomenon known as *feature creep*. Feature creep occurs due to the non-linear nature of the game development process and because of optimistic initial scoping. Early in the development cycle, game developers may spend considerable time writing code that is considered expendable, due to early creative and development efforts being focused on exploring various game design ideas, the subjective nature of the game’s requirements and trying to narrow in on the fun aspects of the game (Murphy-Hill et al., 2014). Because of the tension between creative aspirations and technical constraints, Murphy-Hill et al. (2014) suggest that the game development industry is resistant to formal processes that may restrict the creative process, and instead adopts a *cowboy coding* approach to development that rejects any formal methodological approach. Scope issues can cause games to exceed their proposed budgets and to contain defects, as developers try to balance cost and quality with releasing a game on time (Petrillo et al., 2008; Scacchi & Cooper, 2015). Post-mortem documents contain reflective summaries of the development process, and describe reasons for failed projects, such as feature set change or substantial feature reduction occurring during development (Petrillo et al., 2008, 2009).

Changes to the game design requirements can occur to allow the game to remain on budget and within the projected time-frame, for example, by removing or halting development for one or more target platforms early in the production process (Petrillo et al., 2009).

The early detection of game-breaking bugs is an area for research (Murphy-Hill et al., 2014). Scacchi and Cooper (2015) state there are many areas of software testing to explore that can support the development of video games, such as that formal software architectural design may help facilitate automated testing, while Murphy-Hill et al. (2014) found that there is significantly less automated testing conducted in game development when compared with general software development. Petrillo et al. (2008) support the need to develop specific techniques to improve verification of the requirements in game development.

### 1.2.1 Graphical Bugs

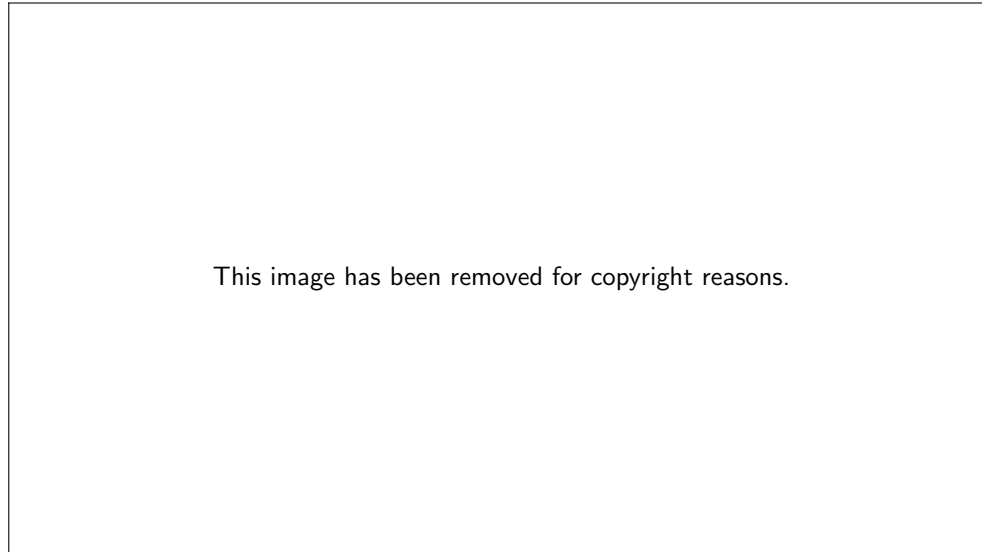


Figure 1.2: *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) screen capture featuring a graphical defect where the character's face is incorrectly rendered ([Assassin's Creed Unity image screen capture], n.d.).

Computer graphics rendering errors are often present in commercially released games. Recent AAA titles such as *Mafia II* (2K Czech, 2010), *Fallout: New Vegas* (Obsidian Entertainment, 2010), *Call of Duty: Modern Warfare 3* (Infinity Ward, 2011), *Assassin's Creed IV: Black Flag* (Ubisoft Montreal, 2013), and *Assassin's Creed Unity* (Ubisoft Montreal, 2014a), have been released containing a variety of graphics defects (Asif, 2013; Hartel, 2011; Iftikhar, 2010; Yezpitelok, Cantrell, & Rio, 2012). An example of a graphical defect, or bug, in the game *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) can be seen in Figure 1.2, where the in-game character's face is incorrectly rendered with potentially hilarious and terrifying results. Finding images like this in news articles and community game forums



led me to want to investigate further, as it is clear that these types of graphical bugs have an impact on the way consumers view particular games, franchises, publishers and developers. While initially garnering laughs from the gaming community, there is a more serious undertone to their criticisms, and these types of bugs do have a negative impact on game play, reviews, sales, and therefore stock values for the companies involved in releasing these games to market (Bell, 2013; Thier, 2014). For example, Ubisoft's share prices dropped 9.12% after the launch of *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) (Cullinane, 2014) and *Battlefield 4* (EA Digital Illusions CE, 2013) launch bugs led to the filing of a class action suit against Electronic Arts (Bell, 2013).

Defects in game software can arise from traditional computer programming mistakes, such as flaws in source code (Kanode & Haddad, 2009). Defects can also arise from changes in game content and related assets (Murphy-Hill et al., 2014), and the difference between development and consumer hardware can lead to bugs appearing on one hardware instance of a device and not another (Rabin, 2010). A potential cause of graphical bugs is the graphics device driver, which can be unreliable (Abrash, 1997). For example, game developers attributed graphical issues present in *Rage* (id Software, 2011) to specific graphics cards drivers (Sharke, 2011).

On one hand, it is understandable that games could be released with these types of graphical bugs present, since finding errors in the graphical user interface (GUI) of a software application is difficult because it is labour intensive (Myers, Badgett, & Sandler, 2012), and computer graphics systems rely on humans to interpret graphical results (Hunt & Thomas, 2000). The finding and repairing of defects is considered to be the most time consuming and expensive aspect of software development (Petrillo et al., 2009). Testing is difficult due to the unpredictability of non-deterministic aspects of games, such as randomness, artificial intelligence, and multi-threaded algorithms (Bourg & Seemann, 2004; Hunt & Thomas, 2000; Murphy-Hill et al., 2014). Currently, it is common in the game development industry to utilise manual testing techniques, hiring up to hundreds of human testers to play builds of the game to inspect for defects (Kanode & Haddad, 2009; Lewis & Whitehead, 2011b; Murphy-Hill et al., 2014). This includes testing the game under usual playing conditions and also stress testing it to its limits (Kanode & Haddad, 2009). It is difficult for a tester to explore an entire game world, or to sometimes even identify what is considered to be correct behaviour (Hunt & Thomas, 2000; Murphy-Hill et al., 2014). A project's bug count can reduce slowly since fixing one bug can cause other defects to occur (Petrillo et al., 2009). The developers of the game *Tom Clancy's Rainbow Six* (Red Storm Entertainment, 1998) were still finding new crash bugs the week before its Gold milestone and in the development of *Draconus: Cult of the Wyrms* (Treyarch, 2000) over 3000 bugs were identified. *Black & White* (Lionhead Studios, 2001) also contained more than 3000 bugs that had to be fixed within a six-week timeframe (Petrillo et al., 2009). Further background on game industry testing practices can

be found in Appendix A.

Patton (2001) describes speed, efficiency, accuracy, precision and relentlessness as benefits of automated testing tools and David Price, Test Lead at THQ, Inc., stated in an interview with Levy and Novak (2010) that automated testing will become essential in verifying the functionality and visual appearance of games. However, though Carucci (2009), Technical Director at game developer Crytek, also suggested that future development of higher-level functional testing is necessary, he highlighted an observed resistance by the game industry to the adoption of automated testing.

With these things in mind, it seemed clear to me that further investigation into testing was warranted, and that there might be a way to create an automated testing system with a focus specifically on detecting graphical errors before the commercial release of a game. In this research project, I investigate and create a system for the automation of testing and validation of cross-platform computer graphics systems for video game development. In conducting research, the proposed research problem must be worthy of investigation (Zobel, 2004). The resulting knowledge from my research could change industry practice, encouraging automated testing of computer graphics systems to improve quality assurance (QA) when developing game software simultaneously across multiple platforms. This system could be utilised by independent development studios, as well as AAA developers. This will help to create robust cross-platform game software, potentially reduce development costs, maximise cross-platform market opportunity, increase the likelihood of a game being delivered on time, and improve the game players' experience.

## 1.3 Research Questions

In this research project, I ask *can testing and validation of computer graphics implementations for cross-platform game development be automated?*

To answer the core research question, I have broken it down into the following sub-questions:

- What technological developments have led to the current state of video games?
- What examples of graphical defects exist in commercial video game software?
- What testing practices are currently used in the development of video games?
- What technology and techniques are used to develop cross-platform video games?
- How can manual quality assurance testing for visual defects in cross-platform video games be automated?

Section 1.5 addresses the technological developments that have led to the current state of video games and related computer graphics. Chapter 2 provides insight into game development processes,

related academic works, and game industry testing practices. Chapter 3 addresses the research methods employed. Technology and techniques for development of cross-platform video games are addressed and the automation of quality assurance for visual defects in cross-platform video games is realised through the creation and evaluation of artefacts in Chapter 4 and Chapter 5. Finally, Chapter 6 summarises concluding thoughts and proposed future works.

## 1.4 Research Method

To investigate this research question, I use Design Science Research Methods and Patterns as described by Vaishnavi and Kuechler (2008). This provides a framework for conducting research through iterative design, creation and evaluation of artefacts — “exploring by building” (Vaishnavi & Kuechler, 2008, p.2) — which further inform the nature of the problem under investigation. *Problem Selection Patterns*, such as *Research Domain Identification*, emphasise personal interest as a core reason for the selection of a research domain. Additionally, the pattern *Leveraging Expertise*, states that research should be based on the current strengths of the researcher, giving the project the best chance of success. My personal and professional interests are in the area of video game development, particularly cross-platform development techniques, and the construction of high-quality video game software and tools, and my research question has grown out of these interests, as well as my strengths and experience in this domain.

## 1.5 A Brief History of Computer Graphics and Video Game Technology

This section provides a selected overview of the background and historical elements leading to the development of the modern computer graphics systems that are used in video game development today. Modern computer graphics has a variety of applications beyond video games, such as information visualisation, medical imaging, computer-aided design (CAD), simulation, visual effects, and animated films and television (Peddie, 2013; Shirley & Marschner, 2009). The history and development is multifaceted, happening across a combination of hardware manufacturers, software developers and end-user market demands (Peddie, 2013). Historical innovation occurred in a competitive environment and a significant number of early computer graphics manufacturers and developers no longer exist.

At a high level, video game consoles and modern mobile game devices share internal similarities, such as a CPU and GPU. But they each have differing display capabilities, input systems, and sound capabilities (B. Sutherland, 2013). On a modern target platform, a video game’s computational wor-

kload is shared between the CPU and the GPU (Akenine-Moller, Haines, & Hoffman, 2008; Peddie, 2013) and the work done by both processors, in combination, results in games featuring 3D character animation, artificial intelligence, and dynamic physics simulations, including projectiles and particle effects, among other features (Peddie, 2013). Computer graphics can be compared to a computational “black hole” (Peddie, 2013, p. 122) because its complexity is such that no matter how much processing power is made available, programmers will continue to utilise all of it.

Video game console hardware and graphical abilities are fixed by design, ensuring that performance is consistent across devices of the same type. Exact specifications of game console hardware are often secret, and developers are only given access to the behind-the-scenes knowledge once they sign a non-disclosure agreement which prohibits them from publicly discussing technology used in development (Irish, 2005). Home consoles prior to the mid-1990s mainly featured 2D graphics capabilities, evolving to support 3D graphics during the mid-1990s and beyond. In contrast to home consoles, there is a large variety of consumer hardware available for PCs, such as dedicated graphics cards, which can be configured in a number of ways. This creates the potential for varied graphical performance across the PC platform (Shirley & Marschner, 2009). However, differences can exist within a specific console type also, for example, where the development hardware utilised by developers is preliminary and continues to develop up until the final consumer release of the console. This can lead to the appearance of bugs on one hardware instance of a device, but not another (Rabin, 2010).

### 1.5.1 Computer Graphics APIs

A variety of graphics libraries and standards exist (Fahy & Krewer, 2012). Graphics libraries for programming languages, also known as computer graphics APIs (CG API), allow developers to author video game software using languages such as C++, and hence create graphical output on the hardware through calls to the CG API. Over time, APIs are updated to add functionality for features supported by newer hardware (Zerbst & Duvel, 2004). In the past, software standards were utilised in computer graphics with the aim of aiding the creation of graphical software that was portable, allowing graphical software to be utilised with a variety of hardware without the need to completely rewrite programs (Hearn & Baker, 2003). Historical graphical programming libraries include Graphics Library (GL) and Virtual Reality Modeling Language (VRML).

Today, the main graphics hardware manufacturers are Nvidia, AMD and Intel. In the late-1990s and early-2000s, ATI was the number two supplier of graphics chips behind Nvidia. It had acquired Tseng Labs in 1997 and ArtX in 2000, the company responsible for the graphics chip used in the Nintendo *GameCube* home console. In 2006, AMD purchased ATI, expanding from CPU manufacture and development into GPU manufacture, and becoming a major innovator in 3D graphics for games

(Peddie, 2013). This cannibalisation of companies is illustrative of the competitiveness in the hardware market. Many of AMD’s innovations have been integrated into the DirectX API (Peddie, 2013), which along with OpenGL is one of the main APIs currently used in computer graphics (Sherrod, 2008).

Interactive computer graphics was pioneered by Ivan Sutherland with his 1963 seminal doctoral work (Foley, van Dam, Feiner, & Hughes, 1997), *Sketchpad* (I. E. Sutherland, 1963). Early evolution throughout the 1970s resulted in Plot10 and CORE, before the emergence of Graphical Kernel System (GKS) in 1984, the first ISO standard for low-level computer graphics (Peddie, 2013). GKS was initially a 2D graphics software standard, but was enhanced to become a 3D-capable standard. Sutherland’s work influenced the development of Programmer’s Hierarchical Interactive Graphics Standard (PHIGS) (Foley et al., 1997), also an extension of GKS, which introduced hierarchical object modelling, and was extended upon to create the 3D-capable PHIGS+ (Carson, van Dam, Puk, & Henderson, 1998).

Computer graphics further evolved with Silicon Graphics (SGI), who developed Integrated Raster Imaging System Graphics Library (IRIS GL) in 1982 (Peddie, 2013). IRIS GL was a proprietary API for the company’s graphical workstations, however in 1991, SGI started to license IRIS GL, which made an industry-wide 3D graphics standard possible. Internally, SGI began development on what would become OpenGL. After removing their proprietary code, SGI teamed up with Microsoft to further develop OpenGL, releasing version 1.0 in mid-1992 through the Architecture Review Board (ARB), an organisation comprised of the industry leaders at the time (Hearn & Baker, 2003; Peddie, 2013; Segal & Akeley, 1994).

During the 1990s, computer graphics processor manufacturers, such as SGI and Nvidia, looked to the console and arcade-gaming manufacturers to offset the development costs of new graphics chipsets (Peddie, 2013). Prior to the mid-1990s, graphics processor manufacturers had their own APIs. In 1995, Microsoft released version 1.0 of DirectX, which became the standard for graphics development in the Windows OS environment (Peddie, 2013). Following the release of DirectX 1.0, Microsoft purchased RenderMorphics, which allowed them to implement a 3D graphics engine for Windows 95. DirectX3D shipped with DirectX 2.0 and DirectX 3.0 (Peddie, 2013). This was the beginning of a trend where new versions of DirectX are generally released with new versions of the Windows OS. In the mid-1990s, Apple also developed their own API, QuickDraw 3D (QD3D) for Macintosh computers. However, by 1999 Apple announced that OpenGL would be used by Mac OS X instead of their proprietary API (Peddie, 2013). Also in the mid-1990s, 3dfx Interactive developed the Glide API for the PC. Other proprietary APIs of the time include: Hierarchical Object-Oriented Programming System (HOOPS), VESA Advanced Graphics Interface (VAGI), 3DRender (3DR), WinG and Java 3D (Peddie, 2013). The number of graphics processor manufacturers began to decline throughout the mid-1990s and into the 2000s, however consumer demand continued to increase (Peddie, 2013).

At this time, DirectX and OpenGL both used the fixed-function graphics pipeline (FFP). The FFP allowed programmers to configure the rendering pipeline by setting their desired render states, but the stages in the graphics pipeline were not flexible (Akenine-Moller et al., 2008; Sherrod, 2008). In 1999, the DirectX 7.0 API was released, which added hardware transformation and lighting support via GPU hardware acceleration. The mobile console, Sony *PlayStation Portable* — released in 2004 — had a fixed-function graphics pipeline, and a proprietary low-level graphics API, known as Geman (Schertenleib & Hirani, 2009). The most recent home game console to utilise the FFP was the Nintendo *Wii*, which contained an ATI GPU and was released in 2006 (Akenine-Moller et al., 2008; McShaffry & Graham, 2013).

As GPUs continued to develop, OpenGL and DirectX implemented support for *shaders*. These are small programs that control the programmable graphics pipeline. A variety of shader programming languages exist, and these have evolved as graphics hardware has advanced. In contrast with the FFP, programmable shaders allow the programmer to choose exactly what operations are conducted at each stage of the rendering pipeline, by implementing their own algorithms as shader program source code. As such, programmers have greater flexibility to utilise the underlying graphics hardware with a programmable graphics pipeline than when using the FFP (Akenine-Moller et al., 2008; Sherrod, 2008). Early shader languages include the low-level ARB assembly language and Nvidia’s high-level Cg (C for Graphics), supporting vertex and pixel shaders. DirectX 8.0 introduced programmable vertex and pixel shaders using assembly code (Peddie, 2013). Microsoft released their first console in 2001 — the *Xbox* — which had a graphics API similar to DirectX 8.0.

DirectX 9.0 featured fixed-function and programmable shaders, and throughout its releases introduced Shader Models 1.0, 2.0 and 3.0, adding support for High-level Shader Language (HLSL). The OpenGL equivalent, OpenGL 2.0, also added high-level shader support through the OpenGL Shading Language (GLSL). Microsoft’s *Xbox 360*, first released in 2005, contained the PowerPC Xenon CPU, with three cores, and an ATI GPU (McShaffry & Graham, 2013) with a variant of the DirectX 9.0 API (Sherrod & Jones, 2012), and had a unified shader architecture (where all shaders have the same rendering capabilities) supporting Shader Model 3.0 (Akenine-Moller et al., 2008). The Sony *PlayStation 3*, released in 2006, utilised the Cell Broadband Engine CPU and an Nvidia GPU, known as the RSX (Akenine-Moller et al., 2008; McShaffry & Graham, 2013), which had a full shader pipeline, and a proprietary low-level graphics API, known as Libgcm (Gregory, 2009; Schertenleib & Hirani, 2009).

The shader pipeline continued to expand to include geometry and tessellation shaders. For Windows PC, DirectX 10.0 introduced Shader Model 4.0, featuring unified shaders (Peddie, 2013), and fully removed the FFP, only supporting programmable shaders (Sherrod & Jones, 2012). DirectX 11.0 saw the introduction of Shader Model 5.0 (Sherrod & Jones, 2012) and was backwards compatible,

working with DirectX 11.0- and DirectX 10.0-class hardware. OpenGL 4.0 was equivalent feature-wise to DirectX 11.0 (Sherrod & Jones, 2012).

Meanwhile, OpenGL ES (Embedded Systems) was developed for mobile devices. This started with FFP support in version 1.1, while version 2.0 included support for programmable shaders. Due to the large variety of mobile devices in the market, with varying quality of hardware, both FFP and programmable shader versions of OpenGL ES are used by developers (B. Sutherland, 2013).

Modern OpenGL on Windows OS, Mac OS X, Linux, and mobile devices such as *iPad* and *iPhone*, allows native device creation which differs from platform to platform, however the API is considered platform independent (Sherrod & Jones, 2012). OpenGL provides C++ API interface bindings (Hearn & Baker, 2003).

New graphics libraries and APIs continue to be developed, such as Almost Native Graphics Layer Engine (ANGLE), which converts OpenGL ES API calls into DirectX API calls on Windows PCs (Fahy & Krewer, 2012). In the mid-2010s, CG APIs have also evolved, with a recent focus on new low-level graphics APIs, which target enhanced performance by reducing architectural overheads. The Sony *PlayStation 4* has a proprietary low-level graphics API, known as *GNM*, which is similar to DirectX and OpenGL, and utilises the platform-specific shader language *PlayStation Shader Language* (PSSL) (Schertenleib, 2013). Apple have also developed the low-level *Metal* CG API for their iOS platform and Microsoft have developed DirectX 12.0, which was released for the Windows 10 OS in 2015 and is in development for the *Xbox One* home game console. The Khronos Group released the low-level *Vulkan* CG API in 2016.

### 1.5.2 Middleware and Game Engines

As the complexity of developing video games increased, the need for source code reuse also increased. Third-party libraries, also known as *middleware*, have distinct purposes — such as rendering, physics simulation, artificial intelligence, and sound — and provide flexibility and a platform independence layer (Gregory, 2009; Kanode & Haddad, 2009). Middleware can be utilised to avoid redeveloping aspects of video game software which have previously been developed, thus potentially decreasing development time and increasing software reliability (Fahy & Krewer, 2012; Kanode & Haddad, 2009). Third-party solutions can also be used for faster prototype creation (Kanode & Haddad, 2009). An example of middleware is the graphical library Simple DirectMedia Layer (SDL), which supports desktop operating systems such as Windows OS, Mac OS X, and Linux, as well as the mobile platforms iOS and Android (Fahy & Krewer, 2012).

Games are composed of a significant amount of multimedia assets and resources that make up the virtual environment (Kanode & Haddad, 2009; Murphy-Hill et al., 2014). The asset pipeline

converts and prepares the multimedia assets from their authoring format into forms that are able to be utilised in real-time simulations on the appropriate target platforms (Murphy-Hill et al., 2014). It is an essential area of game production, and the creation of such a pipeline can be a major software development project on its own (Kanode & Haddad, 2009; Murphy-Hill et al., 2014).

Instead of writing the source code for each game from scratch, reusable *game engines* have emerged, allowing developers to build their video game on top of an existing abstraction (B. Sutherland, 2013). When using a reusable game engine, the game’s source code is written once and recompiled for each target platform (Gregory, 2009; Harbour, 2004). Using a game engine can reduce the financial cost of developing a video game (B. Sutherland, 2013). Console video game developers utilise game engines, and mobile platforms are also moving in this direction (B. Sutherland, 2013). A game engine could be responsible for efficient graphics rendering, obtaining user input, providing network connection and playing sound, as well as managing game data (Zerbst & Duvel, 2004). Game engines support different target platforms requiring different graphics renderers. For example, the game engine utilised in the development of *The Sims Medieval* (The Sims Studio, 2011) supported the DirectX API for Windows PC and the OpenGL API for Mac OS X (McShaffry & Graham, 2013).

Game development companies create proprietary game engines and sometimes license their engine to other game developers (Gregory, 2009; B. Sutherland, 2013). Some examples of game engines are:

- *CryENGINE* (Crytek GmbH, n.d.): a proprietary cross-platform game engine. The third iteration of the engine is compatible with Sony *PlayStation 3*, Microsoft *Xbox 360*, and PC. The engine supports cross-platform development of 2D, 3D and stereoscopic games and features high-end rendering including real-time editing of multi-platform game environments (Crytek, n.d.).
- *PhyreEngine* (Sony Computer Entertainment, n.d.): a proprietary game engine developed by Sony and available to registered developers. PhyreEngine supports the Sony first-party target platforms *PlayStation 4*, *PlayStation 3*, and *PlayStation Vita*, as well as Windows OS, with rendering using OpenGL and DirectX 11.0. The developer’s interface to the engine is the same for all target platforms, making it possible for a developer to target their game for all supported platforms (Schertenleib, 2013).
- *Unity* (Unity Technologies, n.d.): a closed-source game engine commercially available as middleware to third-party developers. The engine is capable of supporting 2D and 3D game development and deployment to multiple platforms. The Unity game engine can deploy to twelve target platforms (Alistar, 2014), and is widely used for Android and iOS game development (B. Sutherland, 2013). Unity uses Cg as the main cross platform graphics shader language, which is then cross-compiled to generate HLSL, GLSL or GLSL ES shader code depending on the target platform



(Zioma & Pranckevicius, 2012).

- *Unreal Engine* (Epic Games, n.d.): supports the development of a variety of game types, including 2D and mobile games, as well as high-end AAA products (Tavakkoli, 2015). Unreal Engine 4’s entire C++ engine source code is available to developers, and game code written using the engine is compiled using Microsoft Visual Studio 2013 (Tavakkoli, 2015). In addition to this, the engine also provides a proprietary visual scripting language, known as *Blueprints*, in which developers can program games for Unreal Engine.

Other game engines include: Ubisoft Montreal’s *AnvilNext*, EA Digital Illusions CE’s *Frostbite*, id Software’s *id Tech 5*, and Ubisoft’s *Disrupt Engine*.

While middleware can help facilitate the development of a game, it can also have disadvantages. Technical solutions can impose design constraints and this can lead to the pre-determining of functionality, for example, the types of games developed are influenced and limited by the type of game engine or middleware utilised in development (Scacchi & Cooper, 2015). Third-party APIs for a platform or target hardware could fail to meet requirements, or become unsuitable during development, and may need to be replaced with internal studio-developed solutions (Petrillo et al., 2009).

## 1.6 Summary

In summary, publishers desire the creation of cross-platform games to increase profitability due to a wider target audience, and hence developers build them. There are advantages to developers beyond the financial rewards, as cross-platform development can reduce the costs of development, as well as the time taken to develop and market games, and help create more robust game software due to the availability of platform-specific development tools. However, building across platforms enhances complexity due to the differing underlying hardware and software technologies within each target platform. Additionally, technology is always evolving, and computer graphics technology changes quickly, creating further challenges for game developers. Graphical bugs exist for a number of reasons, and one of the consequences of developing games within this complex environment is that games are released containing graphical bugs that can affect the gameplay, which in turn affects the user experience and can damage the reputations of the companies involved. The aim of this research project is to create a tool that will aid in the automated detection of graphical defects in cross-platform video games, using the Design Science Research methodology to create and evaluate artefacts to answer the research question.

## Chapter 2

# Literature Review

“Game testing is not something  
that can be automated. Period.”

---

Jerome Strach, QA Manager,  
Sony Computer Entertainment  
America; Game Development  
Essentials: Game QA & Testing  
(Levy & Novak, 2010, p.205)

### 2.1 Structure of the Literature Review

Chapter 1 introduced the scale and complexity of the game development industry today, highlighting the rapid evolution of computer graphics technologies and providing an overview of cross-platform game development and its advantages and associated risks to developers, publishers and consumers. Additionally, the existence of graphical bugs in commercially released video game software was revealed as the primary motivation for this research project, given the significant impact they have on the game development industry and its stakeholders. My research question focuses on whether testing for these graphical defects can be automated.

In this literature review, Section 2.2 will use a taxonomy to define and describe the graphical bugs detected in a number of commercially released games, along with pictorial examples to illustrate the problem. Section 2.3 will discuss the use of automated testing in the game industry and present two industry examples alongside research into the current state of automated testing for game development. Section 2.4 will look into research in related fields, such as 3D, virtual reality, film and television, focused on automated testing of computer graphics systems. Finally, Section 2.5 will review the research presented and summarise common themes.

## 2.2 Video Game Graphical Defects

There are a variety of commercially released games which contain graphical defects. These have been reported by the game consumer community in game industry news articles and video game company forums. Players can detect the defects, and often developers must then respond and fix them after the initial release of a game. This can occur in the form of post-release updating (or *patching*) (Levy & Novak, 2010), and also through the developer providing additional instructions for players to follow that can potentially fix issues. Game developer The Creative Assembly, released instructions to customers to fix graphics bugs which were present in *Total War: Rome II* (The Creative Assembly, 2013) at the product’s launch (Younger, 2013) and Ubisoft published a known issues list for *Assassin’s Creed Unity* (Ubisoft Montreal, 2014a) to their online forums which outlined issues on target platforms and included progress updates for the resolution of various issues (Xhane, 2014). Patches released after a game has shipped require testing, and this post-release testing adds to the overall cost of the game’s development (Schultz, Bryant, & Langdell, 2005).

Video game testers classify bugs by severity, which relates to priority for repair, and category, which broadly describes the type of defect (Hight & Novak, 2008; Levy & Novak, 2010). *Class A* bugs have the highest priority because they impact a player’s ability to complete a game. Games cannot ship if they contain Class A bugs. *Class B* are high priority bugs that affect a player’s experience of a game, and should be resolved prior to release. *Class C* bugs are considered medium priority, and generally cause aesthetic or balance issues in a game, however, no single Class C bug will prevent commercial release of a game (Hight & Novak, 2008). One of the categories is *visual bugs* — these are bugs that affect a video game’s graphical output (Levy & Novak, 2010). The impact of these bugs can be minor, where the player may not notice the defect, or significant, where the bug hinders the player’s ability to play the game as the game designer intended (Levy & Novak, 2010). While some bugs might be obvious errors, such as the graphical defect in Figure 1.2, it can be difficult to ascertain if a game’s behaviour is a desired outcome without having access to a studio’s internal game design documentation (Lewis, Whitehead, & Wardrip-Fruin, 2010). Correct classification of bugs is important to avoid causing extra effort in locating and fixing game defects.

### 2.2.1 Examples of Graphical Defects in Commercial Games

Lewis et al. (2010) developed a taxonomy of possible failures that can occur in video games to help guide game designers and enable human testers to be more systematic when trying to discover defects. Failures are divided into two categories. *Temporal failures* require knowledge of a previous game state to enable the identification of game defects, while *non-temporal failures* can be identified at any

moment, with no knowledge of previous game state required. The taxonomy could be applied to an entire game, or to individual game objects, and could be applied to the game production process by inserting it into the tools used for game development, allowing for the detection and repair of bugs (Lewis et al., 2010). This would lead to better testing and improved design of video games, ensuring that they feature fewer defects when released.

The categories of the Lewis et al. (2010) taxonomy that are significant to this research project are: *Invalid Graphical Representation*; *Invalid Information Access*; *Lack of Required Information*; and, *Implementation Response Issues*. These are broad categories that contain a number of sub-categories of bugs.

I conducted an internet-based search to find examples of graphical bugs in commercially released cross-platform games and have used the Lewis et al. (2010) taxonomy to group these bugs. Each category is defined and described alongside pictorial examples that help to illustrate what players have seen when they have encountered these types of defects.

## **Invalid Graphical Representation**

*Invalid Graphical Representation* refers to certain aspects of the game world being rendered incorrectly. Examples of this include:

- *Clipping*: This occurs when 3D polygons intersect one another, for example, parts of in-game characters may be able to penetrate the virtual environment that the player would assume to be solid (Levy & Novak, 2010). *NBA 2K13* (Visual Concepts, 2012) featured this type of defect, as seen in Figure 2.1.
- *Z-fighting*: This occurs when two polygons of different depths within a 3D scene are rendered incorrectly due to both being computed as having the same z-buffer value. This can be caused by a lack of precision in the depth buffer for the desired 3D scene. As a result, the program is unable to resolve which polygon should be rendered in front (Levy & Novak, 2010). This graphical defect is examined by Lapidous and Jiao (1999), with an example of a 3D virtual character whose clothing polygons interpenetrate the polygons of the character’s skin. In *Garry’s Mod* (Facepunch Studios, 2004), this type of defect can be found in the game environment, as seen in Figure 2.2.
- *Visible artifact*: These bugs occur when elements that should not be in the scene are rendered (Levy & Novak, 2010). This can be seen in *Battlefield 4* (EA Digital Illusions CE, 2013) which launched with graphical problems that hindered game play. Red spot artifacts were rendered in the game scene which the player could mistake for a sniper rifle sight, as seen in Figure 2.3 (Staggs,

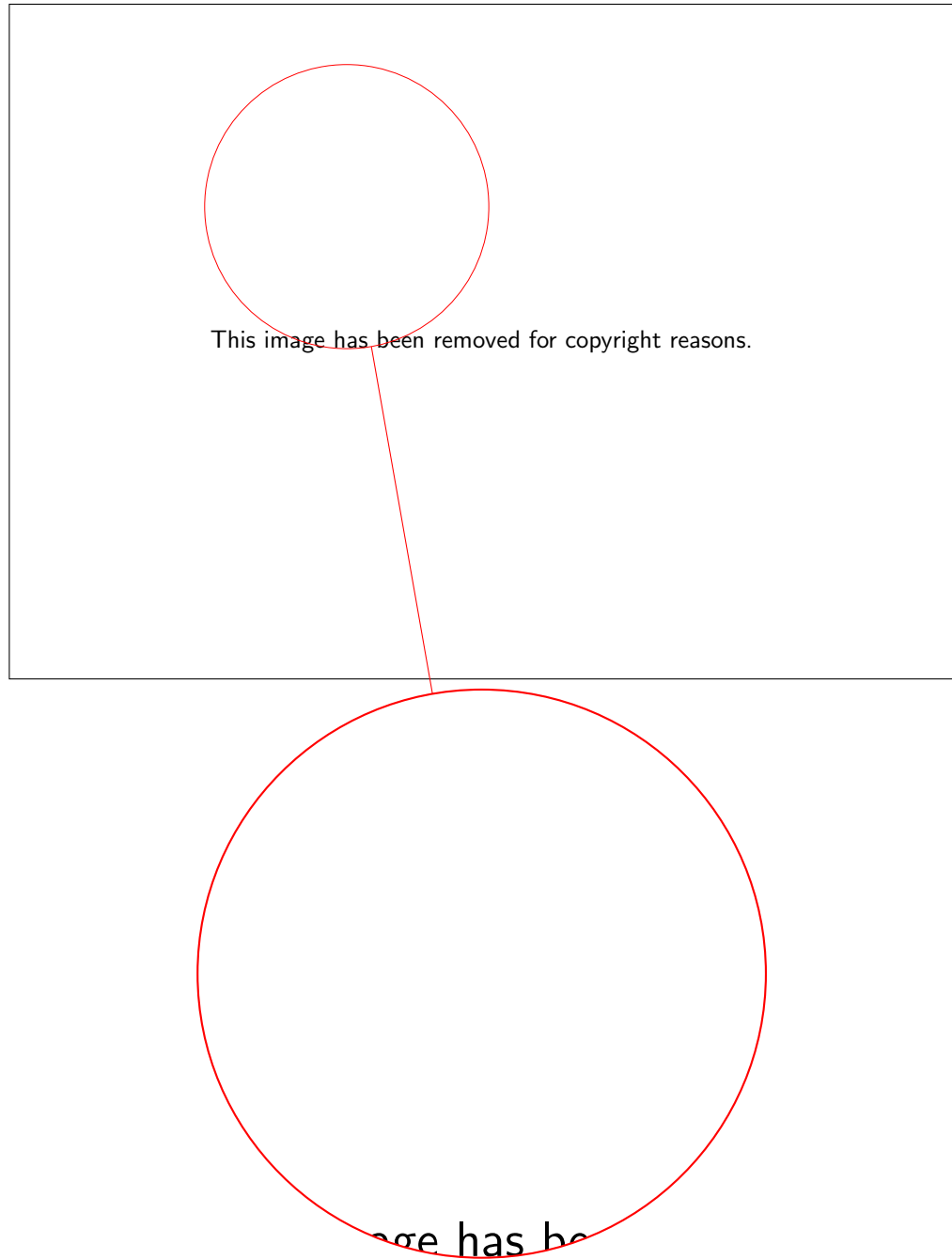


Figure 2.1: *NBA 2K13* (Visual Concepts, 2012) screen capture featuring a clipping issue where one character's forearm penetrates the face of another character ([NBA 2K13 image screen capture], n.d.).

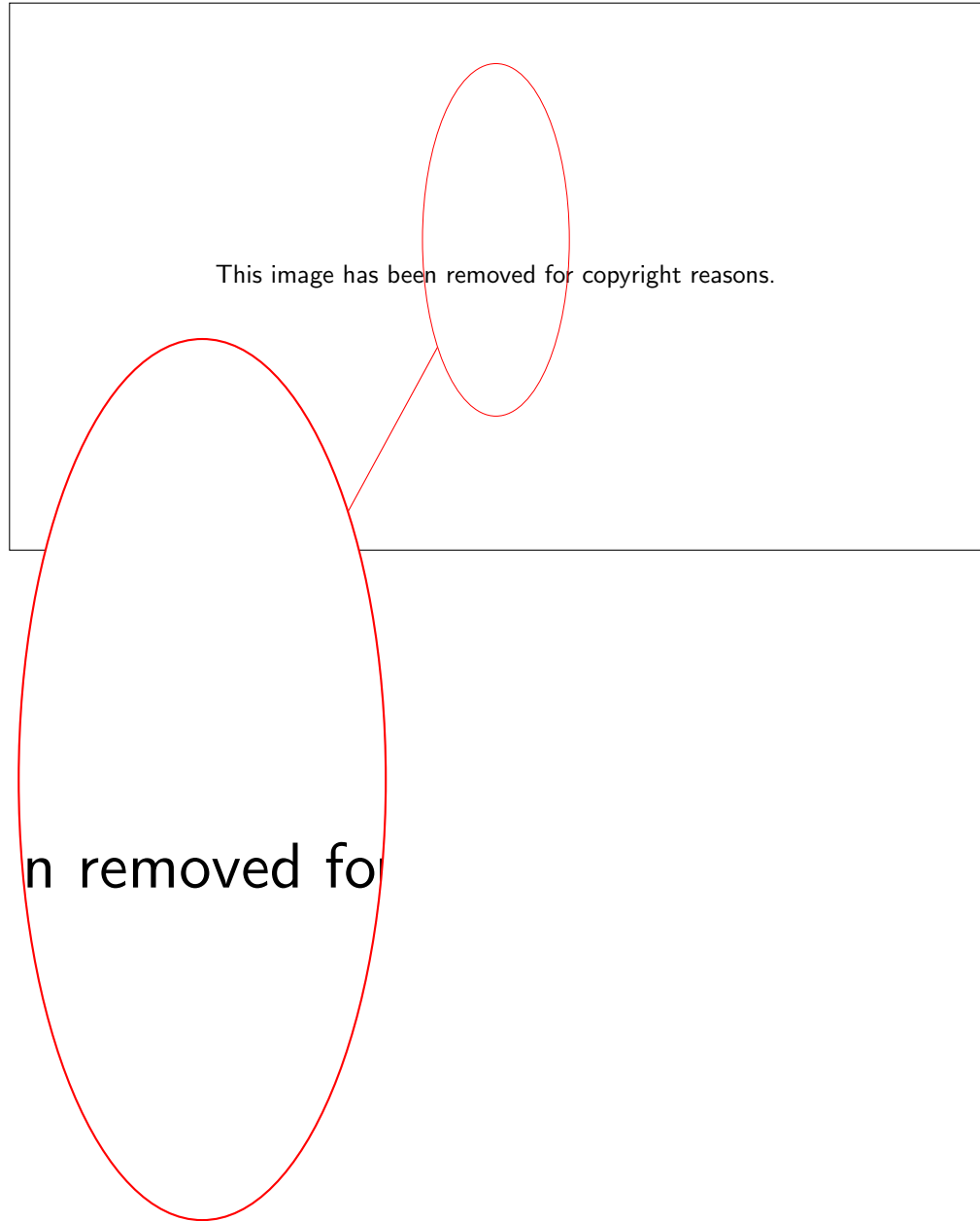


Figure 2.2: *Garry's Mod* (Facepunch Studios, 2004) screen capture of a video demonstrating a z-fighting issue, where the wall in the centre of the frame flickers from the dark grey texture to a different lighter grey texture as the player moves (Exho, 2014).

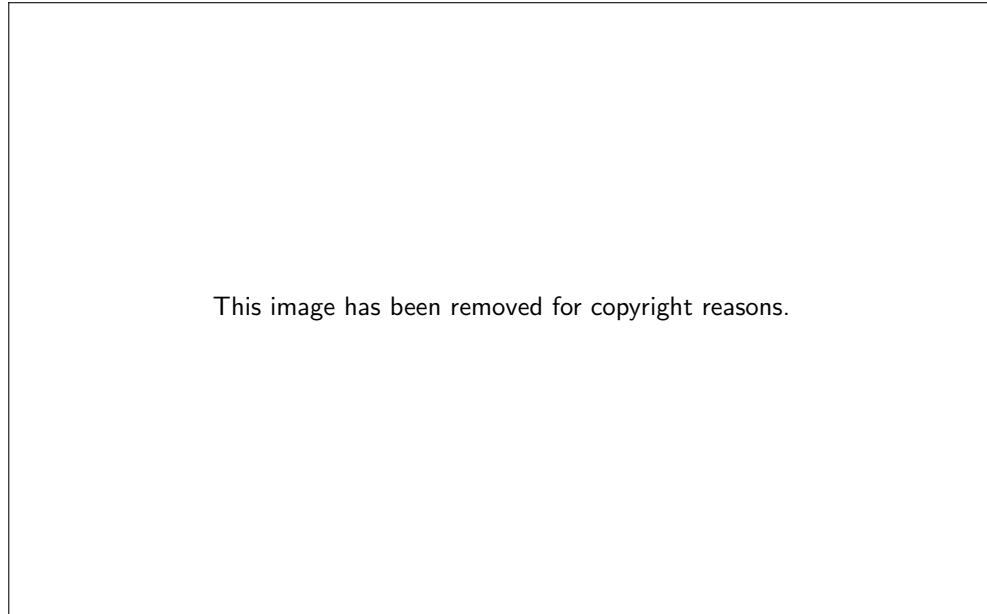


Figure 2.3: *Battlefield 4* (EA Digital Illusions CE, 2013) screen capture featuring a rendering error where red spots are rendered ([Battlefield 4 image screen capture], n.d.).

2013). Another example of a visible artifact bug occurred in the game *Batman: Arkham Knight* (Rocksteady Studios, 2015), as seen in Figure 2.4, and *Mafia II* (2K Czech, 2010) contained graphical artifacts which caused black lines to be rendered across game scenes (Iftikhar, 2010).

- *Shadows: Mafia II* (2K Czech, 2010) contained shadow rendering problems, as shown in Figure 2.5 (Iftikhar, 2010), and *Fallout: New Vegas* (Obsidian Entertainment, 2010) also featured shadow rendering issues (Hartel, 2011).
- *Missing head: Grand Theft Auto: San Andreas* (Rockstar North, 2004) featured rendering errors where a character's head was not rendered, as seen in Figure 2.6 (Yezpitelok & Cantrell, 2011). Also, *Call of Duty: Modern Warfare 3* (Infinity Ward, 2011) featured rendering problems where the playable game character's head was not displayed, as seen in Figure 2.7 (Yezpitelok & Cantrell, 2011).
- *Facial defects*: This type of defect can be seen in the *PlayStation 2* version of *Rocky* (Rage Software, 2002) which exhibited problems rendering a character's facial features, shown in Figure 2.8a. *Assassin's Creed Unity* also featured significant facial defects, as seen in Figure 1.2.
- *Character animation hierarchy defects*: The *PlayStation 2* version of *Rocky* (Rage Software, 2002) exhibited graphical problems, with limbs becoming detached from a character's body, as shown in Figure 2.8b (Yezpitelok & Cantrell, 2011), and *Battlefield 3* (EA Digital Illusions CE, 2011) featured characters with elongated necks, as seen in Figure 2.9. *Fallout: New Vegas*

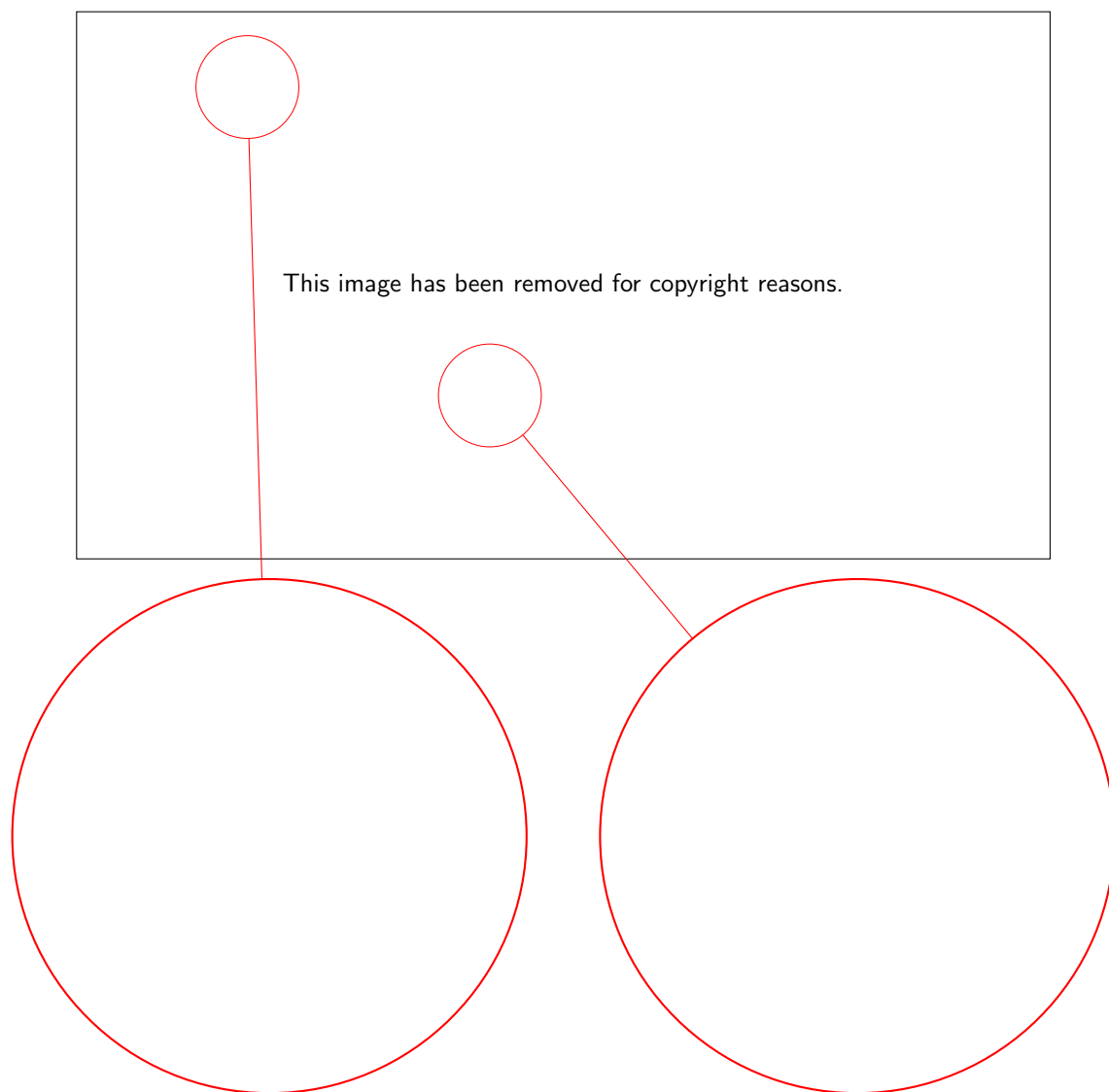


Figure 2.4: *Batman: Arkham Knight* (Rocksteady Studios, 2015) screen capture featuring a rendering error where the in-game character has spherical items rendered around its eyes, as well as artifacts across the lower third of the frame ([Batman: Arkham Knight image screen capture], n.d.).



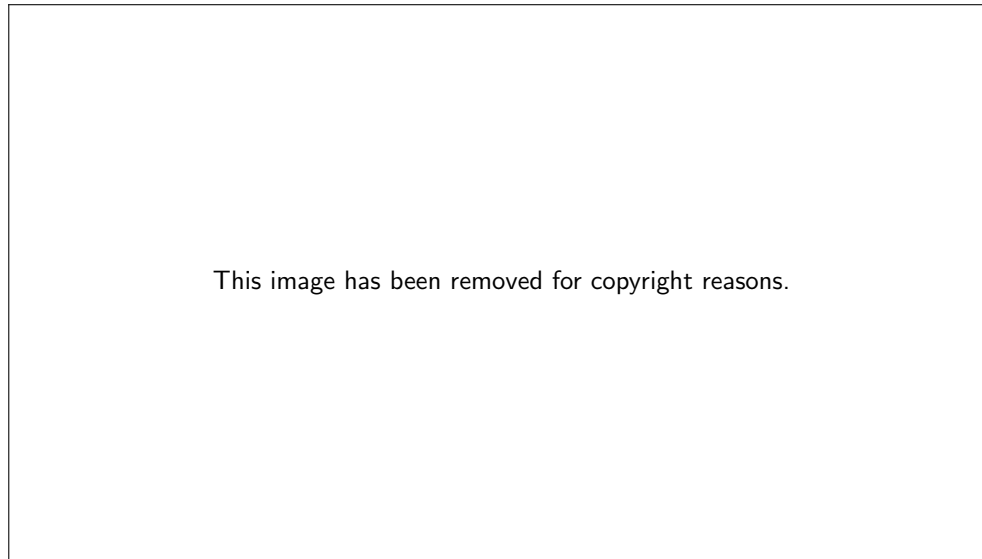


Figure 2.5: *Mafia II* (2K Czech, 2010) screen capture featuring a rendering error where shadowing is incorrectly rendered across half the frame ([Mafia II image screen capture], n.d.).

(Obsidian Entertainment, 2010) also featured character animation problems, where character animations appeared stiff and a character became distorted (Hartel, 2011).

### Invalid Information Access

*Invalid Information Access* occurs when players are able to obtain more information than the game should allow. Examples of this include:

- *Seeing through walls*: This bug occurred in *Fallout 4* (Bethesda Game Studios, 2015), as seen in Figure 2.10, where the player was able to see through buildings at times throughout the game. *Call of Duty: World at War* (Treyarch, 2008) is another example of a game where this occurred (Lewis et al., 2010).
- *Fog of war glitch*: This bug appeared in *Age of Wonders III* (Triumph Studios, 2014) where players could gain access to information they should not have about enemy placements due to a tooltip rendering defect, as seen in Figure 2.11.
- *Graphics settings disparity*: A disparity between the graphics settings in *Overwatch* (Blizzard Entertainment, 2016) allowed players using a lower graphics setting in the game to see an unobscured game environment which players using a higher setting were not able to see, as seen in Figure 2.12.

### Lack of Required Information

*Lack of Required Information* occurs when some required graphical information is not presented on screen. Examples of this include:

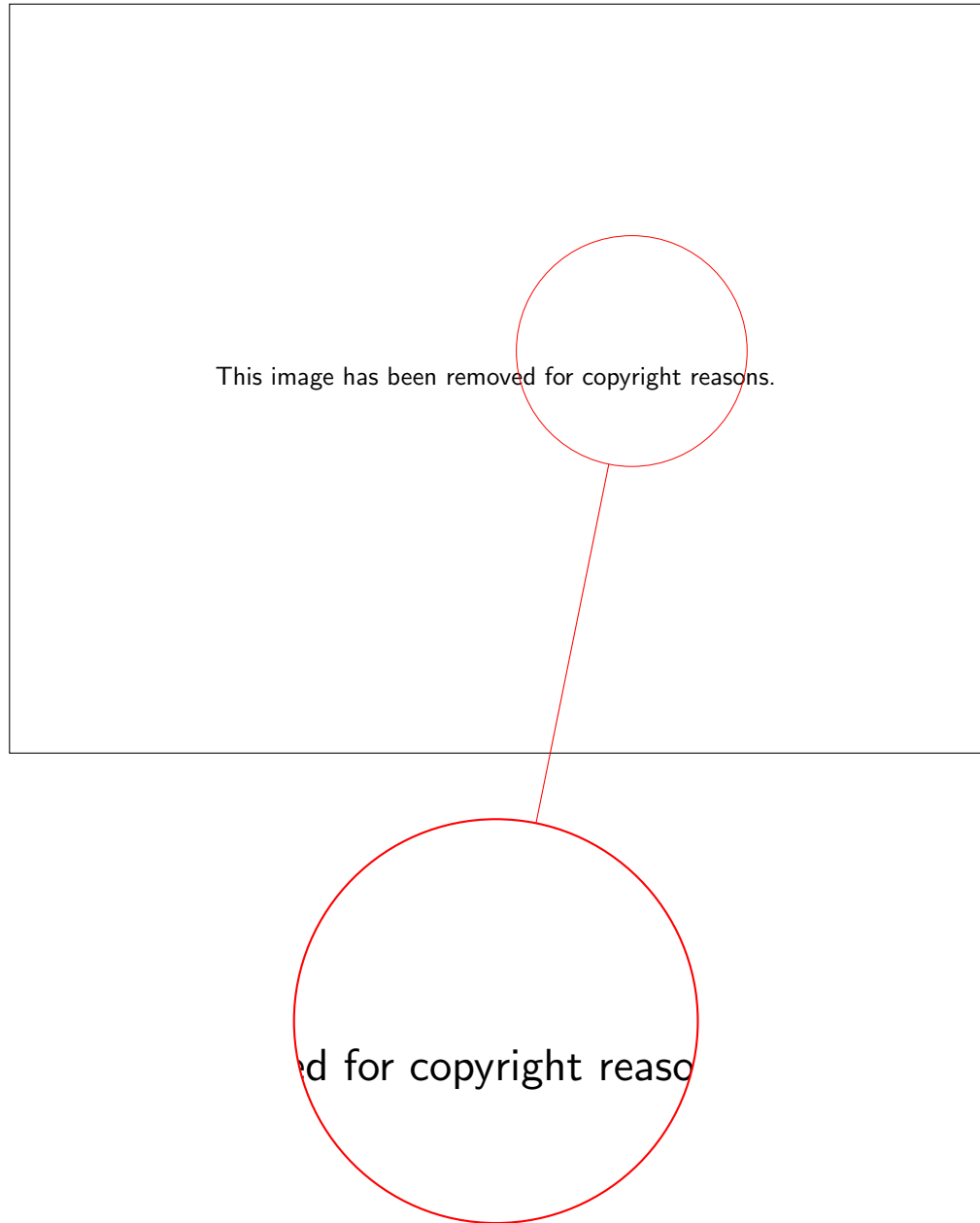


Figure 2.6: *Grand Theft Auto: San Andreas* (Rockstar North, 2004) screen capture featuring a rendering error where the in-game character on the right is missing its head ([Grand Theft Auto: San Andreas image screen capture], n.d.).



Figure 2.7: *Call of Duty: Modern Warfare 3* (Infinity Ward, 2011) screen capture featuring a rendering error where the in-game character is missing its head ([Call of Duty: Modern Warfare 3 image screen capture], n.d.).



(a) Game character's face is incorrectly rendered around the eyes and mouth ([Rocky, facial defects, image screen capture], n.d.).



(b) Game character's limbs have been rendered detached from its body ([Rocky, detached limbs, image screen capture], n.d.).

Figure 2.8: *Rocky* (Rage Software, 2002) screen captures featuring graphical errors.

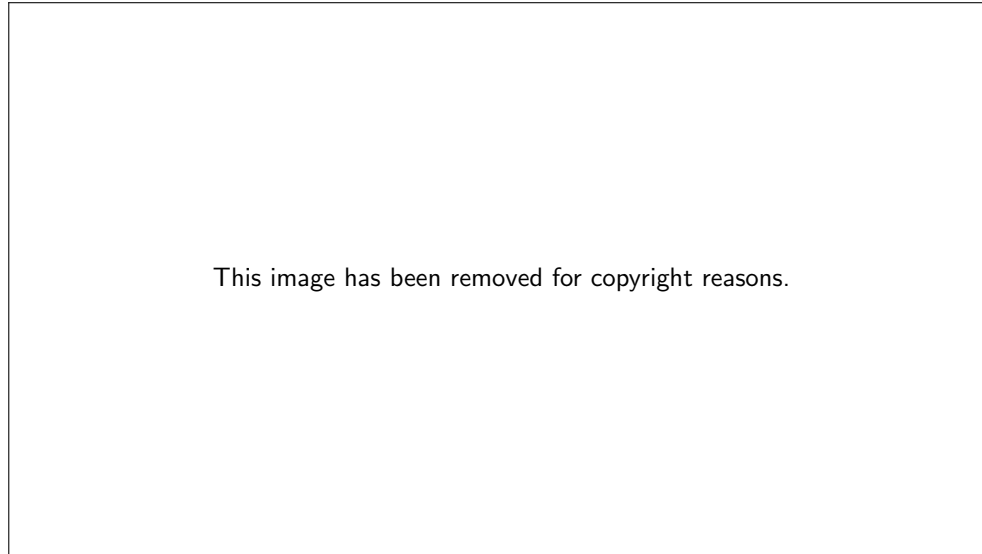


Figure 2.9: *Battlefield 3* (EA Digital Illusions CE, 2011) screen capture featuring a rendering error where the in-game character features an elongated neck ([Battlefield 3 image screen capture], n.d.).

- *Visible artifact*: These types of bugs can also be seen when an artifact obscures or replaces a required element. For example, *Fallout: New Vegas* (Obsidian Entertainment, 2010) featured a crosshatch artifact that was rendered on the terrain in place of shadows (Hartel, 2011).
- *Missing textures*: This occurs when required polygon texturing is not displayed and instead a placeholder texture or flat-shaded polygons are presented (Levy & Novak, 2010). This causes the scene to lack visual detail that the texture would otherwise contain. The Windows PC version of *Rage* (id Software, 2011) suffered from texturing issues on launch, as seen in Figure 2.13. *Fallout: New Vegas* (Obsidian Entertainment, 2010) featured textures that would not load correctly (Hartel, 2011). The *Xbox 360* version of *The Elder Scrolls V: Skyrim* (Bethesda Game Studios, 2011) contained texture scaling issues that required the developer to patch the game after its initial release (Metro, 2011) and the cross-platform *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) featured delays in texture loading that caused graphical bugs (NOWGamer.com, 2014).
- *Obscured UI*: The Windows PC version of the cross-platform *Assassin's Creed IV: Black Flag* (Ubisoft Montreal, 2013), which was released after the console platforms, featured graphical problems including the erroneous blurring of user interface (UI) elements, as shown in Figure 2.14 (Asif, 2013).

### Implementation Response Issues

*Implementation Response Issues* refer to bugs that occur when the video game experiences frame rate fluctuation. Examples of this include:

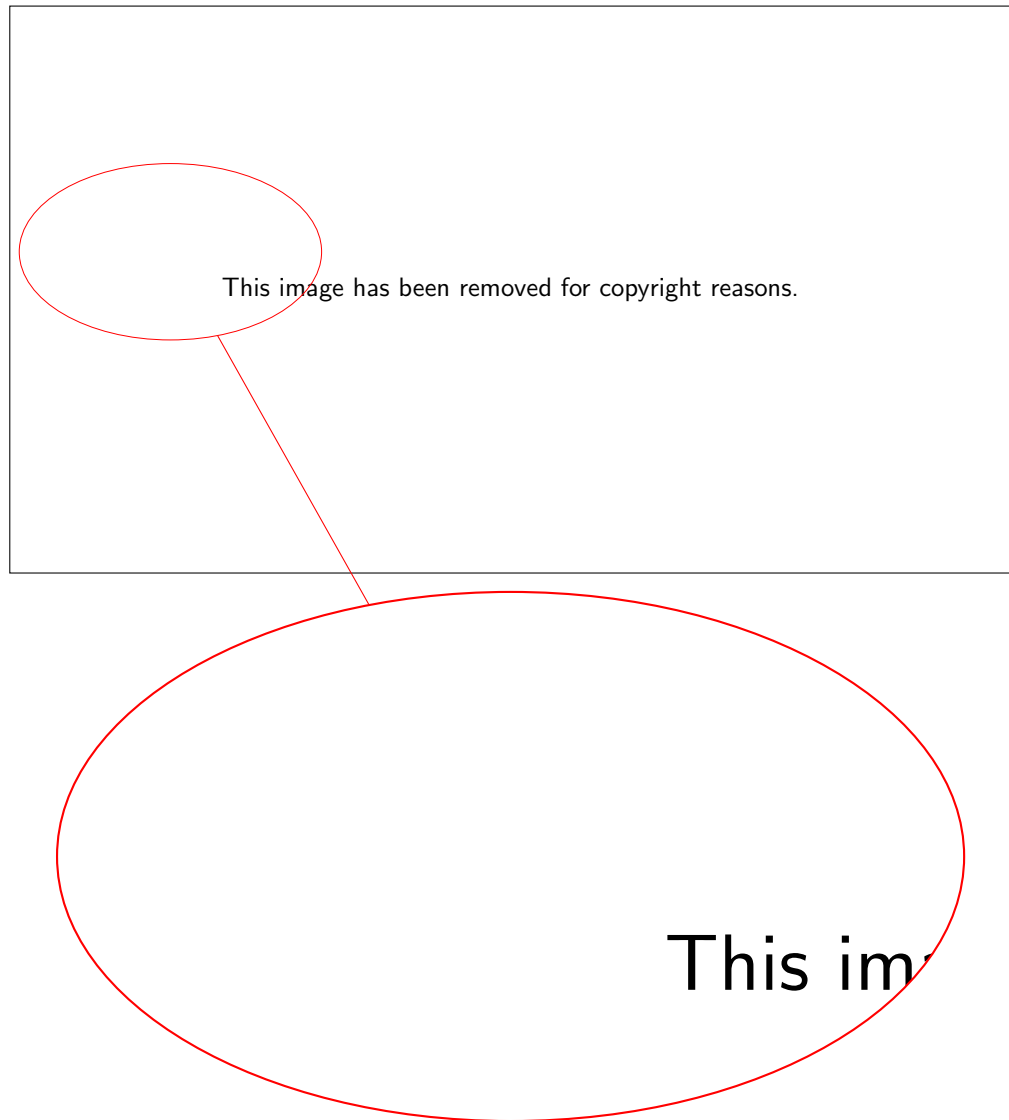


Figure 2.10: *Fallout 4* (Bethesda Game Studios, 2015) screen capture featuring a rendering error where the player can see through the building on the left of the frame ([Fallout 4 image screen capture], n.d.).

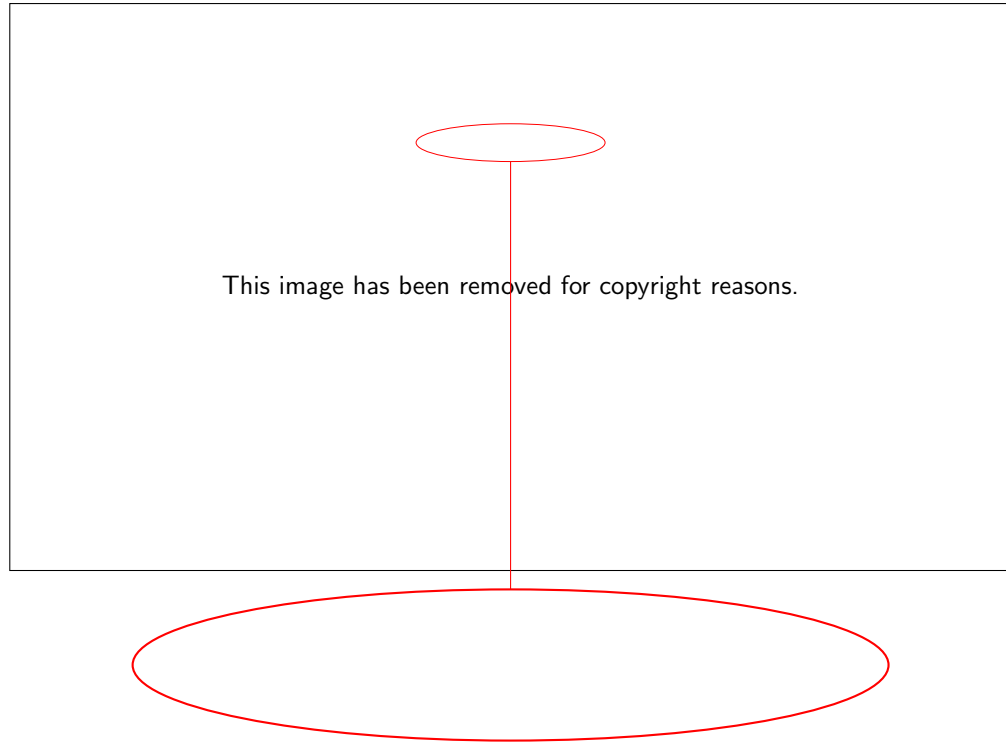


Figure 2.11: *Age of Wonders III* (Triumph Studios, 2014) screen capture featuring a rendering error where red tooltip text is incorrectly rendered at the top-centre of the frame, revealing the secret location of an enemy army obscured behind clouds ([Age of Wonders III image screen capture], n.d.).



(a) Low graphics setting ([Overwatch, low graphics setting, image screen capture], n.d.).



(b) Higher graphics setting ([Overwatch, higher graphics setting, image screen capture], n.d.).

Figure 2.12: *Overwatch* (Blizzard Entertainment, 2016) screenshot comparison of the same scene rendered on two different graphics settings, showing that a player using a lower graphics setting can see more of the game environment than a player using a higher graphics setting, due to missing foliage.

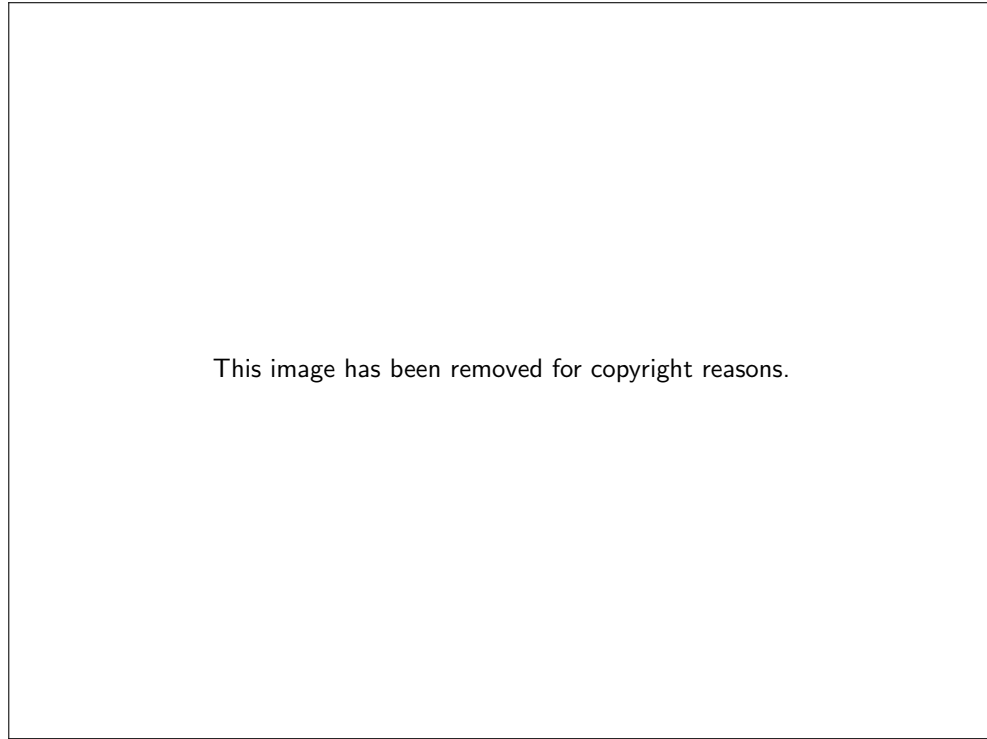


Figure 2.13: *Rage* (id Software, 2011) screen capture featuring a texturing issue where the game world exhibits a checkerboard pattern ([Rage image screen capture], n.d.).

- *Screen-tearing*: This bug is generally performance-related, where the GPU and display system are not in sync (Levy & Novak, 2010). This causes the display to show portions of two different frames rendered by the GPU at the same time. Levy and Novak (2010) suggest this bug could be related to the performance capabilities of a target platform. This occurred in the Windows PC version of *Rage* (id Software, 2011) which was released featuring screen-tearing (Sharke, 2011) and in the *Xbox One* version of *Watch Dogs* (Ubisoft Montreal, 2014b) (Usher, 2014).
- *Low frame rate*: This bug occurs in *Tom Clancy's Rainbow Six: Vegas 2* (Ubisoft Montreal, 2008) (Lewis et al., 2010). *Total War: Rome II* (The Creative Assembly, 2013) had poor performance with low frame rates and graphical glitches (Younger, 2013). The PC version of *Watch Dogs* (Ubisoft Montreal, 2014b) had frame rate issues and slowdown when played with an AMD graphics card (Usher, 2014). *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) also featured frame rates lower than 30 frames per second on both *PlayStation 4* and *Xbox One* (NOWGamer.com, 2014).

Graphical errors in commercial video games are not limited to a single developer, game engine or platform. Table 2.1 shows the variety of game engines used across the video game examples found in the internet search, alongside their associated target platforms. This shows that bugs occur on different platforms, across different developers and on different game engines, some of which are licensed and

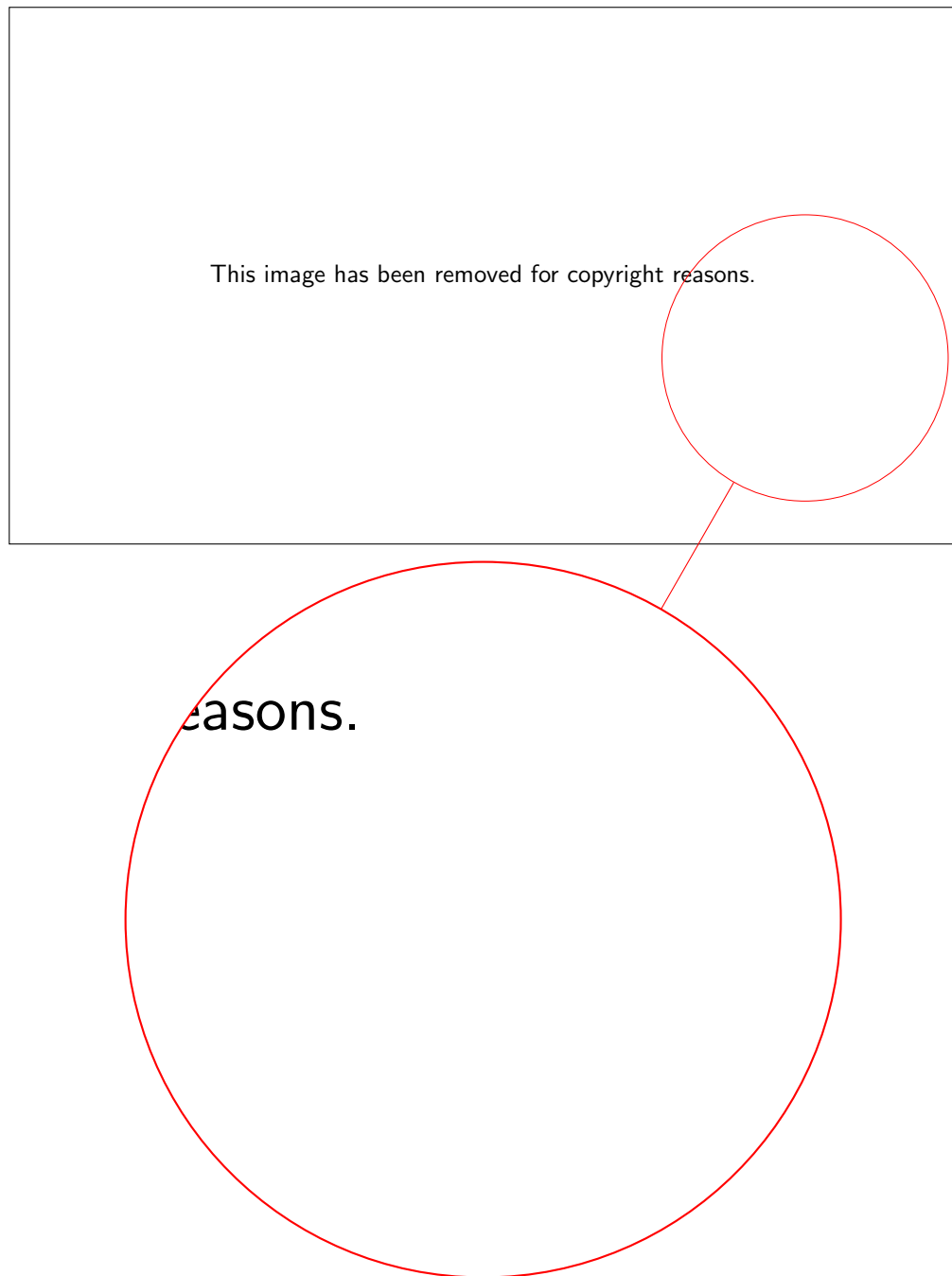


Figure 2.14: *Assassin's Creed IV: Black Flag* (Ubisoft Montreal, 2013) screen capture featuring blurred text in the heads-up-display of the user interface ([Assassin's Creed IV: Black Flag image screen capture], n.d.).



some of which may be proprietary, and therefore cannot be blamed on any one of these individually. As Hight and Novak (2008) state, Class C bugs that cause aesthetic issues do not prevent the release of a game. Since these bugs are all graphical in nature, they may have been considered Class C, and therefore not considered important enough to address prior to release, despite the impact they have on gameplay, performance, and reputation.

Published Game	Game Engine	Target Platforms	Visual Defect
Age of Wonders III	Creator Engine	Windows, Mac OS X, Linux	Fog of war glitch
Assassin's Creed IV: Black Flag	AnvilNext	Windows, PlayStation 3, PlayStation 4, Wii U, Xbox 360, Xbox One	Obscured UI
Assassin's Creed Unity	AnvilNext	Windows, PlayStation 4, Xbox One	Facial defects, Missing textures, Low frame rate
Batman: Arkham Knight	Unreal Engine 3	Windows, Mac OS X, PlayStation 4, Xbox One, Linux	Visible artifact
Battlefield 3	Frostbite 2 Engine	Windows, PlayStation 3, Xbox 360	Character animation hierarchy defects
Battlefield 4	Frostbite 3 Engine	Windows, PlayStation 3, PlayStation 4, Xbox 360, Xbox One	Visible artifact
Call of Duty: Modern Warfare 3	MW3 Engine	Windows, Mac OS X, PlayStation 3, Xbox 360, Wii	Missing head
Call of Duty: World at War	IW 3.0	Windows, Playstation 3, Xbox 360, Wii	Seeing through walls
The Elder Scrolls V: Skyrim	Creation Engine	Windows, PlayStation 3, Xbox 360	Missing textures
Fallout: New Vegas	Gamebryo	Windows, PlayStation 3, Xbox 360	Shadows, Character animation hierarchy defects, Visible artifact, Missing textures
Fallout 4	Creation Engine	Windows, PlayStation 4, Xbox One	Seeing through walls
Garry's Mod	Source	Windows, Mac OS X, Linux	Z-fighting
Grand Theft Auto: San Andreas	RenderWare	PlayStation 2, Windows, Xbox, Mac OS X, Xbox 360, PlayStation 3, iOS, Android, Kindle, Windows Phone, Fire TV	Missing head
Mafia II	Illusion Engine	Windows, Mac OS X, PlayStation 3, Xbox 360	Visible artifact, Shadows
NBA 2K13		Windows, PlayStation 3, Xbox 360, Wii U, Wii, PlayStation Portable, iOS, Android	Clipping
Overwatch		Windows, PlayStation 4, Xbox One	Graphic settings disparity
Rage	id Tech 5	Windows, PlayStation 3 Xbox 360, Mac OS X	Screen-tearing, Missing textures
Rocky		GameCube, PlayStation 2, Xbox, Game Boy Advance	Facial defects, Character animation hierarchy defects
Tom Clancy's Rainbow Six: Vegas 2	Unreal Engine 3	Windows, PlayStation 3, Xbox 360	Low frame rate
Total War: Rome II	Warscape Engine	Windows, (Future targets: Mac OS X, Linux)	Low frame rate
Watch Dogs	Disrupt engine	Windows, PlayStation 3, PlayStation 4, Xbox 360, Xbox One, Wii U	Screen-tearing, Low frame rate

Table 2.1: Summary of example video games with rendering errors, their associated game engine and target platforms.

## 2.3 Automated Testing in Games

Finding bugs is repetitive, time consuming and challenging for human testers, and this could cause testers to overlook defects (Nantes, Brown, & Maire, 2008). Removing the human testers could reduce the number of hours spent testing. Scholars (Murphy-Hill et al., 2014; Petrillo et al., 2008; Scacchi & Cooper, 2015) highlight a lack of test automation in game development and suggest that there is some reluctance in the game industry to adopt automated testing practices. A report by the International Game Developers Association (IGDA) identified automated testing as best practice (Rees & Fryer, 2003) and Carucci (2009) encouraged AAA game developers to adopt automated testing practices. Automated testing has been broadly accepted in the software development industry, yet automated testing practices go against the common practices undertaken by the game development industry. The excuses used in the game industry to avoid automated testing include: the belief that writing tests takes time away from the actual development process; automated tests cannot cover everything; source code is temporary and highly likely to change as development progresses, hence the code does not need to be tested when written; the process of game development is different to software development, and automation may not be suitable; and, games are about fun, and capturing fun with an automated test is not possible (Carucci, 2009).

Automated testing can require the involvement of programmers, who cost more to write code than testing staff, and the automated testing code may not be bug-free, nor be reusable from one game to another, or from one platform to another (Rees & Fryer, 2003). Automated testing is also considered to reduce the agility of development practice (Murphy-Hill et al., 2014). Since almost all games can be considered emergent, where the execution of the software results in unpredictable game outputs, they become difficult to test. Testing requires knowledge of the expected output, and emergent games hinder the ability to predict what is expected (Lewis & Whitehead, 2011b). The IGDA report argued that automated testing cannot substitute for human testers because humans have an eye for detail that cannot be replaced. The fragility of automated testing, due to constant changes to a project, was cited as another reason for the reluctance to adopt automated testing practices, with human testers considered more resilient. However, there are disadvantages to using human testers to identify and report upon the discovery of bugs. For example, tester feedback may be influenced by being observed while playing a game (Schultz et al., 2005). Developers may have issues replicating the circumstances that discovered the bug, and test automation could enhance bug reproducibility. A split between human and automated testing is proposed, where automated testing tools are used to speed up the testing process rather than replace human testers (Rees & Fryer, 2003; Ruskin, 2008). This multifaceted approach could be used to catch a wider range of defects (Keith, 2010).

The IGDA report also discussed when best in the development cycle for testing to occur, stating

that testing early in production allows bugs to be caught sooner, but warned that this could be overwhelming for developers, since a game is likely to contain a large number of defects early in development (Rees & Fryer, 2003). Kanode and Haddad (2009) suggested that testing should occur throughout development. A game should be planned to be playable at the end of each development cycle, allowing testers to check the game for defects at these points while the game is in a playable state (Kanode & Haddad, 2009; Keith, 2010). Lewis and Whitehead (2011b) suggested that constant experimentation is required for successful game development, and that this style of development needs to be kept in mind when creating or adopting testing practices.

Carucci (2009) stated that since game source code is algorithmic in nature, it is easier to test than GUI source code. Since game source code changes often, there should be more incentive to use automated testing to ensure that the side effects of changes are avoided. Automated tests can be used to discover defects and bugs effectively, and hence allow more time for game developers to create the fun aspects of a video game. They can be small, simple and straightforward in their approach, testing a single condition, and with the added benefit of being self-documenting and repeatable.

Scripted simulation is used in game testing, suggesting some form of automation, however, regression testing, which ensures that previously working features are not broken as development continues, and unit testing, which tests individual procedures, are not utilised to the same degree as in software development (Hunt & Thomas, 2000; Murphy-Hill et al., 2014; Myers et al., 2012). Carucci (2009) proposed that the game industry should adopt more test automation by creating high-level testing frameworks. A strong commitment from game development studio management, alongside entire team member buy-in, mentoring and training, is required to make test automation within game development successful (Carucci, 2009).

While it is clear that there is support for automation from the IGDA and industry insiders, such as Carucci (2009), the competitive nature of the industry and the prevalence of non-disclosure agreements makes it difficult to find details on the exact kinds of automation that are taking place. There is academic research available, for example regarding Unity Technologies and Electronic Arts, however it seems to only scratch the surface of what automated testing is happening and how. Further insight can be gained from employee blog posts that support the published academic research presented in the following sections.

### **2.3.1 Industry Insight: Unity Technologies**

In their SIGGRAPH article, Zioma and Pranckevicius (2012) shared their experiences and lessons learnt in maintaining acceptable quality and performance levels across Unity game engine target platforms, including multiple mobile devices and operating systems. Unity supports several different mobile

GPU architectures, each with their own unique characteristics, such as texture compression, API extensions, performance analysis tools, and different graphical drivers. The authors noted that each graphical driver has its own set of bugs. They adopted a functional test suite for automated testing on their target mobile platforms. Zioma and Prankevicius (2012) did not present any findings of their activities, however in a related presentation, Zioma (2012) provided more insight into the overall process, as well as providing some of their results.

The presentation was intended for developers who wish to develop their own technology from scratch, or who want to obtain insight into the under-the-hood details of the Unity engine (Zioma, 2012). It also provided some insight into automated graphics testing within Unity and highlighted platform-dependent challenges, such as the different GPU architectures and their associated rendering strategies, and the large variety of mobile device resolutions and varying performance scales. No single texture format is supported across all of Unity's OpenGL ES 2.0 graphics devices. There were performance variations that reminded the author of developing for graphics devices in the 1990s. Unity automated testing by executing the same content on different devices, including different operating system updates (Zioma, 2012). They captured screenshots which were compared on a per-pixel basis to template images. The test suite featured 238 test scenes, each one a simplified scene that tested a specific graphics feature. The target devices used were the Nexus One (Adreno 205), Samsung Galaxy S2 (Mali 400), Nexus S / Galaxy Nexus (SGX 540) and Motorola Xoom (Tegra2). The automated tests were run automatically upon internal code changes. The author found test results differed from device to device. Examples of the test scenes were shown — an alpha blending scene, a blend mode scene and a scene that used `ReadPixels` to demonstrate particular issues that occurred. These test scene results were shown for two different hardware devices where the resulting renders were different. Graphics shader variation was found to produce incorrect results on some targets. Also, some devices simply crashed on execution of some tests, which Zioma (2012) attributed to driver issues, and connecting multiple devices to a single host computer caused device connection failures.

In a personal website blog post, Prankevicius (2007) provided further insight into the automated graphics testing conducted at Unity Technologies. Although the automation setup took time, it was beneficial, as it highlighted the differences between hardware devices, device drivers, and the potential problems that can arise as a result of different behaviour on various hardware targets, or code behaviour based upon hardware and driver combinations. The author highlighted the need for the right hardware to conduct testing on and suggested this could initially be conducted with a couple of graphics cards that can be swapped in and out of a computer. Unity Technologies' setup involved buying a variety of test machines, with integrated graphics cards that were of interest, but also allowed discrete graphics cards to be utilised with the same test hardware. This resulted in shelves of graphics cards, including

legacy hardware such as the ATI Rage, Matrox G45 and S3 ProSavage (Pranckevicius, 2007).

Functional tests were created based upon small test scenes that featured a variety of possible rendering scenarios. Examples of these were: do all blend modes work; do light cookies work; does automatic texture generation work; do texture transforms work; does the rendering of particles work; does glow image post processing work; does mesh skinning work; and, do shadows based on a point light work. This resulted in many tests, each of which was small and isolated. Tests were loaded into the Unity engine in succession, and a screenshot taken of each resulting scene. When simulation was required, such as in an animation or particle test, time was updated at a fixed rate. Screenshots were taken on the fifth frame of the simulation to give the test scene a chance to warm up (Pranckevicius, 2007). When new graphics card hardware, drivers, or operating systems were released, the tests were run and obvious errors could be found quickly by a human tester who compared the screenshots. When changes in the rendering code occurred, tests were rerun to check if any graphics features had become defective.

Further automation occurred via scripts which produced a suite of test images for current hardware, and another where all tests were executed and the results compared with a known set of correct images. The stored correct images were different for each graphics card due to the capabilities of the individual hardware. The automated tests could be run for various driver versions on every graphics card in the test suite. If a test failed the defective screenshot was copied alongside the correct image, and these were uploaded to a wiki which logged the test results. Overall, this automated system was considered helpful to the development of the Unity game engine, specifically when the developers implemented the Direct3D renderer, and in the discovery of techniques to work around graphics card- and driver-related issues (Pranckevicius, 2007). Pranckevicius (2007) noted that automated testing could be extended to all driver versions. A test could install one driver in silent mode, reboot the test machine, rerun the graphics test scenes script, and then move onto the next version of the driver.

From the initial work undertaken in 2007, Unity Technologies continued to evolve their automated testing practices. In a blog post on the Unity Technologies website, Dunnett (2010) discussed regression testing of games in Unity's web player. The goal of the regression testing was to ensure that the web player was able to play back content identically in all its published versions, so that engine customer's games were not broken by new versions. The development team built a dedicated test system which was composed of a server, and seven PC and two Mac client machines. During execution of a game, the client machines took a screenshot every second which was then forwarded to the test server. A human tester played the game on the regression system, which captured all the input and stored it. This created a golden set of screenshot images based upon the input, which became the known set of images that the game should generate. The regression system could compare images from the game

playback using the daily development build of the web player against the golden images. Developers were alerted when a mismatch occurred (Dunnett, 2010).

In 2011, Pranckevicius (2011) revisited his 2007 blog post with a follow up post which discussed regression testing of games developed with the Unity engine, showing more automation development and providing more behind-the-scenes details. Graphics programmers, who care whether features developed actually work or if changes to the code base caused a feature to break, utilise the types of small functional tests described earlier (Pranckevicius, 2011). The post noted that a graphics programmer was responsible for creating the tests which validated the graphics features they created. Further example screenshots of the system were shown for two cases — a fog-handling test scene and a deferred lighting test scene. Engine features were also tested by QA and a beta testing group. When QA discovered a fault that did not yet have a test, a test could be added to check for that particular situation, and then retained for use in future automated testing. TeamCity, a third-party continuous integration and build management system, was utilised for the build and test farm where several build machines were set up as graphics test agents (Pranckevicius, 2011). These agents ran graphics test configurations on all branches of the engine automatically. Daily graphics tests were also run for each branch. Any branch with high levels of activity in the graphics code had tests run more frequently than daily. Tests could also be run manually at the click of a button. Pranckevicius (2011) summarised that the overall approach was to run test scenes for a fixed number of frames and take a screenshot at the end of each fixed time-step. Screenshots were then compared against known good images and any difference was considered a failed test result. On some platforms a tolerance for some number of wrong pixels was allowable. The test controller was a C# application that listened on a socket, fetched the screenshots and compared them with the known good images. Failed tests were reported with an expected image, a failed image, and a difference image with increased contrast.

In 2012, a further Unity Technologies blog post highlighted the introduction of a testing team and significant resource being directed towards test automation. Petersen (2012) stated that there was a QA team of seventeen, comprised of software engineers and students, and the company was hiring more. The roles were broken into different categories. Firstly, *Software Engineers in Test*, who were responsible for developing tools, frameworks and automation that would improve the overall development work flow. Secondly, *Software Test Engineers*, who were manual testers that became the first users of newly developed engine features and produced feedback for developers. Finally, the student workers, who looked at the bugs submitted by the Unity user community and then tried to reproduce them in the engine, before forwarding them onto the engine’s developers. Petersen (2012) stated that Unity was one of the most highly tested products he had ever worked on. The strategy for testing required automation due to the lifespan of the features the engine contained. Repeated

manual testing of features which could have a lifespan of years did not make sense, and so automation was needed. The infrastructure required to run the test automation was small and the core part of Unity required only a single install on a single machine. In a regular test run, all target platforms were covered by the testing. Bug reports detailed specific platform, operating system and graphics card specifications. Writing automated tests was extremely difficult, and required developers who knew about testing techniques, infrastructure, and the product in development, and were programmers who could write solid code (Petersen, 2012).

Andersen (2013) also provided insight into the *Toolsmith Team* at Unity Technologies. This team consisted of six developers responsible for working on tools, frameworks infrastructure, and testing Unity. The team was additional to the team categories described by Petersen (2012). Andersen (2013) identified challenges of testing Unity, such as how to break the product down into logical components and the more than ten target platforms that the engine ran on. This made the testing matrix extremely complex due to different platform-specific areas and features. The post provided insight into the runtime API test framework where platforms were treated separately from engine features. The framework consisted of different modules: automation; players; frameworks; runner; and, runtime test cases. A test was written once and the framework would ensure it ran on each target. This framework was supported on all Unity target platforms and the build system was able to execute tests on multiple platforms in parallel. There were more than 750 unique runtime API test cases and more than 9000 test cases executed in a single automated test build verification. However, average execution time for the entire test suite was less than 30 minutes, with half of this time spent setting up the environment and the other half conducting the tests. Test cases were written in C# and executed in either milliseconds or in under a few seconds, and individual test cases could be disabled or could be platform-specific. The back-end of the testing system had close to real-time monitoring of key test metrics. In a single day, more than 150000 API tests were executed. This post from 2013, highlights the growth of multiple testing teams within Unity Technologies, and reveals, to some extent, the scale of automated testing within the company. In 2012, Zioma (2012) stated that there were 238 graphics test scenes, and now there were over 9000 automated test cases for the engine.

Alistar (2014) discussed why it is critical to have a team of expert developers focused on testing the Unity engine. Unity Technologies employed 450 people, across 27 locations and had 2.5 million registered developers using the Unity engine. The engine supported twelve major platforms, as well as some smaller platforms. The engine's codebase grew fast, with 2.2 million lines of source code at the time of writing, and the developers needed to ensure that the engine performed well with each new release. The team placed great emphasis on automation. 7000 files and 400000 lines of code were dedicated to the testing of runtime, graphics, integration, unit, UI, and performance. There were





Figure 2.15: Unity graphics test result example ([Unity reference image, test result image, difference image], n.d.).

more than 13000 test cases in a single automated build verification suite, up from 9000 test cases the previous year (Andersen, 2013). The team could be working in 100 development branches at the same time, with half a million test points executed on the build farm every day. Automation-based work was not tied to any single release of the Unity engine. The *Software Development Engineers in Test* (SDET) wrote, maintained, optimised and improved the tests on a daily basis. The SDET team was formed in 2012, as mentioned in the Petersen (2012) blog post, as the few testing frameworks that existed were not owned, maintained or improved by any particular development team members. They cleaned up, documented and optimised all existing tests, and ensured that all developers knew how to utilise the frameworks, as well as create new tests. The SDET were required to be strong collaborators as they worked with other teams in QA and research and development. They had to be knowledgeable in programming, object-oriented design, unit testing, test driven development, high-level testing and test automation.

Bay (2016) revealed further details about Unity Technologies’ automated graphics testing in a 2016 blog post on the company website. The system has grown to over 13700 graphics tests over 33 build configurations, again highlighting that different platforms, device models and graphics cards produced slightly different results. Graphics test scenes have been created which run on all supported devices and render a resulting image, which is then compared with an approved reference image. When a test fails, a difference image is created and then a human needs to verify why this failure has occurred. An example of a test scene and its corresponding failed result and difference images can be seen in Figure 2.15. Overall results are displayed on a webpage, an example of which can be seen in Figure 2.16. The tool allows toggling between reference image, test result image and difference image to help the human identify the graphics issue. There are a number of steps required of the operator in validating test results and updating reference images. Running the tests and waiting for the results can be time consuming, however, overall, the test system has helped reduce manual overhead when dealing with graphics issues.

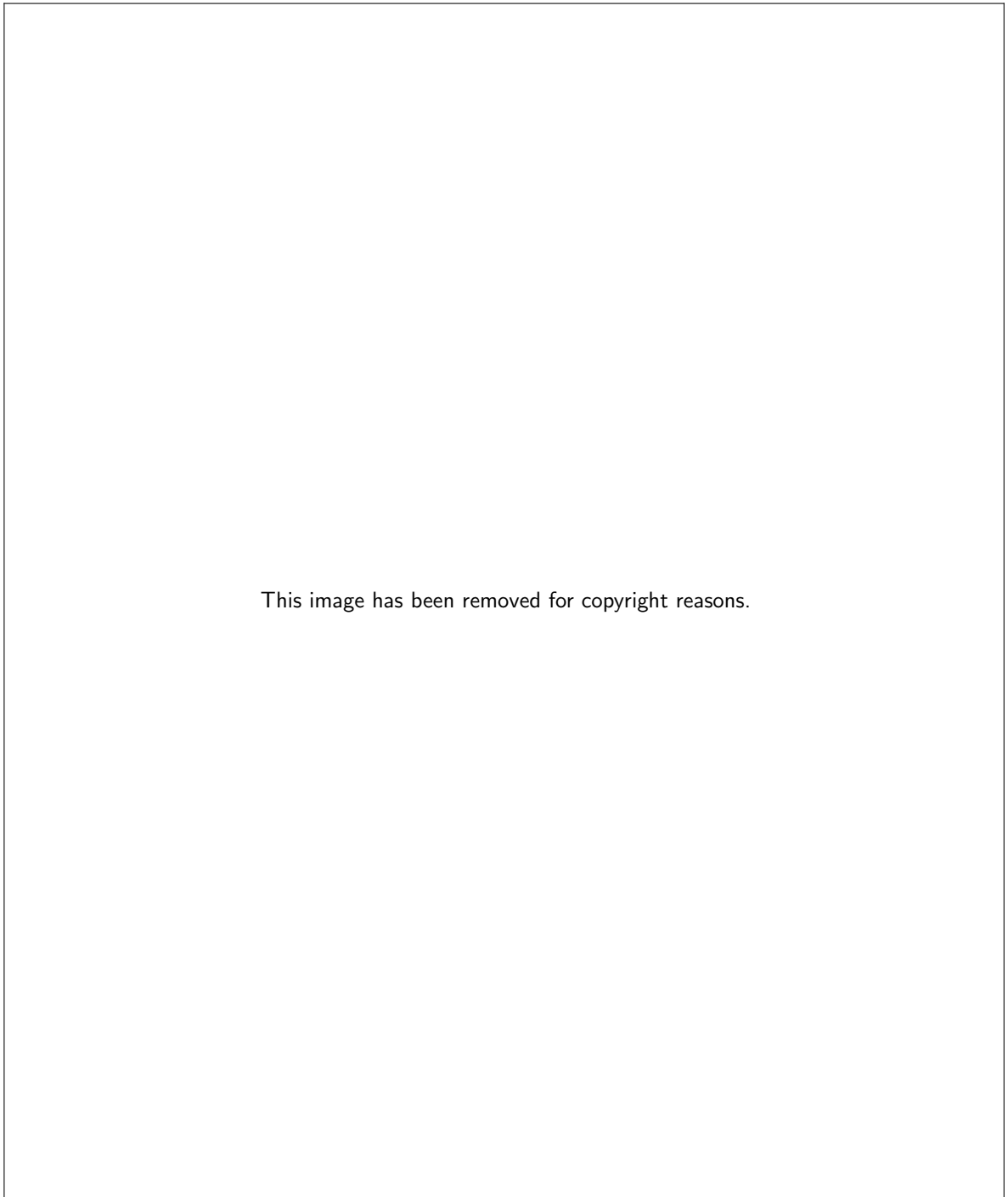


Figure 2.16: Unity *Graphics Tests Tool* screenshot ([Unity Graphics Tests Tool image], n.d.).

Automated testing has helped keep the quality of the Unity game engine high, by ensuring that known bugs are resolved and never reoccur (Alistar, 2014). Though the literature contains minimal detail on how automated testing is implemented within Unity Technologies, a broad picture can be formed from employee insight. The works show how quickly automated testing has grown in scale and complexity within the Unity game engine, and highlight the continued growth and dedication of significant human and technological resources to this area. It is important to note that the automated testing undertaken by Unity Technologies is only for their own engine and is not open and transferrable to competitors' engines. Nor do they make games, and so do not have to focus resources on content development. However, it is an excellent example of the automated testing that could be used industry-wide if game development studios are prepared to direct human, technological and financial resources to this area. Through the growth of their testing staff and technology, Unity Technologies demonstrate what is possible when there is buy-in from management, as championed by Carucci (2009).

### 2.3.2 Industry Insight: Electronic Arts

In conference proceedings, Buhl and Gareeboo (2012) presented a case study and lessons learnt from the application of automated testing during game development at EA Tiburon, an Electronic Arts (EA) development studio. Automated testing was identified as a key factor in improving the development of the *NCAA Football* game franchise created by the studio. In the case study, the Alpha phase of development was problematic, as significant QA resources were allocated to testing the game, while the game development team focused on fixing identified bugs (Buhl & Gareeboo, 2012). The team had a ratio of five software developers to one QA staff member. Developers found it was difficult to see how stable the game was and game features that were not used for weeks would instantly crash when executed. Developers were left with no detail as to when or how defects were introduced to the game. The high number of defects caused an increase in work hours for the development team, which could have led to staff burnout and higher staff turnover, and hence could have impacted the next game the studio developed (Buhl & Gareeboo, 2012). Buhl and Gareeboo (2012) identified the need to introduce new processes to development during production to reduce the number of defects in a game prior to the Alpha milestone. QA staff were used during development to test the game prior to repository check-in, however it was argued that this practice would not scale to larger teams.

To implement automated testing, the development team's project manager took responsibility for championing the idea and obtaining buy-in from senior management (Buhl & Gareeboo, 2012). The project manager also prioritised the investment of time and resource into automation. Initially, a reporting dashboard was created to track the automation project, with green tick and red cross status feedback. This provided a simple, visual way of checking whether features were broken or not.

Over time, more symbols were added to identify different issues: orange triangles denoted initialization issues, exclamation marks denoted missing script issues and red diamonds were used for assert failures. The EA Tiburon studio had access to previous work created by the *EA Central Test Engineering Group*, which contained a foundation for proprietary automated game testing. They went on to create an automated testing infrastructure that obtained every new build of a game, copied the build to target consoles, ran a series of test scripts and then reported the results to a dashboard that was accessible to the whole team. Automated bots ran every night, where AI scripts played the game continuously over a set period of time. These results were also reported to the dashboard. Software engineers could use a simple, non-time consuming process to execute tests from their desks. A critical path test for entering and exiting each game mode in under ten minutes was also created; this was convenient for engineers to run and it avoided interference with the general work flow. A team member, who was recognised by the development team as an individual who consistently made stable updates to the game, noted that the critical path testing saved time, as the individual had previously conducted this style of test manually prior to each of their code check-ins. Buhl and Gareeboo (2012) note that by keeping tests in sync with the current version of the game and by running the tests prior to check-in, previous issues with automated testing were avoided.

As a result of their system, the longest period of time where *NCAA Football 12* (EA Tiburon and EA Canada, 2011) went without a passing build was three weeks, compared to *NCAA Football 11* (EA Tiburon and EA Canada, 2010), which went five months without a passing build (Buhl & Gareeboo, 2012). Automation improved the product quality, and the authors noted the positive impact of three weeks compared to five months on the development time, cost and team member morale. They also found that automated bots were stable earlier in the development cycle and the extra hours required by the development team were reduced during the Alpha stage of development. There was a major reduction in the Class A bugs found compared to the previous release of the game, which allowed QA more time to discover low severity bugs. Buhl and Gareeboo (2012) noted that automated testing would not have been possible without management buy-in. The associated presentation by Buhl and Gareeboo (n.d.) provided examples of the dashboard output, testing result graphs and identified *NCAA Football 08* (EA Tiburon, 2007) as having a particularly bad Alpha. This indicates that there were a number of yearly releases of the *NCAA Football* game franchise before significant work was undertaken to improve the automated testing process.

Donat, Husain, and Coatta (2014) discussed the potential for automated QA testing and provided further insight into automated testing conducted at EA. The automation of testing was initially explored for *FIFA 10* (EA Canada, 2009). The developers were motivated by the cost of manual testing which had become significant due to the complexity of the game, highlighting the expense of

code changes that require manual retesting. Automated testing had the benefit of allowing testers to refocus on the authenticity and fun of the game, rather than spending time testing for stability. Testing required 22 humans to play the game. Therefore, to test two simultaneous matches, over 40 human testers were required. This required the organisation and coordination of the human testers, and test scenarios would be hindered when the testing group encountered a trivial bug that caused a fault at the beginning of a match. During testing the budget had to be increased, which prompted the studio to seek ways to avoid this happening in the future and provided the motivation for automated testing.

A time-based script was utilised, which sent commands to 20 test consoles, stepping through the initial match setup phases of the game (Donat et al., 2014). The timing of this script was critical; if it was not perfect, the setup routine would fail. The developers analysed 300 crash bugs from a QA cycle and found that more than half the bugs occurred during the initial screen transitions. This helped them realise testing was required in both the front-end and the back-end of the game.

The initial developers of the *FIFA 10* automated test framework needed the test program to be aware of where it was on a console and then to move forward in an error-correctable way. The capability to do this was developed, but was neglected, and by the time the development team of *FIFA 11* (EA Canada, 2010) needed the testing feature for the next version of the game, it did not exist in a usable form. The development team of *FIFA 11* added an auto-assist feature, where automation could be left to run the game. This reduced the number of human testers required from over 40, to just one or two. The work conducted on *FIFA 11* helped to convince EA that there were benefits to automated testing. However, Donat et al. (2014) also highlighted that a dedicated software engineer was required to maintain the automated test script, as it was fragile. The implementation developed was also described as too specific, and it was suggested that a higher-level abstraction should be developed.

Testing logic using the view layer of the game was identified as error-prone, as it relied on waiting for buttons to load and the simulation of mouse clicks and keyboard event messages, which can break easily (Donat et al., 2014). Automated systems need to be separated from the data they require, and development strategies need to be employed where the game software utilises observer pattern behaviour and the Model-view-viewmodel (MVVM) architectural pattern. This would allow events to be fired, and listened for, when interesting events occur in-game. Donat et al. (2014) identified that software engineers must be convinced to change their approach to development.

Development of the *FIFA* game franchise is undertaken worldwide, and Donat et al. (2014) highlighted that the savings attributed to the adoption of automated testing were hard to quantify. They warned that adoption may require moving resources from one type of testing to another and the cost of developing automated testing systems could be significant due to the need for expensive programming

staff. They advised that it would be important to find a balance between the work conducted by low-cost human testers and high-cost specialists. However, it is noted that even though they may be expensive, specialised developers utilising the right set of tools, languages, frameworks and paradigms, could provide great benefit (Donat et al., 2014).

The literature provides insight into automated testing conducted at various EA studios and across two game franchises which feature yearly releases. Automation reduced the burden on both developers and human testers, however it could have introduced added cost to game production, and it required programmers with a specialised focus on automated testing. Benefits existed, as later games in the *NCAA Football* franchise were more stable earlier in the production cycle, and in the *FIFA* franchise, they were able to reduce the number of human testers required to test the games. While the literature does not provide specific detail as to how the automated testing was implemented in either case, nor is it specific about whether the testing focused on graphical defects, it does show that automation can be beneficial to the game industry.

### 2.3.3 Games-related Testing Automation Research

The previous sections provided insight into the large-scale testing occurring within industry. Unity Technologies conducts graphics testing for its proprietary game engine, using an internally developed automated testing solution. EA demonstrates that automated testing is possible within game studios, however there are EA studios all over the world and despite the existence of a central testing framework, there is no evidence that the same levels of automated testing, if any, are applied in all EA studios for all games. Further knowledge can be gained from other examples of research into automated testing in games. These studies are limited in scope, testing only one or two features, and not necessarily graphics-related nor cross-platform, but they offer valuable insight into automated testing techniques and their applications in game development.

Ostrowski and Aroudj (2013) proposed a testing model designed to create and execute fully automated regression tests for game GUI elements. Their model combined record-and-playback techniques with tests written in a game-specific scripting language to help testers create tests using basic programming skills and a GUI. User interactions, such as a tester’s mouse and keyboard events, were recorded. Screenshots were generated to represent the current state of the game, then compared to existing picture templates which represented previous user interactions. A network server was created, which accepted connections from the game being tested, then delivered messages that documented the game’s state. This testing model was utilised during the development of *Anno 2070 - Deep Ocean* (Blue Byte & Related Designs, 2012). Every build of the game caused the test suite to execute and the results were emailed to its developers. The test model was then utilised in the production of *Might and*

*Magic Heroes Online* (Blue Byte & Related Designs, 2014). It allowed testers without programming experience to create, execute and maintain automated tests from a GUI. Additionally, in *Might and Magic Heroes Online* several tests replaced the human player with artificial intelligence, allowing for automated battles to be played.

Varvaressos, Lavoie, Massé, Gaboury, and Hallé (2014) presented a runtime monitoring system as an alternative to traditional testing. Runtime verification can observe the sequence of events generated, and compare the resulting events to a formal specification to detect potential violations. The authors showed that this method could speed up the testing phase of a video game under development by automating the testing of bugs when the game was being played. To implement their system, the game loop source code was modified to add instrumentation and generated events based upon the game's internal state. The monitor was implemented as a separate application process to the game. The authors reported they successfully instrumented and effectively monitored various temporal properties over the game's execution. Manual testers were still used to play the game and as the game was executed, snapshots of game state were produced.

Lewis and Whitehead (2011a) described *Mayet*, a system for monitoring game software at runtime using instrumentation, which evaluates the state of a program as it executes, checking for invalid states. If Mayet detects an erroneous state, it instructs the game to repair itself. When the system does not behave as expected, a failure occurs. The authors provided an example from the game *Crysis* (Crytek, 2007), where a player fell through the floor of the virtual world, attempted to return to the starting point to continue playing the level, however, found the path blocked by an in-game object. It was unclear why this failure occurred in the game, and the player had to reload a previously saved game to continue playing. Instead of trying to find the cause of the failure, Lewis and Whitehead (2011a) focused on trying to solve the problem another way. They stated that problems in emergent software had more possible causes than potential solutions and therefore it could be easier to focus on fixing the problem than trying to discover the cause. The Mayet architecture facilitates this detection and repair.

Using Mayet, a game being tested must be modified to include instrumentation for monitoring game events, in a similar way to how a programmer would use debug statements to trace program flow. Instrumented game events are sent to a separate, asynchronous program, which has a rule engine that evaluates the messages to see if they break human-authored game integrity rules. Similar to writing unit tests, an experienced programmer is needed to write the rules that identify game failure scenarios. The programmer also defines how the scenarios can be repaired. The authors kept the instrumented game separate from the rule engine to decouple game implementation from testing implementation, as this guarantees language, implementation, and execution independence. The authors demonstrated

testing with Mayet using a game of their own creation in which they instrumented the source code and added intentional game faults. In their game, a bug was created to cause a game character to fall off the screen. They instrumented the game by sending the character’s location to Mayet once per second. Mayet then performed boundary checks, and if the character was deemed to be outside the virtual world boundary, Mayet sent a character kill message to the game. To conduct repairs, the game was also modified to listen for Mayet messages as part of its game loop and was able to choose when a repair should occur. The authors identified that as Mayet focuses on repairing game state, its architecture is limited (Lewis & Whitehead, 2011a). Mayet is not capable of detecting or repairing graphical pipeline bugs and Lewis and Whitehead (2011a) suggested the need for further work in this area.

Ostrowski and Aroudj (2013) focused on automated GUI testing, with tests written and executed by non-programmers, in contrast with the broader range of automated testing at Unity Technologies and EA which required highly-skilled programmers to write and maintain tests. Lewis and Whitehead (2011a) and Varvaressos et al. (2014) both explored runtime monitoring systems which required the instrumentation of game source code. Neither runtime system focused on graphics-related defect detection, and the GUI-related work was limited to the testing of graphical user interface elements. However, the techniques used in these studies can be used to inform future work, and demonstrate some cross-over with techniques, including the use of record-and-playback systems and image comparison, being utilised on a larger scale at Unity Technologies and EA. The proprietary Sony *PlayStation 4* development environment also has the ability to capture controller input and then replay the events (Schertenleib, 2013). Dickinson (2001) described the design of an instant replay system for game engine development, which allows bugs to be reproduced by isolating all sources of data input into the game engine, capturing these inputs and replaying them in the same sequence. Building the core of a game engine in this way would benefit automated testing of graphics, allowing real-time testing scenarios to be captured and replayed.

Nantes et al. (2008) proposed a general software framework that could inspect a video game using a combination of artificial intelligence and computer vision to support a game testing team by accelerating and improving the testing process by visually detecting bugs using perception similar to that of a human player. The proposed testing system was abstract and independent from the game software that it would need to test. The authors presented a limited prototype which could detect one graphical defect — shadow-aliasing artifact issues — to support the effectiveness of the framework. The prototype removed all non-shadow objects from an image prior to a computer vision algorithm processing the image, only allowing the algorithm access to evaluate potentially defective shadows. The results showed that it was possible for the algorithm to detect jagged shadow edges in the images. Nantes et



al. (2008) suggested that future work could include the development of automated regression testing tools that integrate debugging features, monitoring of user activity and replication of user behaviour in subsequent tests.

## 2.4 Automated Testing in Related Fields

While there is limited availability of details of automated testing for computer graphics within the game industry, and games-related research investigating automated testing in non-graphical scenarios is narrow in scope, it is possible to draw further insight into automated testing from studies in computer graphics-related domains.

*Graphical Kernel System* (GKS) was designed for low-level computer graphics drawing, providing basic 2D vector graphics rendering capability. Pfaff (1984) introduced an early example of computer graphics testing with a scheme for the conformance testing of GKS implementations based upon functional blackbox testing, where a GKS implementation could be checked to ensure that it produced suitable picture output on a graphical device. The GKS interface defined around 200 functions, which had complex effects on internal data structures in environment-dependent ways. The author identified that most data output by the graphics system was not accessible by the client application through the graphics interface, and hence it could not be tested from within program source code. GKS output could be observed in the form of rendered pictures, data files or interaction sequences. The combination of blackbox interface and the number of functions, made verifying the correctness of a GKS implementation difficult.

Pfaff (1984) proposed a comparison model, using a suite of test programs that could be utilised with an implementation under test. Firstly, the test suite was executed with a reference system to generate known reference results, which were then sent to the comparator. The implementation being tested produced output results that were also sent to the comparator to check for conformance with the reference results. The author's comparator operation was conducted by a human operator. This technique was error-prone, as the human operator had to evaluate the results using their subjective impression of comparison, they potentially lacked the ability to check the preciseness of the graphical images visually, and they could be overloaded with large amounts of data to compare. In contrast to using a human comparator, Pfaff (1984) presented a defined data format that described picture elements which were produced each time a visual output occurred on a graphics device, and a comparator that was designed to accept these picture element descriptions. Test programs were run on two implementations of the GKS system using the same graphics device, and the reference results were compared with the results produced. Since GKS produced basic 2D vector graphics output it would

have been feasible to compare image output using textual picture element descriptions.

At SIGGRAPH 96, the *Virtual Reality Modeling Language* (VRML) community met with staff of the National Institute of Standards and Technology (NIST) to discuss approaches to testing the VRML standard (Brady et al., 1999). VRML virtual worlds had to be compliant with this standard, regardless of the hardware or software platforms used. Following the discussion, NIST developed tools to support the VRML testing process.

Stone (1999) noted that making VRML worlds look the same on all target platforms was difficult due to different software and hardware systems, and identified colouring and shading as one of the greatest challenges. This was attributed to variation in the implementation of lighting and shading models, as well as triangulation and interpolation algorithms. These algorithms were not implemented by the VRML platform, but were instead implemented by APIs, such as OpenGL and Direct3D, which were utilised by VRML to render the virtual environment (Stone, 1999). VRML browsers were tested using a group of conformance files known as the VRML Test Suite (VTS), where firstly a parser was used to check VRML files for validity, then conformance tests, using VRML test worlds, were run. These tests were known as the VTS Method (Brady et al., 1999). Brady et al. (1999) stated that test cases needed to be defined to test combinations of features. For example, to create adequate coverage of a lighting model, a test utilised a combination of unlit, greyscale, RGB, and RGB-and-Alpha properties that were applied to the test world.

Expected results were captured as JPEG for static scenes, and MPEG for animated scenes. Links to the original VRML scenes were retained, with the ability to do a side-by-side comparison of the test case and the expected result (Brady et al., 1999). In testing, three different browsers successfully passed 90% of the tests. This information was then shared with the developers of the browsers, allowing for further bugs to be uncovered and fixed. This created an increased level of confidence that a VRML world could be rendered similarly across various hardware and software platforms, resulting in positive user experiences. The author's testing results also uncovered conditions where various implementations differed, which led to the development of explicit tests that focused on those particular differences. Overall, this served as a way to raise awareness in the VRML community and helped to refine the VRML standard specification.

Lorensen and Miller (2001) described automated testing of the *Visualization Toolkit* (VTK), an open source C++ library for visualisation and imaging algorithms. Development was started in 1993, and targeted the Unix, Linux and Windows platforms. The developers recognised the need for regression testing to identify changes in the software that affected its output. VTK regression testing compared images generated by VTK with developer-validated baseline images. The image comparison routine was able to control how well the images matched, noting that for example, OpenGL output

did not need to match pixel for pixel, whereas the imaging algorithms had to strictly match. Until 1998, regression tests were run manually and adhoc prior to a major release of VTK. As this testing was infrequent, it was difficult to determine which of the hundreds of development changes had caused the differences between generated and baseline images. The development team then increased the number of regression tests, and introduced more frequent, and automated, testing. Test scripts were run nightly on eleven different configurations of operating system and hardware. The combinations of operating systems and compilers helped to ensure VTK remained portable every day. Test results were logged for each platform, and for each failed test, a JPEG was generated alongside the expected image and a difference image of the two. The tests were also timed on each platform, and a summary of faster or slower tests showed variation in timing over the last ten days and the last twelve weeks. An HTML dashboard summarised the results. A programming style-checker was also run, and documentation was generated from the C++ header files. A downloadable installation executable was created for each of the target platforms. As developers added new code to the library, they were expected to add new tests to the regression suite to cover their new code additions (Lorensen & Miller, 2001).

The VTK development team conducted meetings at the start of each day to review the nightly dashboard report, and simple errors were identified and fixed immediately. More complex problems were assigned to developers to fix. The automated nightly tests allowed the continued rapid development of VTK, while maintaining a high quality, robust toolkit. In 1999, a continuous build and test system was added. This system monitored the source code repository for changes, and upon a change, compiled VTK on one combination of hardware and operating system. It then ran four smoke tests that conducted minimal functionality testing and test failure notifications were emailed to the developers. This resulted in defects being uncovered as soon as they were introduced. Lorensen and Miller (2001) noted that there was a significant limitation to the VTK automation environment as it was highly coupled to VTK, and therefore difficult to replicate outside of the VTK development environment.

The GKS, VRML and VTK examples demonstrate the benefits of the application of automated testing techniques to computer graphics testing. Image comparison methods are used to uncover defects introduced through development changes in cross-platform development targets. In GKS, image comparison required a human to conduct result comparison and this was found to be error-prone. However, the low complexity of the GKS renders allowed for textual descriptions to be generated and compared. The use of VRML test worlds showed that defects could be detected across various implementations, much like the test scenes used in the Unity engine’s automated testing, while VTK’s description of how difference images can be used for enhanced comparison and defect detection is similar to the work shown by Bay (2016) at Unity Technologies. Both VTK and Unity are systems

highly coupled to their specific technological implementations.

Fell (2001) proposed a theoretical embedded graphical application to explore the principles of automated regression testing of interactive graphical software. The graphical application's main output would be node-based graphs, which the user could interact with in real-time to change the layout of the nodes. The application would be run in full-screen mode and would feature an event loop where data was input into the program via a network connection. Importantly, testing would require simulated user input, and the screen output would have to be checked for correctness. He proposed that the application be developed on workstations and that the portability layer be implemented for Unix and Windows, which would allow the application and its user interface to be tested on the more powerful workstations rather than the target embedded platforms. Failures in the application's design could occur if a developer incorrectly implemented the portability layer or if there were faults in the tool chain or hardware. The cross-platform targets and the interactive style of the application meant there would be no clear end point. Interactive debugging could be performed through a network connection using a server embedded within the application and a command line interpreter would be used to dispatch messages which would allow user input events to be inserted into the portability layer of the application. The application would require human operator verification.

The author suggested the following test result states: *Never*, meaning the test has never been performed; *passed*, no fault was found; *failed*, one or more faults found; and, *broken*, the test could not be performed. Each test would have a unique integer identifier and name and a database could be used to store a record of test results, where entries would include when the test was first added, last passed, and last failed. The test harness would have core functionality such as: initialise, register, report, test, retire and verify. Nightly regression testing would be executed, and the results of failed and broken tests would be emailed to the development and QA teams. As new features were added to the application during development, new tests would also be added (Fell, 2001). Basic testing would involve sending data to the application, then checking to see if the application had computed and presented the output. Network debugging would allow the current screen to be captured to file. A human operator, such as an experienced programmer, would then examine the graphical output and judge the results. Subsequent executions of tests would generate new screen captures to be compared against previous captures. If the results were the same, the test would succeed; if different, the test would fail. A human operator would again be required to conduct this comparison. If the new result was determined to be correct, the reference screen image would be updated to this new result (Fell, 2001).

Bierbaum, Hartling, and Cruz-Neira (2003) described a technique for supporting the automated testing of the interactive aspects of virtual reality applications. They argued that there is no common

pattern for this kind of testing. The authors proposed a test architecture which would allow developers to test an entire code base rapidly and reliably, and presented an implementation that allowed virtual reality developers to use automated testing techniques. They stated that to test high-level user interaction of a virtual reality application is difficult, as there is no way to directly test the interaction code. Automated testing utilised in desktop GUI interaction cannot be used because the virtual reality application directly handles a user interacting with a virtual environment, and it may not be possible to programmatically deliver this input to the application. The authors suggested that to automate the testing of virtual reality applications, test cases must be able to test the interaction code.

In the proposed test architecture, the application state can be tested for correctness only once all input has been processed by the interaction code. This method of testing requires the state of input processing to be monitored. The virtual reality test monitor must wait for a checkpoint to be reached, then it can verify that the application is in the correct state. To automate testing, the application must be provided with input simulation which mimics direct user interaction. Input is recorded during an application usage scenario, with user-definable checkpoints. During testing, the application is controlled by the prerecorded input data and when a checkpoint is reached, the test case verifies that the state of the application matches the previously recorded state. The initial state of the application must be guaranteed at the start of the test. Checkpoints are arrived at sequentially, and states can build on each other in sequence. The test runner is responsible for tracking the results and if a match is incorrect, a failure is identified (Bierbaum et al., 2003).

Similar to the testing frameworks investigated by Varvaressos et al. (2014) and Lewis and Whitehead (2011a), Fell (2001) and Bierbaum et al. (2003) proposed theoretical testing applications that utilised event generation and record-and-playback techniques, and again, image comparison is utilised to verify the correctness of test results.

Yee and Newman (2004) described an image comparison process that can help to decide if images appear visually identical, even when they contain differences. They stated that this had been useful in the testing of rendering software for motion picture production. The movie production pipeline depends on software that functions correctly, is stable and bug-free and this means that rigorous software testing is required. The pipeline used in the authors' studio had a proprietary renderer, with many graphics shaders that were written in C and C++, and testing was conducted by nightly automated batch processing. Tests required the re-rendering of test scenes to ensure that no bugs had been introduced during development, and images from the test scenes were then compared with previously generated reference images. The authors stated that pixel-by-pixel comparison between reference and test images did not always work. Source code changes to the rendering software could result in pixel intensity changes that produced pixels that were not numerically identical, however

there could be no visible difference between the images. Yee and Newman (2004) identified sampling as a major source of this issue, but also identified anti-aliasing and shadowing as potential causes.

By using a perceptually-based error metric, false positives could be avoided when using the test suite. This error metric attempted to mimic the human visual system when analysing images, and only alerted to failures that could be identified by the human eye. As the authors dealt with the comparison of thousands of film-resolution images during testing, they utilised an abridged version of a Visible Differences Predictor (VDP) to increase the overall speed of the algorithm (Yee & Newman, 2004). An example test case rendered a scene with a basic object with various lighting conditions and a reference image was created from the rendered frame. If there were any source code changes in the renderer, a new test scene image was generated which was then compared against the known reference image. This process allowed the authors to discover many bugs. They found that in the continued development of shaders to create faster optimisation techniques, subsequent tests would produce images that were not pixel-identical with the reference image, but would still be considered correct. The perceptual metric allowed these code changes to occur without alerting to a failure in the automated testing process. The testing suite produced a mosaic of difference images for every test that failed. The QA group was able to quickly identify problems using this overview image.

Yee and Newman (2004) provided production of the film *Shrek 2* (Warner, A. & Williams, J. & Lipman, D. (Producers) & Adamson, A. & Asbury, K. & Vernon, C. (Directors), 2004) as an example, where a new compiler was introduced to the tool chain. Applying the perceptual metric with their automated test suite helped the film-makers identify shaders that had become broken by the tool chain update. This testing was essential in helping them switch to the new compiler, and it assured the team that their existing film shots would not break. Another example was when the film studio switched to a new operating system and the test suite contributed to the identification of existing tools that had either become inoperable, or behaved differently, on the new platform.

Yee and Newman (2004) highlight an important consideration when using image comparison to compare test images in automated testing. While a human comparator is slower, the advantage is that they can make judgements as to whether an image passes or fails. When this comparison is automated, pixel-by-pixel comparison can be considered too strict, as it will fail images that have any pixel differences, even if these are not visible to the human eye. By using an image comparison technique that mimics the human visual system, these false positives can be avoided, and only images with visible differences will be flagged as failed results.

Grønbæk and Horn (2008) investigated the elements needed for an automated testing tool for virtual environments used in military simulations, networked scenarios, game engines, and scene graph technology. As the correctness of the graphically simulated environments is verified using manual tests

with human testers, the authors identified the need for cheaper, faster and more accurate testing and suggested that there was a lack of testing culture in simulation development.

They discussed image matching, scene graph matching, and video matching as possible graphical testing techniques. They identified that one of the difficulties in graphical testing is that it is unclear what scenes should be collected as references. Additionally, in the case of games, non-deterministic and random aspects which impact graphical simulation, such as weather effects, make it hard to perform comparisons against known references. User interaction is also non-deterministic. They therefore suggested that reference scenes should only be coupled to deterministic behaviour and that non-deterministic behaviour and user interaction should be avoided (Grønbæk & Horn, 2008). The study concluded that in order to realise a fully automated testing tool it is necessary to implement a working prototype. They noted that virtual environments can be very large, and it may be impossible to cover the whole environment with graphical tests to discover defects. They proposed using regression testing, where first, a human tester could locate a defect and report it to the development team to repair and then a test case would be created to ensure that the bug does not reoccur.

Grønbæk and Horn (2009) later presented a process for testing graphical virtual environments. They designed and implemented a test tool to confirm the validity of the proposed process. Tests were conducted on a selection of sample applications developed using the third-party Delta3D engine, however they designed their system for use in any graphical environment. The testing process was evaluated based upon these test results and focused on three core areas. First, the project implemented image processing and analysis methods to compare test scene outputs, and evaluated the value of image comparison methods when used to compare 3D scenes. Second, an evaluation of the feasibility of scene graph comparison in the context of graphical regression testing was conducted. Finally, a set of tools for automation of graphical regression testing was designed and implemented, and combined with existing automated testing tools.

Their testing procedure was as follows:

- *Generate test input*: human testers create test cases by interacting with the application, low-level input to the application captured and stored;
- *Generate test output*: screenshots from the applications captured and stored;
- *Test case selection*: from a list of tests;
- *Test case execution*: a replay is performed using the selected test case, repeating the previously captured user interaction sequence;
- *Comparison of outputs*: a tool presents the generated output images side by side in an HTML document, and a human tester makes the comparison;

- *Failure identification*: an image tool is used to help identify the differences between the two images, as well as reporting related image metrics.

This resulted in the creation of a collection of test support tools, including a semi-automated image collector which required human intervention to capture scene frames. The image collector relied on the judgement of a human tester to determine whether captured frames matched the reference and an image analyser analysed scenes for degree of similarity, using pixel-by-pixel comparison. A test reporter then showed the collected and reference images. The system was tested with a Delta3D application called *HoverTank*. The resulting image comparison was not completely identical in terms of pixel-by-pixel comparison and this was attributed to inaccuracies in the replay timing, and z-buffer fighting. Grønbæk and Horn (2009) noted that human inspection would have considered these images to be identical. They also noted that the length of time taken to generate the test capture corresponded to the length of time taken to re-run the replay when executing the test. The authors believed that the comparison based upon images collected from rendered scenes was not the best approach, as information was lost in the process of projecting a virtual 3D world into a 2D representation. However, they decided not to pursue a scene graph matching technique due to the use of proprietary and closed formats that were incompatible with the middleware they used.

They created a separate tool using the Image Processing Library 98 (IPL98), which features C++ image container classes and image processing algorithms. Additional image metric algorithms were implemented to compute image comparisons using the following methods: *Mean Square*, *Root Mean Square*, *Structure Content*, *Cross Correlation*, *Angle Moment*, and *Czenakowski Distance* (Grønbæk & Horn, 2009). Five tests were conducted, based upon a 3D scene with an animated walking soldier in different configurations. The scene was tested with and without the soldier's weapon, helmet, and the soldier. Simulations were run, and the resulting screenshots were compared with the tool, as well as a generated difference image, which was a subtraction of the two images. They concluded from these tests and the resulting metrics that a programmer or tester would find it difficult to tell if a test should pass or fail based upon the metric calculations alone. The authors suggested that more testing was required to fully evaluate the image analyser tool. Grønbæk and Horn (2009) identified that advancements were required in the image matching method for the image analyser, and that there should be a reduction in the amount of manual work performed, especially when human judgement is required. Like Yee and Newman (2004), Grønbæk and Horn (2009) suggested that metrics which take the human visual system into account should be utilised. The reporting tool should include a test history feature, image scaling, test comparisons and thresholds, as well as better textual descriptions.

Timofeitchik and Nagy (2012) presented a conceptual model for automating the testing of real-time graphics systems for television. The purpose was to increase the probability of finding defects by



making verification more efficient and reliable. They utilised the *Structural Similarity Index* (SSIM) for image comparison in the creation of their automated testing tool, known as the *Runtime Graphics Verification Framework* (RUGVEF). SSIM was developed by Wang, Bovik, Sheikh, and Simoncelli (2004) as an image quality assessment framework that corresponds to the human visual system. Other algorithms such as *Mean Squared Error* (MSE) and *Peak Signal-to-Noise Ratio* (PSNR) are simple to calculate and have a clear meaning, but SSIM is more aligned with human perceived visual quality, and was found to exhibit better consistency with qualitative visual appearance than the other models (Wang et al., 2004).

In agreement with Grønbæk and Horn (2008), Timofeitchik and Nagy (2012) stated that traditional testing techniques were insufficient for real-time graphics testing and identified that non-deterministic behaviour and time-based execution make errors difficult to detect and reproduce. Unit testing did not take into account platform characteristics such as the underlying hardware, operating system, device drivers and external graphics APIs, and this could mean that graphical issues were not detected during development (Timofeitchik & Nagy, 2012). They noted that human visual inspection is commonly used to detect errors in real-time graphical software and this makes regression testing impractical and error-prone, as it relies upon human subjectivity. This led to the author’s development of their automated testing tool. Their research was conducted using the Design Science Research Method, with the purpose of creating an information technology artefact to address an important organisational problem. The artefact was evaluated in the context of a case study.

RUGVEF determined the correctness of graphical output through continuous analysis during the execution of video playing software. Objective comparisons were made against existing reference images by a monitor application that ran in parallel to the video player. The testing framework was implemented during the live development of *CasparCG* (casparcg.com, n.d.), a tool used to create real-time television overlay graphics. CasparCG was incrementally developed and tested fortnightly with code reviews and human visual inspections (Timofeitchik & Nagy, 2012). The authors identified that the CasparCG developers rarely used regression testing, which limited refactoring and the ability to test software fixes. Bug reports from prior public releases of CasparCG detailed the late discovery of defects in the software. RUGVEF added overhead which could reduce overall system performance and affect time-sensitive systems, so to counteract this they investigated image quality assessment methods. They noted that pixel-by-pixel comparison did not take into account the human visual system and there was a possibility that small differences in images could be acceptable. Because of SSIM’s focus on modelling human perception, it was selected as the comparison method for RUGVEF. The authors optimised the implementation of SSIM to allow the algorithm to operate in real-time to analyse full high-definition video output. Improvements to the algorithm included using Single Instruction, Multiple Data (SIMD)

instructions, as well as cache-friendly optimisations. Multi-core architecture was also applied to the algorithm, and the chrominance information from the SSIM calculation was discarded so that only luminance information was utilised. These optimisations allowed the modified algorithm to operate in real-time on consumer-level hardware.

The results of the case study showed that their software was able to detect five previously unknown defects within CasparCG (Timofeitchik & Nagy, 2012). These included minor pixel errors, as well as missing or skipped frames, tinted colours due to incorrect colour transformations, and artifacts that occurred due to rounding errors with alpha blending. The software was also able to detect six out of sixteen known defects which had been deliberately inserted back into RUGVEF. This showed that the automated system could augment manual testing techniques to help discover contextual and temporal defects which could lead to earlier detection and improved development outcomes. The authors identified that the testing of real-time graphics for games remains unexplored, suggesting future research in the field of computer graphics in games.

## 2.5 Findings of Literature Review

The Lewis et al. (2010) taxonomy demonstrates that there is awareness of the existence of graphical defects in commercially released games. There is a demonstrated lack of automated testing in games, and a reluctance to adopt this practice, despite the reported advantages. However, there is also support in the game industry for change. While Bierbaum et al. (2003) argued that there was no clear pattern for automated testing, the literature shows the emergence of common practices, such as the use of small test scenes that demonstrate different graphical features, and screenshot capture and comparison to known reference images. Practice has evolved to use image comparison methods such as pixel-by-pixel comparison and later studies have investigated comparison methods that better mimic the human visual system, in order to increase the speed and efficiency of comparison and avoid false positives.

In game scenarios, an input capture and replay system can aid in test reproducibility, as it records the user inputs to show what the user was doing when the defect occurred. However, this requires access to the game's source code to add additional instructions to accommodate record-and-playback functionality. Randomness and non-deterministic behaviour in games must also be taken into account, since to automate testing requires some knowledge of the expected output, and tests may execute differently each time a testcase is run.

Cross-platform development issues occur when the OS, GPU, device driver, compiler, API and graphics shaders combine to create complexity. This can create a complex testing matrix. Systems have been developed that can cope with the added complexity of cross-platform development, though

there is no evidence of this in the literature related to graphics defect detection in game development. Many of the systems described are highly coupled to their target application and therefore are not transposable to other development scenarios or applications. Developing a system that is abstract is key to widening the availability and adoption of automated graphics testing in the game industry.

Due to the competitive nature of the game industry, it is difficult to ascertain to what extent automated testing is used, and if so, how it is implemented and for what kinds of testing. Research into automated testing in related computer graphics fields has yielded interesting details of testing frameworks that could be applied to testing in game development. Despite the evidence for a common pattern of successfully implemented automated testing techniques, there is no evidence of a system that is able to automate the testing and validation of computer graphics implementations for cross-platform game development. This supports the need for research in this area, and in the following chapters I will discuss the exploration of my research question through the construction and evaluation of several artefacts.

## Chapter 3

# Methodological Approach

“Research is a creative process”

---

Vijay K. Vaishnavi and William  
Kuechler Jr., Design Science  
Research Methods and Patterns  
Innovating Information and  
Communication Technology  
(Vaishnavi & Kuechler, 2008, p.  
121)

### 3.1 Introduction

The term *methodology* refers to a framework used to conduct research and the term *methodological design* refers to the plan for conducting the research, including the methodology, methods and tools (O’Leary, 2004). The important prerequisites in selection of the methodological design are to ensure that firstly, the design addresses the research question, secondly, the design suits the researcher, and finally, the research task is achievable through adequate provision of time, resources, and ethical considerations (O’Leary, 2004). This chapter addresses the selection of the research methodology, and the associated methodological design utilised in conducting this research project. I will discuss why Design Science Research was selected and outline the key components of this methodology, including the process steps and the patterns that were applied at each phase. I will then discuss the development tools utilised in the creation of artefacts and the limitations that were considered in the design of the research methods and the undertaking of the research project.

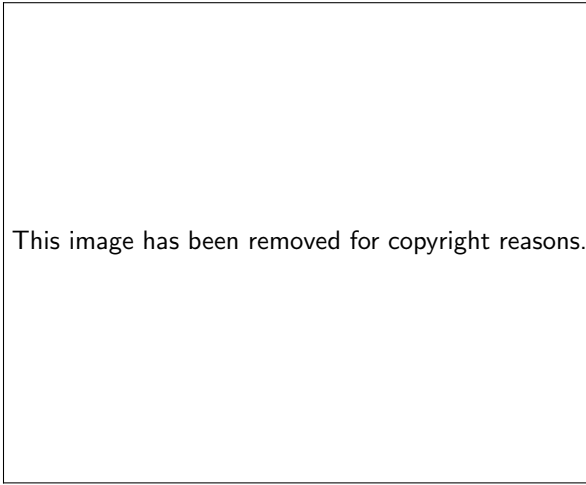
## 3.2 Methodology: Design Science Research

With its origins in engineering and the sciences of the artificial (compared to natural sciences), Design Science Research methodology (DSR) is considered a problem solving paradigm, where important unsolved problems can be addressed in unique or innovative ways, or solved problems can be revisited to find more efficient solutions (Hevner, March, Park, & Ram, 2004; Wieringa, 2014). Vaishnavi and Kuechler (2008, 2013) describe DSR as a way to learn through building, hence providing the framework for a research project that creates an artefact to solve a problem and create new knowledge. Artefact construction allows design to be used as a research method (Vaishnavi & Kuechler, 2013). A key consideration is that where research is described as an activity that contributes to the understanding of a phenomenon, with DSR, all or part of the phenomenon may be created. Design refers to the creation of something that does not yet exist, and so DSR contributes to a body of knowledge about artificial, human-made phenomena and artefacts that have been designed to achieve particular goals (Vaishnavi & Kuechler, 2008). Artefacts include, but are not limited to, processes, human-computer interfaces, algorithms, system design methodologies and languages (Vaishnavi & Kuechler, 2013). Resulting constructs, techniques and methods can contribute new knowledge, or fill gaps in existing knowledge (Vaishnavi & Kuechler, 2013). The creation, design and evaluation of artefacts is an iterative process, whereby the researcher attempts to answer technological problems that are important to an organisation or research community (Hevner et al., 2004; Vaishnavi & Kuechler, 2008; Wieringa, 2014). DSR researchers must analyse the use and performance of artefacts, evaluating and reflecting upon results that can then be incorporated into further iterations (Vaishnavi & Kuechler, 2013).

DSR differs from design, as it produces new knowledge and requires intellectual risk to solve “we don’t know how to do this yet” problems (Vaishnavi & Kuechler, 2008, p.26). There are often a number of unknowns in the design of a proposed solution to a DSR problem, and the researcher can be unaware of what the final outcome will be (Vaishnavi & Kuechler, 2013). As such, technological advances can be the result of creative and innovative DSR processes (Hevner et al., 2004). The creation of theoretical knowledge does not necessarily occur in a single DSR project, it can instead be created through many cycles of research and development and often includes active industry participation (Vaishnavi & Kuechler, 2013). An activity framework for DSR can be seen in Figure 3.1.

March and Smith (1995) suggest there are two processes and four outputs for DSR. The two processes are *build* and *evaluate*, and the outputs are summarised as:

- *Constructs*: Constructs are the vocabulary of the problem/solution domain. They can arise during the conceptualisation of the problem and are refined during the design cycle. Hevner et al. (2004) states that vocabulary and symbols provide the language with which to describe



This image has been removed for copyright reasons.

Figure 3.1: Activity framework for Design Science Research. Adapted from “On theory development in design science research: Anatomy of a research project” by B. Kuechler and V. Vaishnavi, *European Journal of Information Systems*, Volume 17, Issue 5, pp. 489-504. Copyright 2008 by Palgrave Macmillan.

problems and solutions;

- *Models*: Models can be described as constructs used to represent a real world situation (Hevner et al., 2004), and are proposals for how things are, presented in terms of what the model does;
- *Methods*: Methods are a set of steps used to perform tasks. These can be algorithms and practices that define the processes needed to realise a model (Hevner et al., 2004);
- *Instantiations*: Instantiations are the realisation of the artefact in an environment, including implemented systems and prototypes (Hevner et al., 2004). This output can come before the articulation of models and methods.

March and Smith (1995) define DSR as an attempt to create technology-oriented artefacts for human-defined purposes, and therefore the products of DSR projects should be evaluated against two criteria, value and utility — “does it work?” and “is it an improvement?” (March & Smith, 1995, p.253)

Hevner et al. (2004, p.75) state, “knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact” and hence derive and propose seven guidelines for DSR:

1. *Design as an artefact*: An innovative and purposeful artefact is created;
2. *Problem relevance*: The artefact is relevant for a specific problem domain;
3. *Design evaluation*: The artefact is thoroughly evaluated;

4. *Research contributions*: An unsolved problem is solved, or a solved problem is solved in a more effective way;
5. *Research rigour*: The artefact is rigorously designed and formally presented;
6. *Design as a search process*: The process by which the artefact is created provides an effective solution;
7. *Communication of research*: The results are communicated effectively.

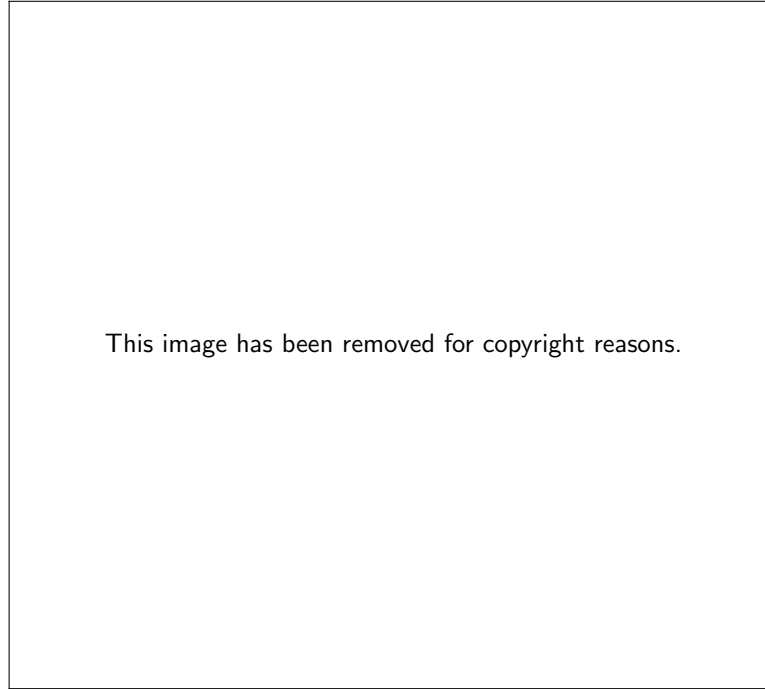


Figure 3.2: Design Science Research Process Model. Adapted from “Design science research in information sciences” by V. Vaishnavi and B. Kuechler, Copyright 2013.

Vaishnavi and Kuechler (2008) further develop the work of March and Smith (1995) and Hevner et al. (2004) into the DSR methodology used in this research project. The methodology consists of phases, as seen in Figure 3.2: Awareness of Problem, Suggestion, Development, Evaluation and Conclusion. These phases map to the research proposal, literature review, data collection, analysis, and publication stages of a research project. The research approach is iterative, allowing for cyclic research activity to occur through circumscription, and for the refinement of the research questions to occur as part of the knowledge discovery process.

DSR was an appropriate choice of methodology for this research project. Artefact construction is part of the computer science research community paradigm, and the knowledge base has been built over time through the construction and evaluation of artefacts (Vaishnavi & Kuechler, 2008). Engineering disciplines also consider design as a valid research methodology, as the community’s research culture

values “incrementally effective applicable problem solutions” (Peppers, Tuunanen, Rothenberger, & Chatterjee, 2007, p.1) and the practical application of these research outcomes. Hevner et al. (2004) aligns DSR processes with industrial practice, and suggests they can be utilised for collaborative industrial and academic research projects. DSR project results can be used to inform technology-focused organisations and the management of these organisations, and to develop technology-based solutions to business problems.

In this research project, there were many components that needed to be brought together, and therefore the methodology needed to be flexible enough to allow for experimentation and the construction of multiple artefacts to test these different components. It was not clear if the research question could be answered, introducing uncertainty to the project. Hevner et al. (2004) states that it is exactly these types of problems, characterised by unstable requirements or constraints, or with complex interactions between subcomponents, that DSR is suited to, since its inherent flexibility allows the design process or artefacts to change during the development and evaluation phases, and there is a need for creativity to produce an effective solution. Prior related research by Timofeitchik and Nagy (2012), as discussed in Section 2.4, is an example of the DSR methodology being used to create an artefact which could automate graphics testing for broadcast television video systems.

DSR also maps to general software development methodologies, as highlighted by March and Smith (1995), who state that a software product is dynamic and is developed incrementally, with functionality able to be added as needed. Game development industry methodologies are similar, being iterative and incremental, as well as agile. For game developers, iteration refers to the practice of creating a version of source code or artwork, for example, examining or evaluating it, and then revising it until it is sufficiently improved (Keith, 2010). These iterative methodologies are proactive rather than reactive, and allow time for planning, as well as the flexibility to change course during development if necessary. The fundamental idea is to develop for a set period of time to accomplish predetermined goals, get the game into an acceptable state, and then start a new development phase to add new features (Rabin, 2010). These multiple production cycles can be mapped to the DSR phases. Preproduction is the Awareness of Problem and Suggestion phases, production equates to Development, testing to Evaluation and wrap-up to the Conclusion phase. When a cycle concludes, the development team moves on to the next cycle: first playable prototype, Alpha version, Beta version, and Gold (Chandler, 2014). The difference between DSR and game development agile methodologies is the knowledge creation; DSR seeks to create new knowledge through artefact creation and evaluation, while game development methodologies seek to create a fun, playable game.



### 3.2.1 Patterns

Vaishnavi and Kuechler (2008) describe patterns that can be applied to each phase of a DSR project. These patterns are a formalised way of recording experiences. Problems will be encountered during a research project, and the purpose of the patterns is to aid the researcher in focusing their thoughts so that the issues can be overcome (Vaishnavi & Kuechler, 2008). The patterns can be used to communicate contextually rich information common to fields such as computer and software engineering.

Ahmed and Sundaram (2011) highlight the concerns of many scholars who question the validity of DSR and the evaluation of artefacts generated because of a lack of rigour in the evaluation process. However, using the evaluation patterns suggested by Vaishnavi and Kuechler (2008) will enhance the validity of the DSR evaluation process. Evaluation is an important step in analysing whether or not artefacts are fit for purpose and allows a researcher to establish how well the research has addressed the problem it intended to solve (Ahmed & Sundaram, 2011). Unlike traditional research approaches, where the end product is evaluated, DSR integrates evaluation at each step of the research process (Ahmed & Sundaram, 2011).

The patterns for each phase will be discussed in more detail in the following section.

## 3.3 Method: Research Project Design

This section describes the application of the DSR phases in this research project, along with the associated patterns utilised during each phase.

### 3.3.1 Awareness of Problem

In this phase, the problem is identified and defined. *Awareness of Problem* can come from multiple sources, including new developments in industry (Vaishnavi & Kuechler, 2013) or professional literature (Wieringa, 2014). The problem can be identified through a literature review, through experience in practice or from a simple conversation about industry (Vaishnavi & Kuechler, 2008). A literature review can determine whether the problem has already been solved and if any previous related research has been undertaken, as well as determining how widespread the problem is, and if the solution will be interesting to the research community (Vaishnavi & Kuechler, 2008). The output of this phase is the proposal for new research (Vaishnavi & Kuechler, 2008, 2013).

Problem selection and development patterns can be applied in this phase, including *Research Domain Identification*, *Problem Area Identification*, *Problem Formulation*, *Leveraging Expertise* and *Being Visionary* (Vaishnavi & Kuechler, 2008). These patterns suggest that interest should be the primary

motivation for choosing the domain, and that the selected problem should utilise the researcher's strength and expertise or build on it. The researcher should have some understanding of the research community, and an awareness of practice and industry. These can be built through a literature search. Being Visionary requires the researcher to imagine an improvement to a situation or problem, even though the current solution might be acceptable to the industry or research community. By identifying the key features of the present situation and envisioning how the situation could be improved, the researcher is able to review the gap between the present solution and their vision for improvement (Vaishnavi & Kuechler, 2008).

The beginnings of the Awareness of Problem phase in this research project occurred through exposure to the current state of the game development industry. Online media articles detailed the release of new cross-platform AAA video game titles which featured game-breaking bugs upon launch. These articles discussed the occurrences of computer graphics problems within the game titles. This phase was discussed in more detail in Section 1.2, which outlined the motivation for this research project. Though the idea began with the internet search and discovery of numerous graphical bugs, the literature review confirmed that there was an existing problem with testing for graphical bugs, and that there was no current publicly available automated cross-platform solution.

This was developed into the research question and sub-questions found in Section 1.3 and the research proposal.

### 3.3.2 Suggestion

The *Suggestion* phase is closely related to the Awareness of Problem phase, and it is in this phase that a tentative design for solving the problem is proposed. This design can be based upon new and or existing elements and can be drawn from existing knowledge, such as academic or professional literature (Vaishnavi & Kuechler, 2008, 2013). It is a creative phase, where the researcher must formulate inventive potential solutions to the problem. It may be necessary to decompose the problem into smaller or simpler subproblems in order to begin to envision how the problem could be solved. This simplification may represent a starting point without having a significant impact on practice (Hevner et al., 2004). This could mean starting with simple technological simulations in idealised conditions, to test if something is even possible, with the aim of generating knowledge that is applicable in the real world (Wieringa, 2014).

Suggestion patterns utilised in this phase include literature search patterns, such as *Industry Practice and Awareness* where the researcher can become familiar with industry practice and academic research and *Framework Development*, which prompts the researcher to organise the literature by theme to enable the identification of knowledge gaps and the analysis of key ideas and concepts

(Vaishnavi & Kuechler, 2008). *Problem Space Tools and Techniques* and *Research Community Tools and Techniques* allow an analysis of how the research or industry community solves similar problems, along with identification of the tools and techniques applicable in the selected domain. *Using Human Roles* is another pattern that can be applied to generate ideas and concepts. This involves defining the activity that the research is targeting for automation, observing how the human tasks for performing this activity are undertaken, and then analysing the performance of the task to develop potential solutions. Creativity patterns, such as *Wild Combinations*, *Brain Storming* and *Stimulating Creativity* can also be applied in the Suggestion phase (Vaishnavi & Kuechler, 2008).

In this research project a tentative design was formed based upon the Awareness of Problem phase. After an initial literature search and review was undertaken and detailed in the research proposal, further literature review (see Chapter 2) occurred post-proposal and this also informed the tentative design. The initial suggestion was the creative imagining of an automated testing system where captures of simulated graphical video game scenes from each target platform could be made and then compared. Selection of the DSR methodology for the research project further informed the Suggestion phase, as this introduced a framework of artefact construction that could be used to explore the development of an automated testing tool.

In this project, as in the development of the Lewis et al. (2010) taxonomy, the focus is not on discovering why the graphical bugs occur and solving this problem, but was instead on how they could be identified for repair before the release of a cross-platform game. This led to the review of literature related to game industry testing practices (see Appendix A), and the specific application of the Using Human Roles pattern to discover and analyse current testing practices using humans, in order to formulate a solution to the problem of automating these roles.

The proposed tentative design was to create a computer graphics abstraction artefact with capability for supporting a simple game product, and to establish model test cases for the graphics features, such as primitive rendering, alpha blending, colour blend modes and render-target size. The abstraction would be implemented on two initial desktop platforms, Microsoft Windows and Apple OS X. The automated testing artefact created would be capable of graphics output comparison and a results reporting mechanism.

### 3.3.3 Development

During the *Development* phase, the tentative design is further developed and implemented into a real artefact through construction and iterative development (Kuechler & Vaishnavi, 2008). This iterative approach can result in multiple artefacts or iterations. Data collection occurs throughout the Development phase. The implementation may be tangible, such as software or hardware development,

or abstract, resulting in constructs, models or methods. This phase should be an attempt at implementing an artefact according to the tentative design proposed in the Suggestion phase and requires some creative effort (Vaishnavi & Kuechler, 2008). The novelty can be in the design or construction of the artefact, or both, and it is appropriate during this Development phase to use state of practice techniques to develop the solution (Vaishnavi & Kuechler, 2013).

Development patterns used in this phase include *Divide and Conquer* and *Building Blocks*, techniques for managing complexity which suggest dividing the research problem into smaller problems, the solutions to which can be combined to solve the overall problem. Each problem can be solved at the lowest decomposition, and recursively assembled until the research problem can be solved (Vaishnavi & Kuechler, 2008). *Simulation and Exploration* allows the researcher to understand and predict the behaviour of a system that cannot be built or that it is infeasible to construct. Simulation allows the creation of objects and environments that imitate real world situations so that observed behaviour can be used to predict how the system would operate in real life.

In the Development phase of this research project, two artefacts were iteratively developed simultaneously, the first of which generated simulated rendering outputs, and the second of which was an automated tool for the comparison of the rendered output. Development of these artefacts occurred independently, such that the number of iteration cycles differs for each artefact, however, the development and evaluation of each artefact was used to inform the further development of the other. The development of each artefact used a combination of existing elements, such as image comparison methods, along with industry standard cross-platform game development tools and techniques, to create a novel result.

### 3.3.4 Evaluation

Following the Development phase, the artefact goes through an *Evaluation* phase, which typically results in information that can be used to further inform another Development, or earlier phases, through circumscription, or iteration. Evaluation could take place continuously throughout the Development phase in the form of *Micro-evaluations*, which can occur whenever a design decision needs to be made (Vaishnavi & Kuechler, 2013). The DSR process generally cycles between Awareness of Problem, Suggestion, Development and Evaluation phases, as opposed to traditional Waterfall approach where a completed phase flows directly onto the next with no opportunity for iteration. This is an important difference between DSR and other research methodologies where evaluation usually comes at the end outcome, rather than throughout the research process (Ahmed & Sundaram, 2011; Kuechler & Vaishnavi, 2008).

The artefact should be evaluated based on specifications determined in the Awareness of Problem

or Suggestion phases (Kuechler & Vaishnavi, 2008), that will determine how well it works in terms of, for example, functionality, consistency, accuracy, performance or usability (Hevner et al., 2004). The Evaluation phase should also identify any deviations from expected behaviours or results (Kuechler & Vaishnavi, 2008). Hevner et al. (2004) describe five kinds of evaluation methods: observational, analytical, experimental, testing, and descriptive.

The artefact undergoes the Evaluation phase with the application of Evaluation patterns such as *Demonstration*, *Experimentation* and *Simulation* (Vaishnavi & Kuechler, 2008).

Demonstration is considered a weak form of validation, but can be appropriate when a solution is new or solves a problem for which no other solution exists. It is generally utilised when it is not possible to use mathematical proofs to validate a solution, but it is still possible to demonstrate that it works in certain situations. Demonstration will show that a solution is feasible, but can also highlight inadequacies (Vaishnavi & Kuechler, 2008).

The Experimentation pattern requires the generation of data from the built artefact and the use of this data to either validate or reject hypothesised results. There are different experiment types. Hypothetical or Deductive experimentation uses the results of past experiments and information gathered in the literature review to build and test the system in various environments. In Prototyping, also known as Hermeneutical or Inductive experimentation, the system and hypothesis are developed through prototyping, and finally in Case-Based experiments, the prototype is built using an initial hypothesis, and is modified using newly gained knowledge as the prototyping progresses (Vaishnavi & Kuechler, 2008). Experiments must be repeatable and reliable, and the results should be generalisable.

The Simulation pattern is used when a complex research problem cannot be explored in a real world setting, but can be modelled or simulated. The researcher should develop a model of the problem and solution, including objects and interactions that can be captured for the purpose of performance evaluation. A suite of test data is then developed that can be used to evaluate the artefact.

In this research project, the Demonstration, Experimentation and Simulation patterns are all used to evaluate the various artefacts. Micro-evaluations occurred throughout the development phases of the artefacts, and this informed the further development of both artefacts. Evaluation of the artefacts at the end of each development cycle also informed the Awareness of Problem, providing additional areas to explore through Suggestion. Though the artefacts were developed as separate artefacts, independently of each other, the evaluation of each artefact in turn informed the Suggestion and Development phases for further iterations of the other artefact.

### 3.3.5 Conclusion

The *Conclusion* phase can be the end of a research cycle, or could be the final outcome at the end of the research effort. The final results might not answer the research questions in full, but they are deemed *good enough*. An artefact is considered complete when it satisfies the requirements of the problem it was intended to solve (Hevner et al., 2004). The results are then written up, but may not be considered firm or final, and any loose ends can become the subject of further research (Hevner et al., 2004; Vaishnavi & Kuechler, 2013). Hevner et al. (2004) state that the research must be presented to allow practitioners to take advantage of its benefits, also allowing researchers to build upon it for further extension and evaluation. Though the end result of a DSR project could be difficult to understand, it could still be considered a successful result, since it could provide a platform for further research (Hevner et al., 2004). Researchers learn or discover when something works or when it does not (Vaishnavi & Kuechler, 2008).

Publication patterns can be applied in the conclusion phase of the research project where the research could be presented to different interested communities.

The conclusion of this research project occurred when it became clear through the multiple iterations that the research question could be adequately answered with the creation of firm knowledge that would be of interest to the research and industry communities. The knowledge contribution to industry will be discussed in Section 4.4.7.

## 3.4 Tools: Artefact Construction

It was identified in the Suggestion phase that Problem Space Tools and Techniques, and Research Community Tools and Techniques were both patterns that could be applied during this phase to identify tools and techniques applicable to the problem space domain, and to identify how the research or industry communities solve similar problems. The research project investigates the automation of testing for cross-platform games, and therefore it was essential to utilise industry standard cross-platform tools and techniques in the design and construction of the research artefacts. These tools and techniques are described in this section.

### 3.4.1 Industry Tools and Techniques for Cross-Platform Game Development

Cross-platform game development is the process of creating games that target a variety of platforms. Cross-platform executables are created by recompiling source code for each target platform, resulting in a unique executable file for each target (Thorn, 2008). The variation between different manufacturer's

game console hardware can be significant (Irish, 2005). Developing in a multi-platform way is difficult, as the developer must take into account the variety of scenarios for different hardware configurations (Shirley & Marschner, 2009), and may require the developer to implement unique solutions for each target platform. Problems encountered in cross-platform development include developer efficiency, certification failure, user experience and programming issues (Ruskin, 2008). Portability involves moving a game’s source code from one OS to another and is a key component of cross-platform development, since modules must be able to be ported between platforms (Wihlidal, 2006). Game developers favour developing cross-platform games on PC workstations, as loading and debugging is faster than on console hardware (Ruskin, 2008). This allows for more rapid iteration during the development phase.

Abstraction is a useful development practice for multi-platform development (Llopis, 2003). A common abstract interface provides encapsulation, which hides platform-specific implementation details (Eberly, 2005) and allows API independence to be achieved (Zerbst & Duvel, 2004). Supporting the large variety of PC configurations for a single operating system, such as Microsoft Windows, is also possible through abstraction (Shirley & Marschner, 2009). An abstraction layer allows for the graphics rendering interface to be utilised by a game developer in a generic way (LaMothe, 2003), for example, so that a game can be built to execute using either the DirectX or OpenGL CG API (Phillips, 2002). The variety of graphics performance on platforms can be taken into account by changing the quality of the graphics based upon the platform’s capabilities (Shirley & Marschner, 2009). Other native platform differences must also be abstracted, such as the platform’s file system (Gregory, 2009). This is due to the need to hide the native platform’s endianness, also known as byte ordering. File systems vary between target platforms (Fahy & Krewer, 2012) and loading files in a cross-platform environment requires the developer to be aware of the target platform’s endianness (Phillips, 2002).

## C++

A cross-platform foundation can be built from a common framework of classes and functions (Thorn, 2011). Abstraction and polymorphism can also be utilised to hide platform specific details within subclass definitions (B. Sutherland, 2014). The common programming language used for high performance commercial game development is C++ (Gregory, 2009; Kelly, 2012; Ostrowski & Aroudj, 2013). C++ is able to be utilised across all major platforms (Fahy & Krewer, 2012) and is platform-independent, with a compiler for each target platform (Llopis, 2003). The language allows game developers to undertake high-level programming practices, with support for low-level system access (Llopis, 2003).

The size in bytes of each primitive type, as well as pointer types, should be the same for all

target platforms (B. Sutherland, 2014). This can be achieved by using the C++ `typedef` aliases to abstract variable types and sizes to ensure consistency in a cross-platform environment (Phillips, 2002). Not taking account of the different sizes of types per platform can be a cause of graphical defects (B. Sutherland, 2014). `sizeof` and `assert` can also be used to ensure that each target platform's types match what the developer expects (Goodwin, 2005).

C++ preprocessor macros can be used to accommodate differences between target platforms (Eberly, 2005; Reddy, 2011; B. Sutherland, 2014). Macros determine which platform the source code is being compiled for and developers can define unique platform macro symbols (B. Sutherland, 2014).

Implementations of the C++ Standard Template Library (STL) exist for all major game platforms (Llopis, 2003). STL is a valuable cross-platform asset (Phillips, 2002), however, game developers need to be aware of STL implementations and behaviours on different platforms and that it may not be suitable for memory-limited platforms (Gregory, 2009). Available memory is a constraint in game development, and each platform will offer different sizes and types of random access memory (RAM) (Irish, 2005). For this reason, developers need to keep track of memory allocations (Ruskin, 2008).

## Engines

A game engine must hide dependencies on the OS from the game application source code (Eberly, 2005), and game logic source code should be separate from reusable engine code (B. Sutherland, 2013). Cross-platform engines isolate the game implementation from the underlying platform's hardware (Gregory, 2009). Game engine developers write wrappers around specific platforms' native APIs to abstract common behaviour provided by the engine (Gregory, 2009). Wrapping the operating system's API allows the game engine to ensure that application behaviours are identical for each target platform (Gregory, 2009). This is described as the *Platform Independence Layer* (Gregory, 2009).

## Unified Modeling Language

A common method for creating and communicating the design of software, including game software and tools, is Unified Modeling Language (UML) (Bethke, 2003; Flynt & Salem, 2005; Wihlidal, 2006). This language provides a visual way to describe how parts of a software's design fit together (Wihlidal, 2006). UML *Class Diagrams* describe classes and their relationships, providing a static view of a system's design (Flynt & Salem, 2005). Class diagrams can detail the entire system design, or focus on particular subsystems (Bethke, 2003). Object-oriented design relationships can be shown between classes, broadly categorised into the *is a*, *has a* and *knows/uses a* relationships.



Layer 8:	Build Configuration's (Debug/Profile/Release) Output Executable
Layer 7:	Cross-Platform Video Game's Native Source Code (Compiler)
Layer 6:	Game Engine (Reusable Framework/Abstraction)
Layer 5:	Graphics Application Programming Interface (API)
Layer 4:	Graphics Processing Unit Device Driver
Layer 3:	Operating System/Kernel
Layer 2:	Graphics Card Hardware: GPU
Layer 1:	Computer Hardware: CPU

Table 3.1: A target platform's layered architecture.

### 3.4.2 Cross-Platform Development: The Layers of a Target Platform

As previously described, the combinations of hardware, operating system, device driver, API, game engine and build configuration lead to a number of combinations which comprise the *target platform* of any single game if it is developed in a cross-platform manner. Therefore, to develop and test for graphical defects in a cross-platform game all layers of abstraction, as shown Table 3.1, must be considered.

### 3.4.3 Image Comparison

Typically, computer graphics rendering output is captured and stored to form a golden truth image which is subsequently compared to future test outputs. As discussed in Chapter 2, image comparison techniques are commonly utilised to test these graphical outputs. These range from binary pixel-by-pixel comparison (Bay, 2016; Grønbaek & Horn, 2009; Yee & Newman, 2004; Zioma, 2012), to human visual perception methods such as SSIM (Timofeitchik & Nagy, 2012; Wang et al., 2004). It is possible to include tolerance for a certain number of incorrectly matched pixels when using pixel-by-pixel comparison (Pranckevicius, 2011).

## 3.5 Limitations

The scope of the research project is limited, as I was the sole researcher and developer. As discussed in Chapter 2, Unity Technologies, an industry example of the implementation of automated graphical testing for a game engine, dedicates entire teams of specialist developers and engineers to the development and testing of their product. A sole developer is unable to match the scope of a commercial development team with years of development time. Commercial AAA games such as *Assassin's Creed Unity* (Ubisoft Montreal, 2014a) and *Bioshock* (2K Boston and 2K Australia, 2007), contain

significantly complex computer graphics renderings which are the result of programming and artistic development over years of iteration and experimentation. As a sole developer it was not possible to replicate this scale of graphical complexity.

Target platform availability was another limiting factor in the research project. Console development hardware is licensed to game development studios and first-party manufacturers do not make their software development kits (SDK) available to the public. Specific platform details are protected by non-disclosure agreements (NDA) and closed proprietary systems. As highlighted in the Chapter 1, NDAs are also prevalent for the protection of game development industry practices, making it difficult to ascertain widespread internal studio details regarding development and implementation of testing for graphical defects in games.

To mitigate these limitations, the project mimicked commercial game development practices by utilising the same techniques and technologies as employed in a AAA or independent game development studio. This enables the research findings and the practices developed to be employed in real world scenarios at the completion of the research project. Small-scale graphical tests scenes were created across a selection of available target platforms, allowing cross-platform game scenarios to be simulated. The available target platforms were:

- Desktop: Generic PC, Windows 7, GTX560Ti;
- Laptop: Apple MacBook Air, Mac OS X, Intel HD Graphics 4000;
- Single Board Computer: Raspberry Pi 1 Model B+, Raspbian, VideoCore IV;
- Mobile Phone: iPhone 6 Simulator, iOS 6;
- Tablet: Samsung Galaxy Tab 2 10.1, Android 4.2.2 Jelly Bean.

DSR can require the simplification of a problem, such that it may only represent a subset of the whole problem (Hevner et al., 2004). This idea is outlined in the Development phase patterns Divide and Conquer and Building Blocks. Breaking the problem down into smaller blocks might result in an outcome not realistic enough to have a significant impact on industry practice. However, the results may form the starting point for future research, leading to more relevant and realistic results (Hevner et al., 2004).

## 3.6 Summary

This chapter has detailed the methodological approach utilised in this research project, including background on the DSR methodology, the methods employed in the design of the research project,

and the tools utilised in the construction of the artefacts. Limitations are outlined, alongside the available target platforms used throughout the project.

Section 3.3 provides an overview of the DSR phases of the project. The details of artefact construction and evaluation, the Development phase iterations, are discussed in the next chapter.

## Chapter 4

# Artefact Development and Evaluation

“Learning and investigation  
through artifact construction”

---

Vijay K. Vaishnavi and William  
Kuechler Jr., Design Science  
Research Methods and Patterns  
Innovating Information and  
Communication Technology  
(Vaishnavi & Kuechler, 2008, p.  
187)

### 4.1 Introduction

This chapter will report the results of each phase of the DSR process, beginning with a recap of the Awareness of Problem phase, which includes defining the scope of both the problem and the research project, before moving on to the Suggestion and Development phases. Throughout the research project, iteration occurred, whereby the Suggestion and Development phases were revisited after each Evaluation phase. The iterative approach ended when a *good enough* solution was reached, which could answer the overall research question.

## 4.2 Awareness of Problem

Prior to developing the initial artefact design in the Suggestion phase of the project, it was important to define the scope of the problem. This was discussed predominantly in Chapter 1 and Chapter 2. Section 2.2 outlined the scale of the graphical defect issue, the primary motivation for this research project. Section 1.5 summarised the environment in which this problem has developed. The technology utilised in the development of games is various, and changes rapidly. As previously discussed, the game development industry utilises human testers to detect bugs, however, there is support for the increased use of automated testing practices.

The scope of the research project was outlined in Section 3.5, which detailed the conditions under which this research project was conducted. These included the researcher as the sole developer, and a selection of available target platforms to develop and test with that could be used to simulate a game development environment. Industry standard cross-platform tools, as described in Section 3.4, were used to create the artefacts presented in each iteration.

## 4.3 Suggestion

The Suggestion phase took into account the *Divide and Conquer* and *Building Blocks* patterns described in Section 3.3.3. These patterns propose that when researching complex problems, it is useful to start with a minimum viable solution, and to iterate to build on this solution, or to subsequently build more blocks that in combination can answer the research question. In the context of this research project, this approach made sense. The game development industry must accommodate new technology regularly and this introduces risk (Kanode & Haddad, 2009). It also means that game developers can be unaware of problems until they are required to develop for a particular platform (Petrillo et al., 2008). This could have been a problem in this research project and I had to plan to accommodate any issues that could arise in future iterations. Pranckevicius (2007) further supported the idea of starting small, stating that initial Unity Technologies test automation could be conducted using only a single computer and different graphics cards that could be switched out for testing, as discussed in Section 2.3.1.

The Suggestion phase also had to be designed for the limited number of target platforms available to the research project, as outlined in Section 3.5. Proprietary consoles were not available, however, a basic cross-platform graphics simulation could be constructed and tested on the available *Generic PC* (running Windows) and *MacBook Air* (running Mac OS X), with the option to add more platforms in later iterations. In proposing a minimum viable solution, each layer of the target platforms, as described in Table 3.1, had to be considered. As OpenGL is one of the main APIs used in CG

(Sherrod, 2008), it was selected as a starting point for cross-platform development. A cross-platform abstraction was needed to mimic a simple game engine, which facilitated code reuse between target platforms, as described in Section 1.5.2. This would allow the same graphical content to be executed on the different target devices, simulating the work of Zioma (2012) at Unity Technologies.

It was important to decouple the testing tool from the game implementation, a feature also seen in the work of Lewis and Whitehead (2011a). In order to understand the way in which cross-platform games need to be developed to utilise automated testing techniques, two initial artefacts were planned — one to create simple game-like graphical output for testing, where the source code would be instrumented to facilitate testing (Lewis & Whitehead, 2011a; Varvaressos et al., 2014), and an automated tool to test the graphical images for defects. Cross-platform development strategies had to be utilised in the construction of the output artefact, as discussed in Section 3.4.

Existing research provided a starting point for other features, such as the use of per-pixel comparison to known template images — something that was identified in Chapter 2 as common to many CG automated testing systems — and the presentation of a test image, result image and difference image (Pranckevicius, 2011). It was expected that as described in the DSR methodology, micro-evaluations would take place during the Development phase, which would allow for rapid refinement of the artefacts being created.

## 4.4 Development

As discussed in Section 4.3, the Development phase consisted of the iterative construction of two artefacts — a *Cross-Platform Test Scene Generation* artefact, and an *Automated Comparison Tool* artefact. An overview of the iterations undertaken during the Development phase for both artefacts is shown in Figure 4.1. Section 4.3 highlighted the need for decoupling the testing tool and the game implementation. The cross-platform test scene generation artefact represents a game implementation which initially produced simple 2D graphical output and in later iterations became more game-like. The automated comparison tool is the testing tool that sourced the graphical output for comparison and presented automated test results. These artefacts were iterated on individually after Iteration 1, however, the iterations of the individual artefacts influenced each other. Graphical output from Iterations 3 and 4 was utilised by Iteration 5 to evaluate and refine the automated comparison tool. Graphical output from Iteration 7 was tested by the Iteration 5 tool to validate the cross-platform game implementation and the tool helped to discover cross-platform graphical defects. These Iteration 7 defects were then corrected and the cross-platform implementation was re-tested. The interrelatedness of the iterations is shown through the dotted arrows in Figure 4.1. Knowledge contribution occurred at

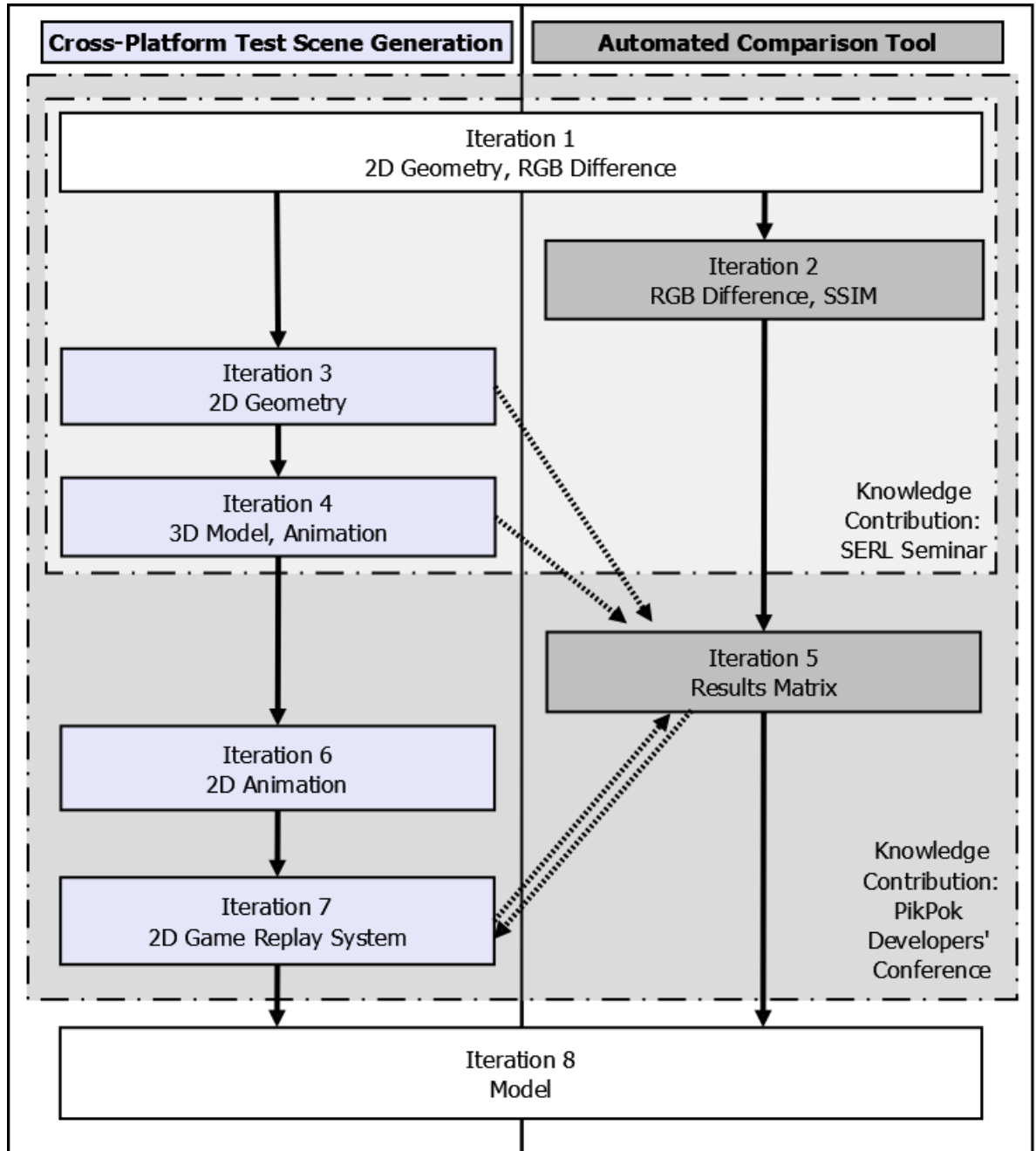


Figure 4.1: DSR artefact iteration overview.

	Layer Description	Target Platform 1 <i>Generic PC</i>	Target Platform 2 <i>MacBook Air</i>
Layer 8:	Build Target's (Debug/Release) Output Executable	Debug	Debug
Layer 7:	Cross-Platform Video Game's Native Source Code (Compiler)	Visual Studio 2012	Xcode 6.3.2
Layer 6:	Game Engine (Reusable Framework/Abstraction)	Bespoke C++	Bespoke C++
Layer 5:	Graphics Application Programming Interface (API)	OpenGL FFP (GLEW 1.10.0, GLFW 3.0.4)	OpenGL FFP (GLUT)
Layer 4:	Graphics Processing Unit Device Driver	Nvidia 10.18.13.5362	OS dependant
Layer 3:	Operating System/Kernel	Windows 7 Professional	OS X Yosemite 10.10.4
Layer 2:	Graphics Card Hardware: GPU	Nvidia GeForce GTX 560 Ti	Intel HD Graphics 4000 1024 MB
Layer 1:	Computer Hardware: CPU	3.30 GHz Intel Core i5 2500K	1.8 GHz Intel Core i5

Table 4.1: Iteration 1: Target platform matrix.

two points during the Development phase — following Iteration 4 and Iteration 7. The work discussed in each contribution is shown in Figure 4.1 through the use of the dot/dash boxes. The final iteration — Iteration 8 — unifies the knowledge gained from the development and evaluation of each iteration into a generalised model which can be used for the construction of a system for the automated testing and validation of computer graphics implementations for cross-platform game development.

#### 4.4.1 Iteration 1

Two target platforms were selected as a starting point for designing and implementing a 2D cross-platform rendering artefact. The target platform matrix can be seen in Table 4.1. The rendering artefact's design, as seen in Figure 4.2, featured a rendering interface which was implemented using the fixed-function OpenGL graphics pipeline. This abstraction allowed for the differences between Layers 1 to 5 (see Table 4.1) of the target platforms to be generalised and for any particular platform details to be implemented by the derived classes, while allowing computer graphics test scene content to be generated through the abstract interface. The artefact's source code was compiled for each platform, and executed. Two test scenes of fixed height and width were created via cross-platform C++ source code (Layer 6, see Table 4.1) which was common to both target platforms. These test scenes were static 2D renders, providing a simulation of cross-platform computer graphics rendering.



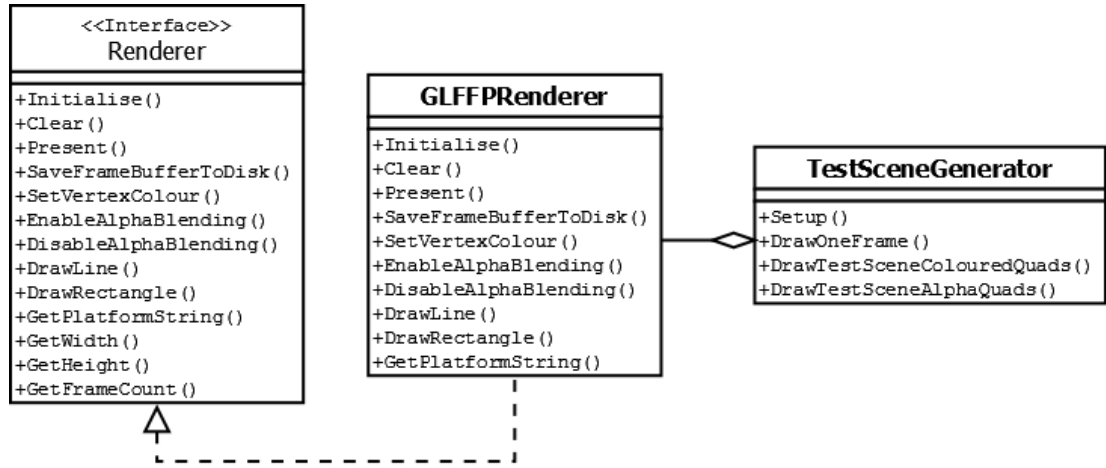


Figure 4.2: Iteration 1: Test scene generation UML design.

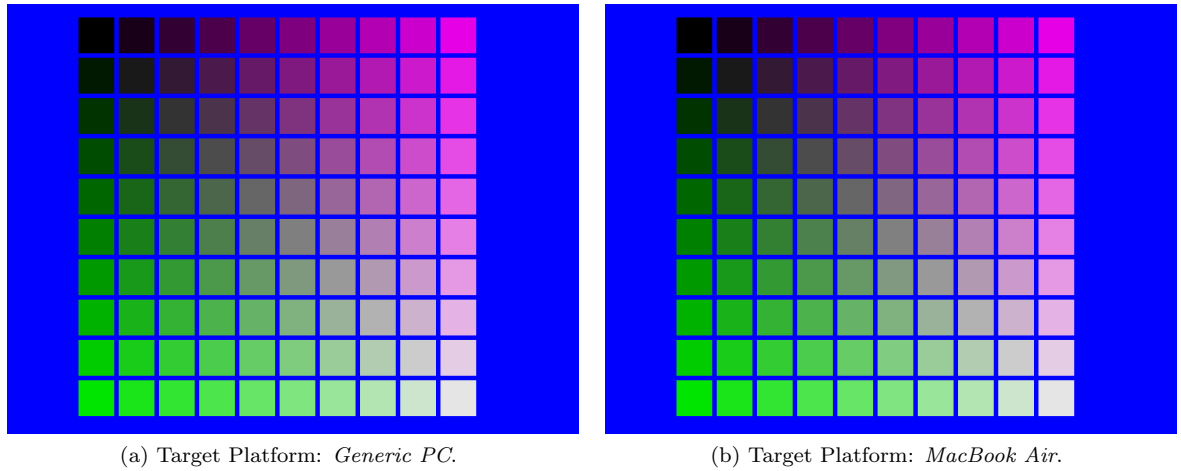


Figure 4.3: Iteration 1: Test scene outputs: Coloured primitive rendering.

One test scene rendered coloured primitives and another rendered alpha blended coloured primitives. The rendering artefact was also instrumented to support saving the rendered framebuffer to file. After each test scene was rendered, the test scene captured the framebuffer to disk as a bitmap image. The resulting image files were then retrieved from their target platforms and then accessed by the comparison tool artefact. Sample output from the coloured primitive rendering test scene from the two target platforms can be seen in Figure 4.3, and from the alpha blended coloured primitive rendering test scene in Figure 4.4.

The second artefact, the comparison tool, was constructed as a Microsoft Windows executable using Visual Studio 2012 and C#, the design of which can be seen in Figure 4.5. This tool could load and compare the resulting test scene image outputs from each target platform. The comparison algorithm utilised was RGB per-pixel and would report to the user a difference image, the number of pixels that matched, mismatched, and the percentage considered correct between the two target platform outputs.

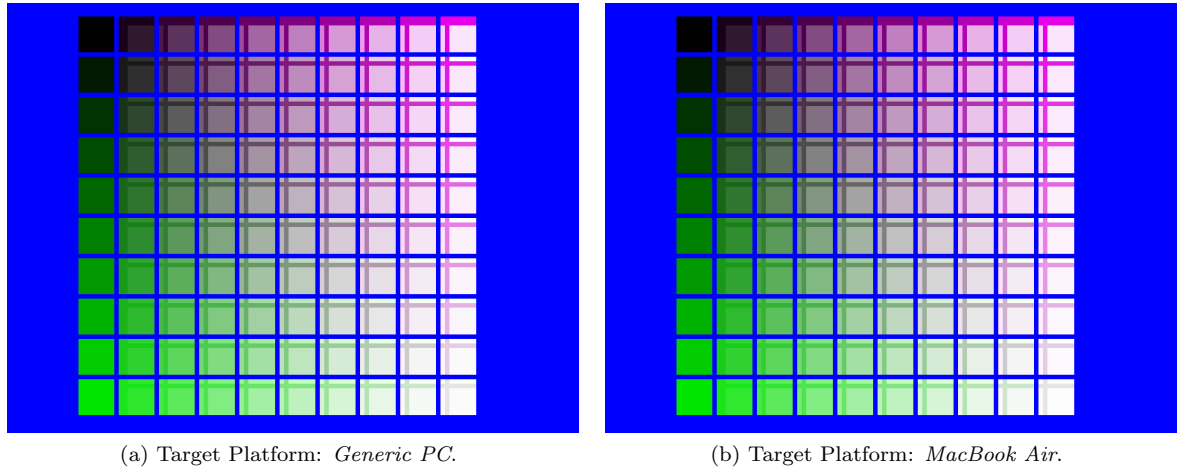


Figure 4.4: Iteration 1: Test scene outputs: Alpha blended coloured primitive rendering.

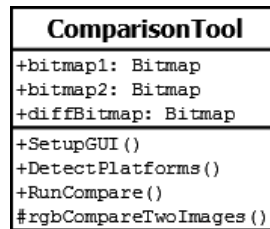


Figure 4.5: Iteration 1: Comparison tool UML design.

The overall pipeline for generation of test scenes and comparison results can be seen in Figure 4.6.

A bug was purposefully introduced to the C++ source code in order to create a simulated flawed graphical output for one target. The flawed image was compared against one from the other target to test the comparison tool’s difference image generation and per-pixel comparison reporting. An example of this can be seen in Figure 4.7. Correct pixels, incorrect pixels, and the percent correct are reported at the top of the difference image.

## Evaluation

The automated comparison tool was able to generate an RGB difference image and report per-pixel difference results based upon output from the two target platforms. The tool had to be manually configured to select which test scenes to compare. Cross-platform test scene generation with C++ implementation was possible, and the resulting renders were able to be retrieved from disk. Test scenes were able to be written once and used on each target. This validated the viability of each platform’s development tool chain.

As a building block, the two artefacts created for Iteration 1 performed as expected and set a foundation to further investigate using other image comparison methods, such as SSIM.

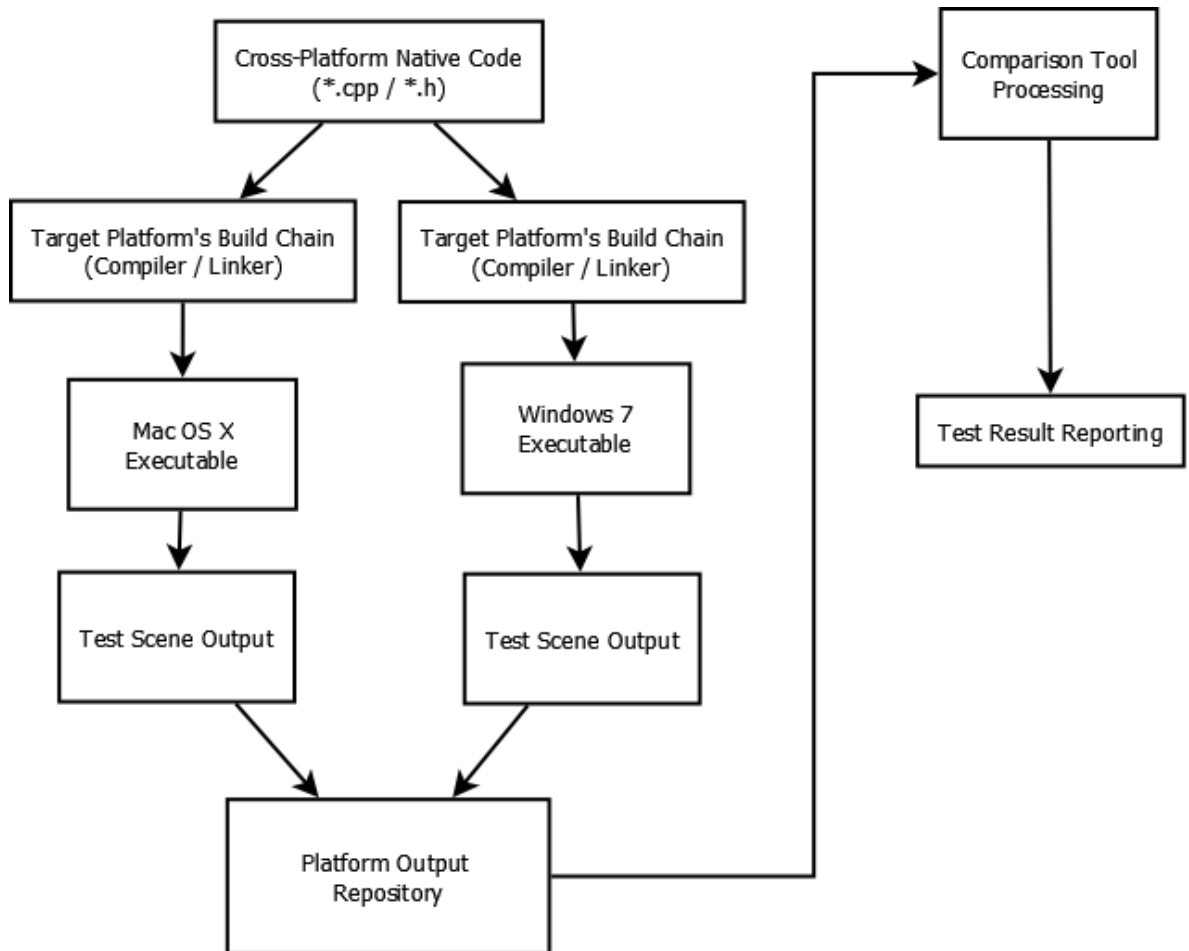


Figure 4.6: Iteration 1: Pipeline overview.

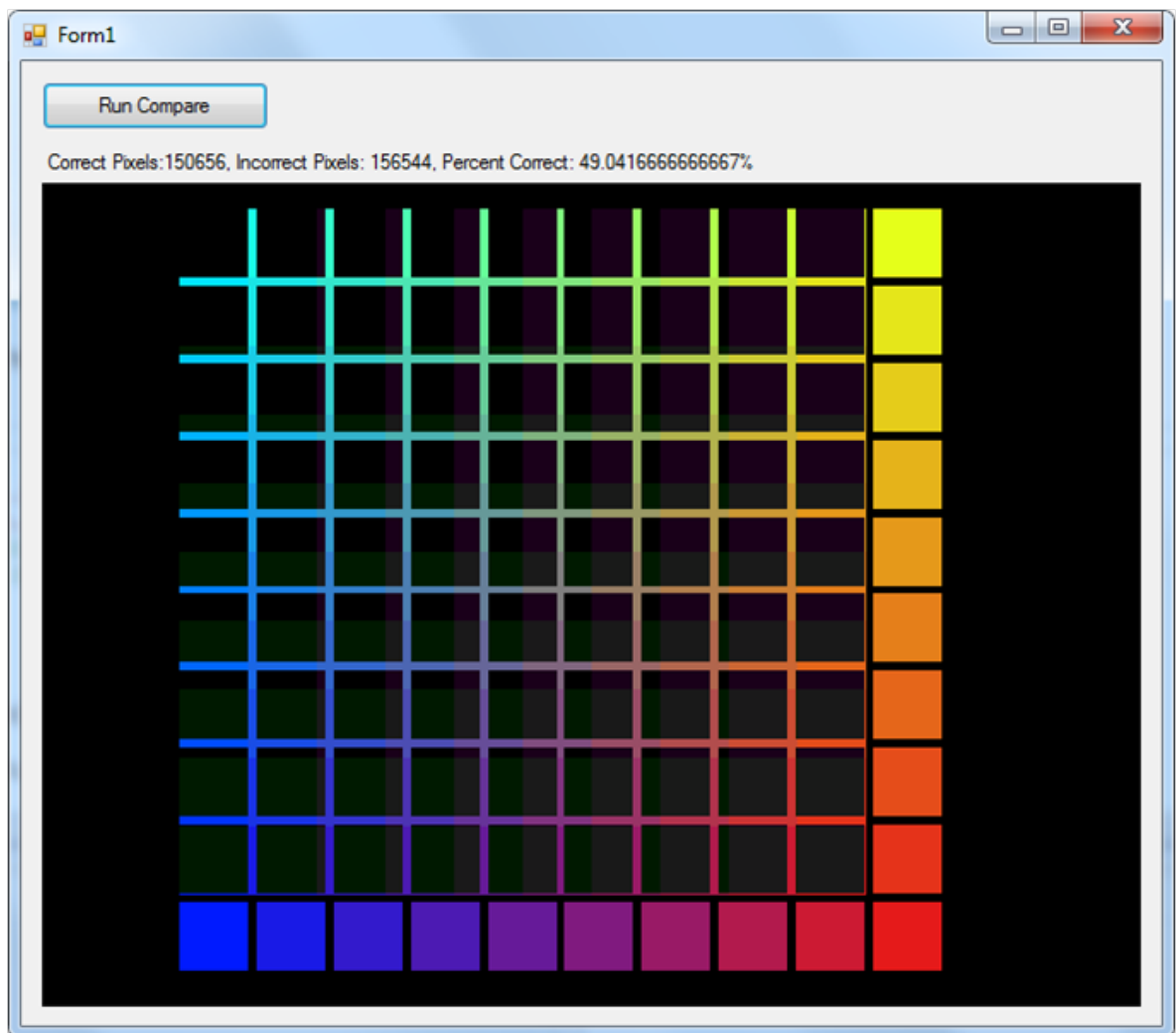


Figure 4.7: Iteration 1: Comparison tool difference image example.

### 4.4.2 Iteration 2

Inspired by the work of Timofeitchik and Nagy (2012), Iteration 2 focused on development of the C# automated comparison tool and the introduction of SSIM as an image comparison method. This allowed for a more human-like comparison metric to be generated based upon the images compared. Per-pixel comparison was retained, as difference image generation from Iteration 1 was able to successfully highlight differences between platform rendering outputs. The design of Iteration 2 can be seen in Figure 4.8.

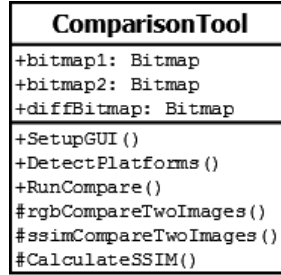


Figure 4.8: Iteration 2: Comparison tool UML design.

The automated comparison tool was able to load test scene data based upon a directory hierarchy, where each directory represented a target platform. A combo box was added to allow a human tester to switch between test scenes. The tool was also expanded to support more than two platforms and each platform’s test scene output was displayed in separate tabbed panes, allowing a human tester to view rendering output from all target platforms for a selected test scene. When clicking between tabs, the tool reported the RGB correct, incorrect and percentage correct metrics for the last two tabs viewed, as well as the SSIM result. This result was displayed as a number between 0 and 1, as seen in Figure 4.9. A checkbox allowed toggling of the generation of the RGB difference image, which when activated was added to another tabbed pane. These features provided a human tester the ability to toggle between reference images and the difference image, similar to what Bay (2016) reported is part of the Unity Technologies graphics test tool.

The addition of two sliders allowed first for the test scene renders to be scaled in size, and second for the RGB difference tolerance to be exaggerated to help a human tester to more easily notice minor per-pixel differences that might otherwise be difficult to detect.

To test this iteration of the automated comparison tool artefact, renders from the PC version of AAA game *Watch Dogs* (Ubisoft Montreal, 2014b) were utilised. These renders were retrieved from the Burnes (2014) article on *Watch Dogs* graphics performance. The images were captured at different graphics settings, resulting in four variations of the same scene: ultra, high, medium, and low, as seen in Figure 4.10. Each graphics setting could be considered a different target platform,

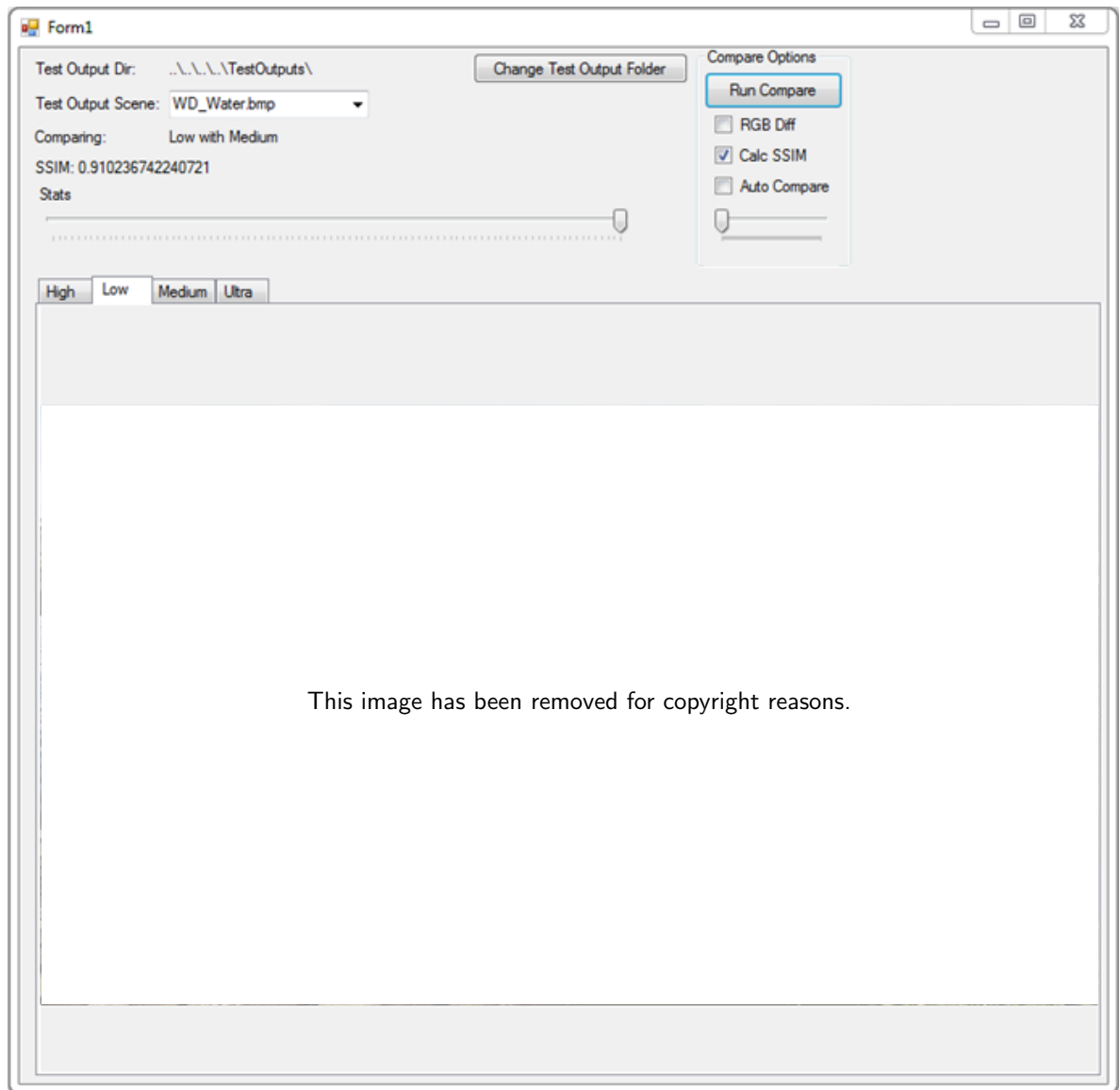
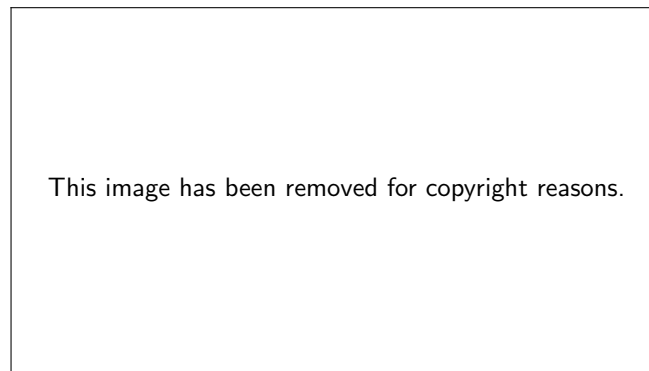


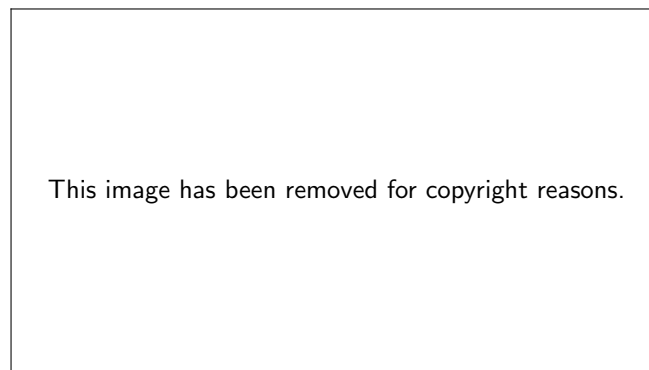
Figure 4.9: Iteration 2: Comparison tool output example.



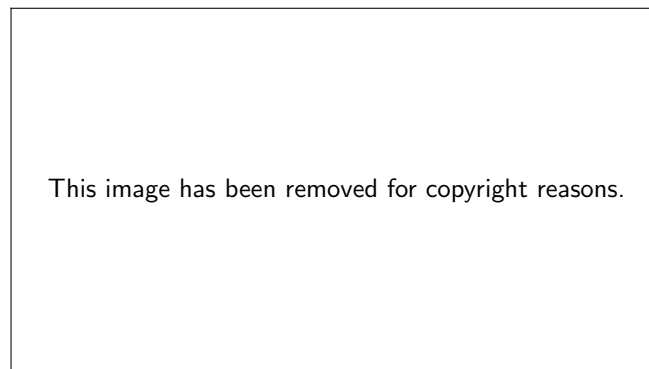
(a) Low.



(b) Medium.



(c) High.



(d) Ultra.

Figure 4.10: Input data: *Watch Dogs* (Ubisoft Montreal, 2014b), rendered scene at various graphical quality settings (Burnes, 2014).

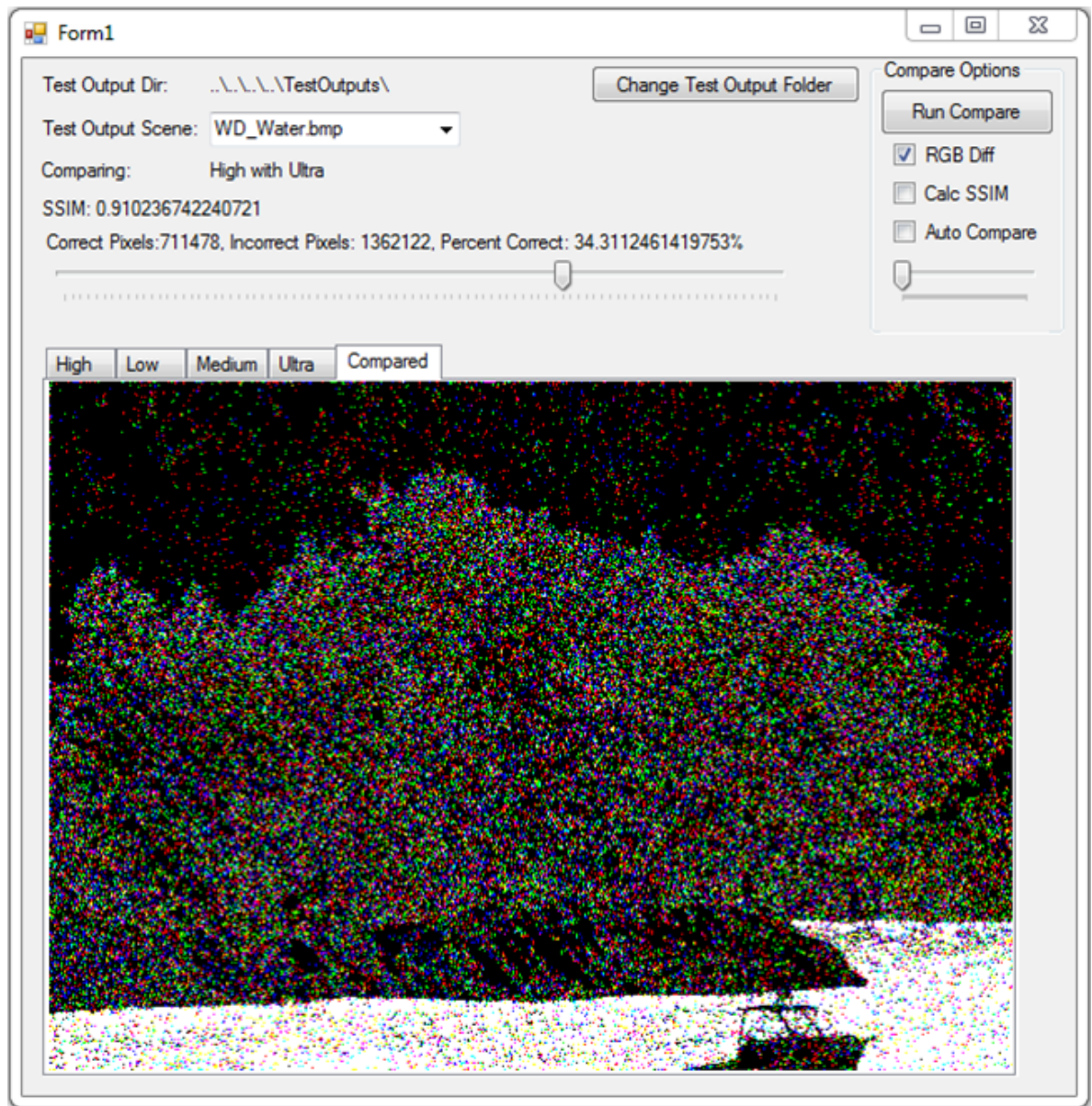


Figure 4.11: Iteration 2: Comparison tool output example where the selected Compared tab shows the RGB difference between the High and Ultra graphical quality settings.



% Pixel Match	Medium	High	Ultra
Low	44.2%	46.6%	32.1%
Medium	—	44.4%	34.1%
High	—	—	34.3%

Table 4.2: Iteration 2: *Watch Dogs* test scenes RGB per-pixel comparison.

SSIM	Medium	High	Ultra
Low	0.910236	0.809589	0.785479
Medium	—	0.888953	0.861518
High	—	—	0.972973

Table 4.3: Iteration 2: *Watch Dogs* test scenes SSIM comparison.

where the application could control the quality of its rendering output. This simulated rendering with different graphics card hardware capabilities. An example of the RGB difference output can be seen in Figure 4.11, where the high and ultra scenes were compared. In this output, the difference image has been scaled to focus on a particular area that could be of interest to a developer.

## Evaluation

The new features added to the automated comparison tool allowed for multiple test scenes and multiple platforms to be loaded and compared. The results of comparing each *Watch Dogs* test scene can be seen in Table 4.2 and Table 4.3.

When visually inspecting the images in Figure 4.10, they look wholly similar to each other, even though there are rendering differences. The RGB per-pixel comparison results scored less than 50% per-pixel matches, suggesting significant difference. However, the SSIM results scored between 0.78 and 0.97, indicating that the images are in fact similar and hence supporting the prior application of this method in the work of Timofeitchik and Nagy (2012).

### 4.4.3 Iteration 3

As the game development industry has to deal with many SKUs and build configurations, as well as platform differences such as hardware, GPU, OS, and drivers (McShaffry & Graham, 2013; B. Sutherland, 2013; Zioma & Pranckevicius, 2012), Iteration 3 focused on adding more complexity to the 2D cross-platform rendering artefact. In addition, the graphical rendering engine middleware Simple DirectMedia Layer (SDL) was utilised at Layer 6 (see Table 4.4) to facilitate cross-platform development, instead of using OpenGL directly as in Iteration 1. SDL is an open source, C-based API that

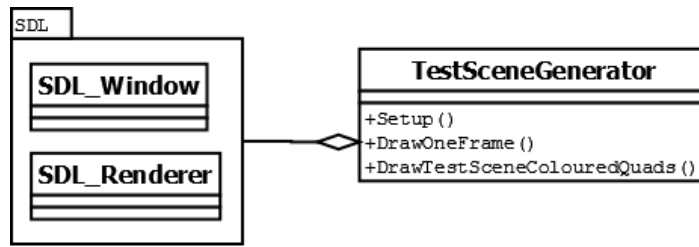


Figure 4.12: Iteration 3: Test scene generation: SDL multi-platform.

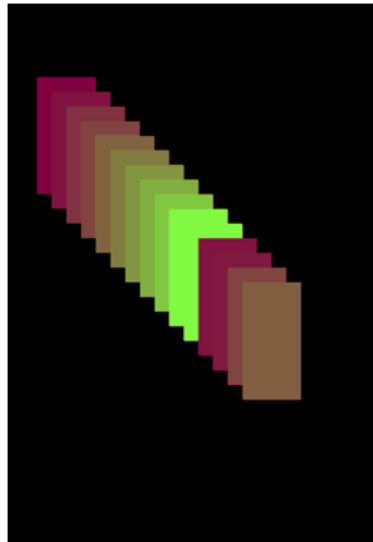
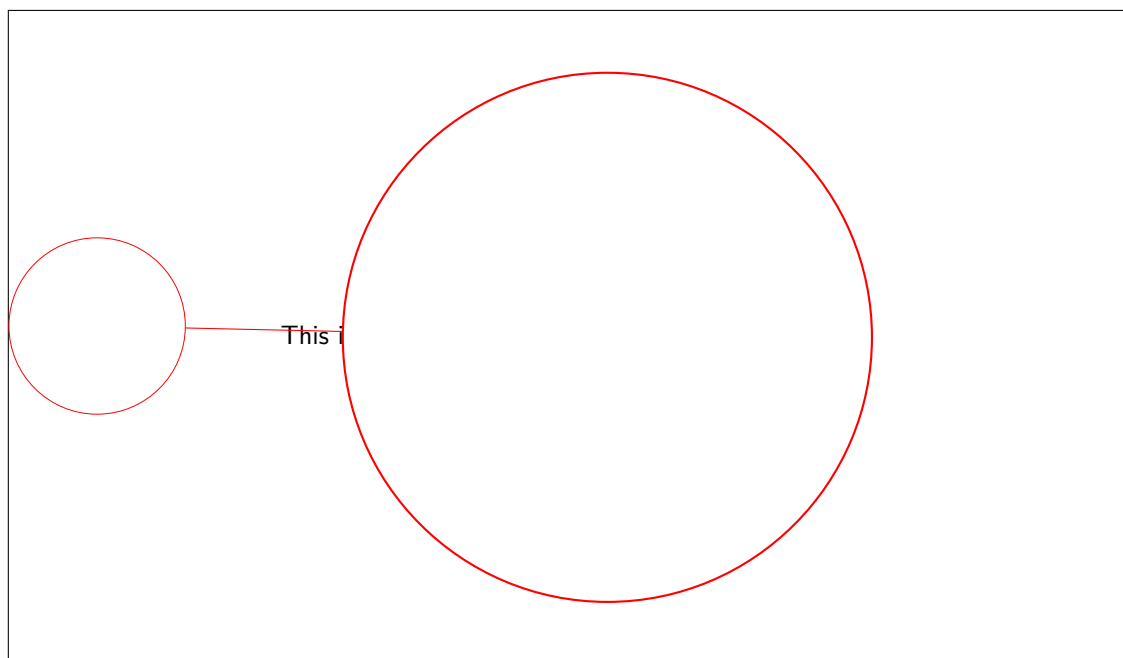


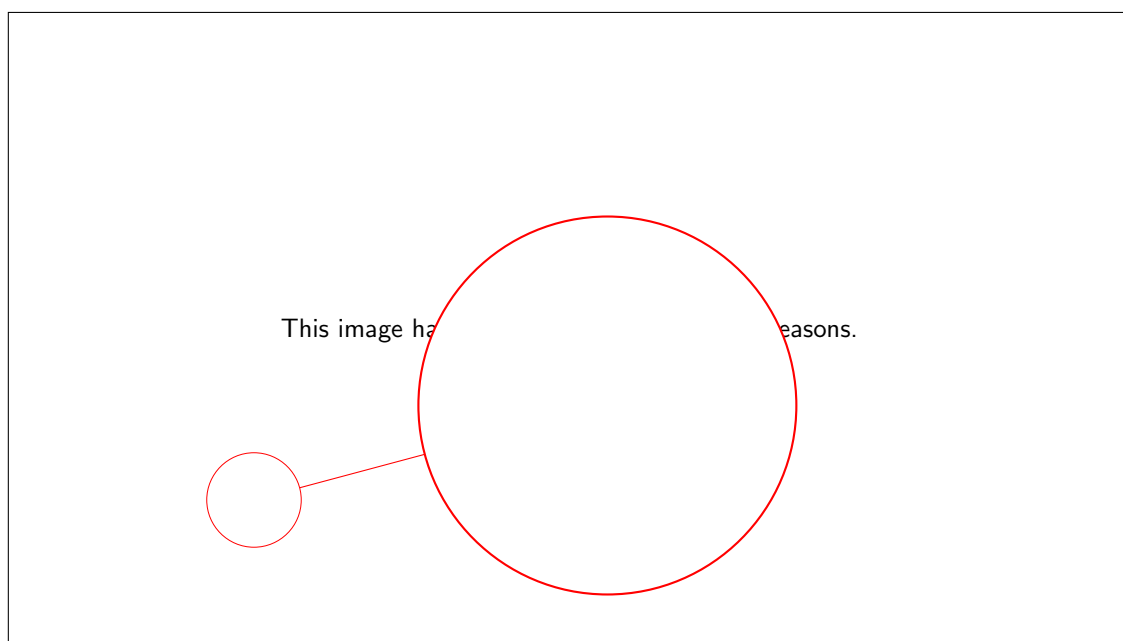
Figure 4.13: Iteration 3: Test scene created with C++ and SDL.

supports rendering to multiple target platforms. C++ code can be written and compiled with SDL to create games. Two recent commercial examples are the cross-platform 2D games *FTL: Faster Than Light* (Subset Games, 2012), published on Windows, Mac OS X, Linux and iOS, and *N++* (Metanet Software, 2015), published on Windows, Mac OS X, and *PlayStation 4*. Both of these games also featured graphical defects, as can be seen in Figure 4.14 and Figure 4.15. Therefore, I considered SDL appropriate middleware to utilise for rendering, as it still allowed game-style C++ code to be written in a cross-platform manner and to further simulate game industry practice. The design of this test scene generator can be seen in Figure 4.12.

In this iteration the aim was to set up the tool chains for each platform, and to generate more test scene data across a larger variety of platforms similar to those used commercially. The target platform matrix can be seen in Table 4.4. This introduced three new hardware targets, and a revised Layer 5 for the two previously used targets. Again, a C++ test scene was developed and deployed to each target platform. The test scene output renders, an example of which is shown in Figure 4.13, could be retrieved for use with the automated comparison tool. A demonstration of a test scene executing on the target platforms can be seen in Figure 4.16.

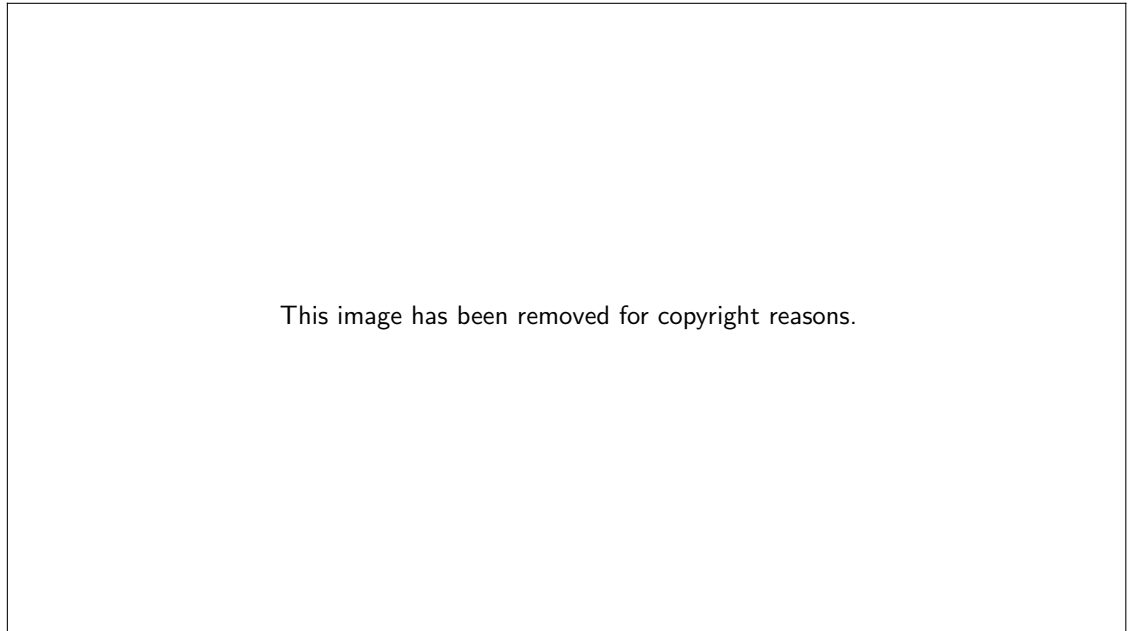


(a) Blocky text on the left side of the frame (Bast, 2012).

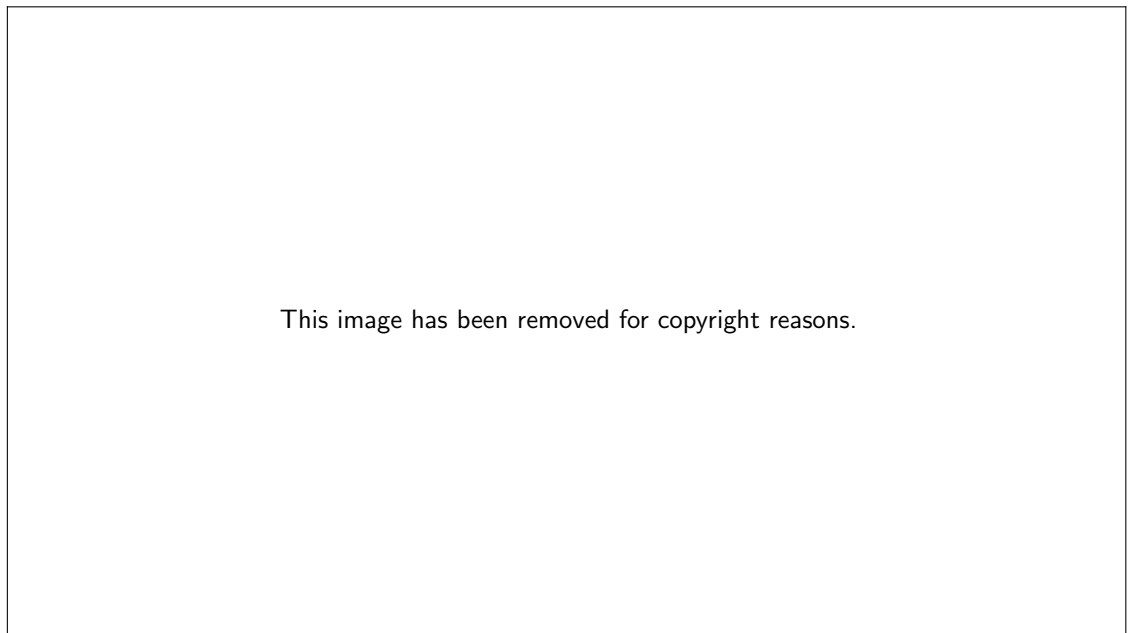


(b) In-game characters are rendered incorrectly, overlapping other game elements (19acomst, 2014).

Figure 4.14: Screen captures of *FTL: Faster Than Light* (Subset Games, 2012) showing two different graphical bugs.



(a) Correctly rendered game level (Pyrarson, 2016).



(b) Game level incorrectly rendered with missing game elements (Razor Volare, 2016).

Figure 4.15: Screen captures of *N++* (Metanet Software, 2015) contrasting a correctly rendered game level with an incorrectly rendered game level.

	Target Platform 3 <i>Generic PC</i>	Target Platform 4 <i>MacBook Air</i>	Target Platform 5 <i>Raspberry Pi</i>	Target Platform 6 <i>iOS</i>	Target Platform 7 <i>Android</i>
Layer 8:	Debug	Debug	Debug	Debug	Debug
Layer 7:	Visual Studio 2012	Xcode 6.3.2	gcc 4.6.3	Xcode 6.3.2	Android Native Development Kit (NDK) r10
Layer 6:	Bespoke C++	Bespoke C++	Bespoke C++	Bespoke C++	Bespoke C++
Layer 5:	SDL 2.0.3 OpenGL 2.0	SDL 2.0.3 OpenGL 2.0	SDL 2.0.3 OpenGL 2.0	SDL 2.0.3 OpenGL 2.0	SDL 2.0.3 OpenGL 2.0
Layer 4:	Nvidia 10.18.13.5362	OS dependant	OS dependant	OS dependant	OS dependant
Layer 3:	Windows 7 Professional	OS X Yosemite 10.10.4	Raspbian GNU/Linux 7 (Wheezy)	iOS 6	Android 4.2.2 Jelly Bean
Layer 2:	Nvidia GeForce GTX 560 Ti	Intel HD Graphics 4000 1024 MB	Broadcom VideoCore IV 250 MHz	iOS 6 Simulator	PowerVR SGX540
Layer 1:	3.30 GHz Intel Core i5 2500K	1.8 GHz Intel Core i5	700 MHz single-core ARM 1176JZF-S	iOS 6 Simulator	Dual-core 1.0 GHz Cortex-A9

Table 4.4: Iteration 3: Target platform matrix.

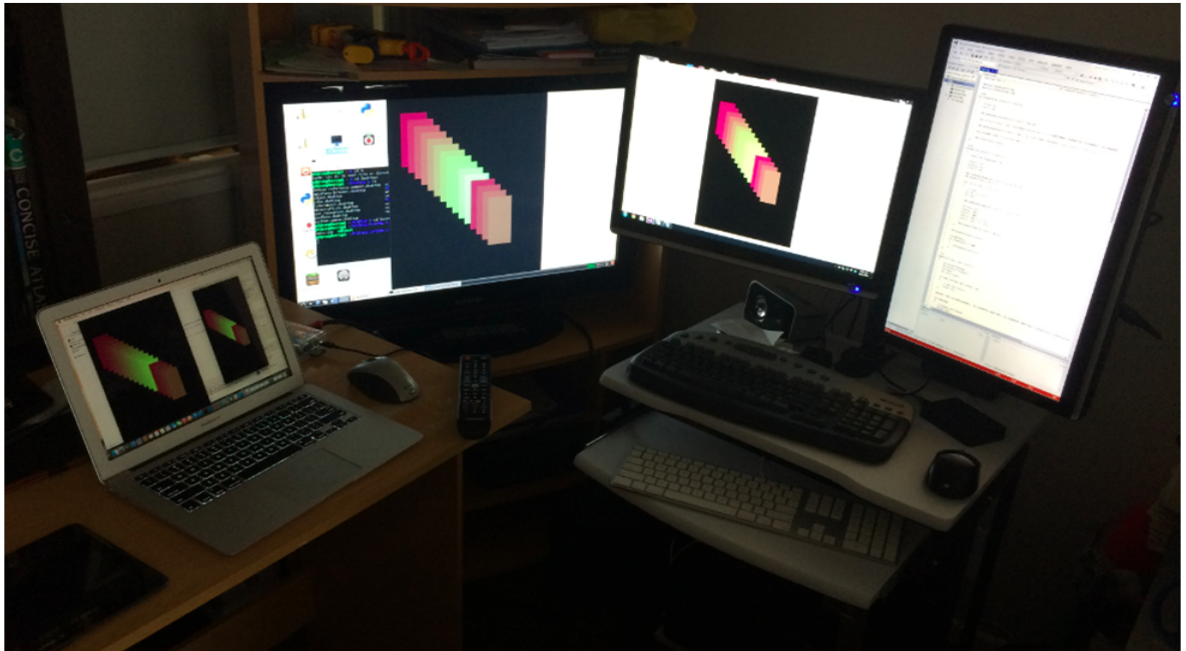


Figure 4.16: Iteration 3: Test scene executing on four of five target platforms.

## Evaluation

C++ test scenes were able to be created and executed successfully on four of the five target platforms — *Generic PC*, *MacBook Air*, *Raspberry Pi* and *iOS*. Framebuffer captures were able to be retrieved from each of these targets, however extracting the render output files from the iOS simulator proved troublesome, as each time the simulator was executed the application was sandboxed in a different directory. This made locating the output files difficult when compared to the Windows, Mac OS X, and Linux targets, which were all relatively simple to retrieve output files from.

The *Android* target, which was built with the Android NDK, did not successfully execute on the target hardware — a Galaxy Tab 2 10.1 — so no test scene output was generated for this platform. The *Android* executable was also unable to be successfully executed in the NDK’s simulator. These difficulties highlighted that while the C++ implementation may work for the majority of platforms, further work would need to be undertaken to discover the causes of the *Android* failure. In this research project, committing resources to diagnosing and solving this problem would take time away from work towards the core research question.

The differences between file systems, and the difficulty in retrieving output files, inspired a future suggestion to build a tool for the network transfer of test scene output from the native game application to a server that listened for test results, rather than saving them directly to a platform’s file system. Test outputs could then be accessed by the automated comparison tool.

#### 4.4.4 Iteration 4

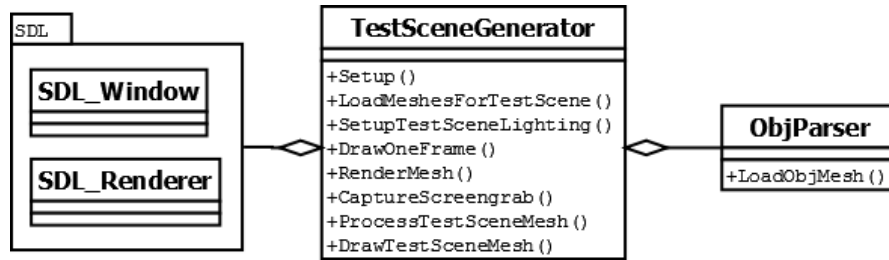


Figure 4.17: Iteration 4: Test scene generation: 3D mesh with procedural animation.

To further explore aspects of cross-platform game development, a cross-platform 3D rendering artefact was designed and built. This artefact targeted the *Generic PC* and *MacBook Air* platforms used in Iteration 3 (see Table 4.4), with 3D rendering using OpenGL 2.0 built on top of SDL. This iteration reduced the number of target platforms to the minimum viable solution, therefore simplifying the test scene output collection and allowing the focus to be on the added complexity of the 3D rendering implementation without the distraction of additional tool chains used in the previous iteration. This created a building block which could be extended in further iterations. A C++ abstraction was developed to generate 3D test scenes, as seen in Figure 4.17. This abstraction implemented the setup of a graphics device capable of rendering 3D content, as well as the loading of 3D mesh data from .OBJ model files, the fixed-function rendering of loaded meshes, and the creation of lights that could be added to the 3D test scene. Cross-platform C++ source code was added to the test scene, where a fixed-rate time update, inspired by the work of Prancevicius (2007), created simple 3D procedural animation. Whereas previous iterations used static 2D programmer-driven test scenes, Iteration 4 allowed art-focused content to form the basis of the test scenes. This simulated a game development scenario where artist-created game assets would be exported into a game engine format and could then be loaded by a game application for rendering.

The 3D rendering artefact could be configured at compile-time, within the C++ source code, to capture rendered scene output at a fixed rate. These test scenes featured an animated 3D model with local y-axis rotation. Figure 4.18 shows an example of one capture from a sequence of captures that took place once every 100 frames. The output generated by each target platform could be run through the automated comparison tool from Iteration 2 (see Section 4.4.2), with example results seen in Figure 4.18 and Figure 4.19.

#### Evaluation

This iteration validated that 3D assets could be loaded and rendered on each target platform using a cross-platform implementation. Test scenes featuring procedural animation and lighting were able

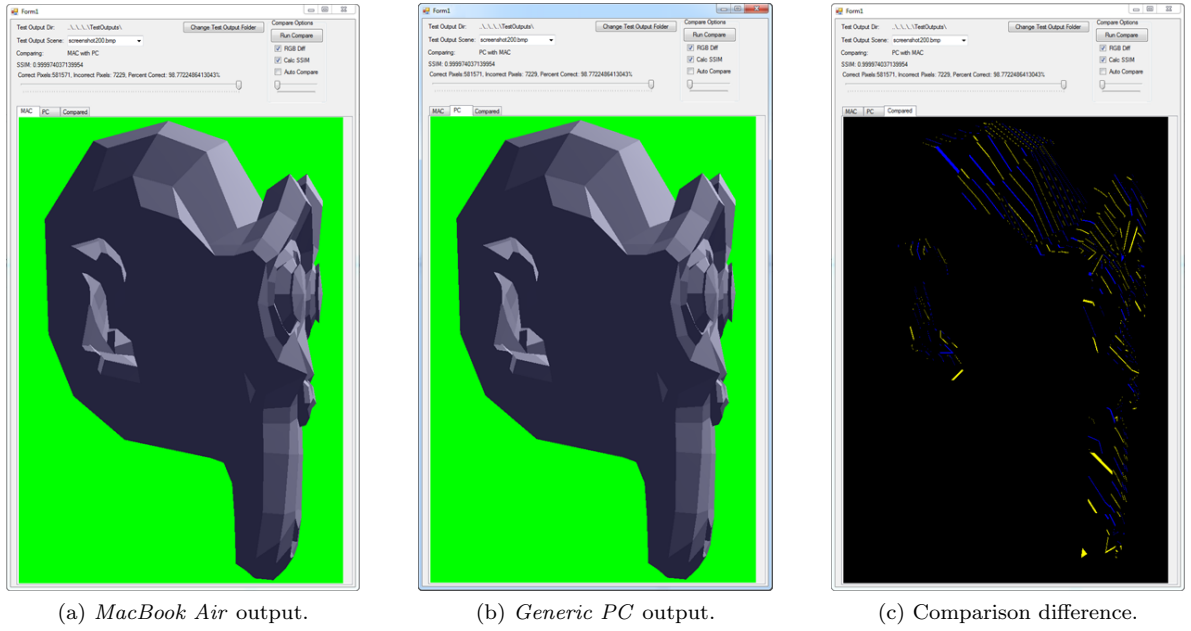


Figure 4.18: Iteration 4: Test scene 3D procedural animation capture, frame 200 example results.

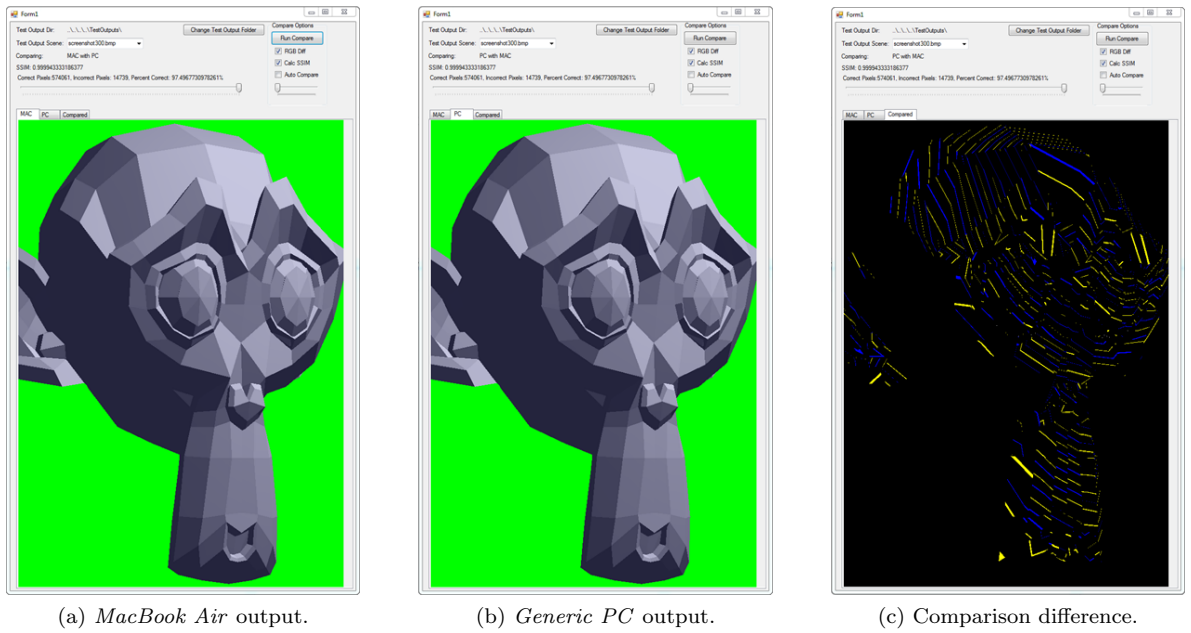


Figure 4.19: Iteration 4: Test scene 3D procedural animation capture, frame 300 example results.



<i>Generic PC vs MacBook Air</i>	Frame 200	Frame 300
Per-Pixel Percent Correct	98.7%	97.5%
SSIM	0.999974	0.999943

Table 4.5: Iteration 4: Example comparison output results.

to be simulated, and fixed-time step framebuffer captures could be captured from each target. These results could then be compared by the automated comparison tool.

The 3D output from both target platforms looked identical to the human eye, and the SSIM results verified this. In these examples, the per-pixel results also indicated a high level of similarity but indicated that the output images were not completely identical (see Table 4.5). While Figure 4.18a and Figure 4.18b appear identical, Figure 4.18c reveals that there are differences between the output from the two target platforms. With the difference slider used to exaggerate the differences, Figure 4.18c shows pixel differences highlighted in yellow and blue, allowing the human eye to detect the non-matching pixels, particularly when the difference is very small. These appear to be at the hard edges of the polygons that make up the 3D model. The same result can be seen in Figure 4.19.

### Knowledge Contribution

Following Iteration 4, I presented the research work to date in an AUT *Software Engineering Research Laboratory* (SERL) Seminar (see Appendix C). This allowed me to present the proposal and research problem, followed by the artefact iterations built thus far, and rendering and test comparison output from these artefacts. Researchers interested in computer graphics, software engineering, and automated testing attended the seminar which included a question and answer session. Fellow researchers were able to critique the work, and offer insights that could support the future direction of the project. The seminar was a platform for the validation of the progress of the research project to date as I was provided with an opportunity to justify the decisions I had made about the project so far, including the choice of research methodology and the selection of target platforms. There was also an opportunity to discuss the potential research contributions the project could make to the field, and seminar attendees were able to confirm that the project was moving in an encouraging direction.

#### 4.4.5 Iteration 5

In Iteration 5 the automated testing tool was further developed to allow output from any number of target platforms to be compared against each other in all possible combinations. Whereas the previous iteration of the tool (see Section 4.4.2) would only compare the last two platform output tabs clicked, the new version of the tool provided a matrix displaying platform versus platform results for each type

of test. All tests were run at the launch of the automated tool. Each test type — RGB correct, RGB incorrect, and SSIM — was presented in a separate tab, inside of which each test scene was presented in its own tab, allowing the tester to select a test type and test scene to view a comparison matrix of results for all target platforms for the selected combination. SSIM results were presented as both a value between 0 and 1, and the equivalent percentage. When clicking on an individual result within the results matrix, the tool would present the related target platforms’ graphical output side-by-side. Examples of this can be seen in Figure 4.20 and Figure 4.21.

As with the previous iteration of the tool, a difference image was generated for the selected test type, in this case by clicking on the RGB Diff button (see Figure 4.22). An Overlay/Flip button was added to the automated comparison tool to give the tester the ability to view overlaid platform outputs and toggle between them. This can be seen in Figure 4.23. Buttons were also added that allowed image scaling, providing the tester with the ability to zoom in and out on a test scene’s output to further inspect defects detected by the automated tool.

## Evaluation

Iteration 5 successfully allowed any number of test scene outputs from multiple target platforms to be loaded and compared. Result matrices presented an overview of comparisons across all platforms in a single view. Any results that appeared anomalous could be identified and then visually inspected by clicking on the corresponding cell in the results matrix. The tester could toggle between test scene results, as well as test types, however this still had to be conducted manually. With a small number of test scenes the user would be able to select each test scene to review the results, however with a large number of test scenes this would become time-consuming. A future suggestion would be for the tool to be able to automatically highlight the tabs containing test scenes with results of potential interest, reducing the burden on a human tester.

The addition of the Overlay/Flip button helped to identify disparities between platform outputs. While any graphical disparity would be highlighted by the difference image, when combined with the Overlay/Flip function it further aids in the identification of mismatched output by helping to train the eye on the defect. For example, a single pixel mismatch would be difficult to detect in the test scene output, but the difference image would detect the single pixel, and toggling the overlaid images would further highlight its location in the original test scene outputs by appearing to flash in and out of the images. The tester could then zoom in on the images to identify the exact location of any difference. A future suggestion would be for further automation to highlight the areas of difference on the test scene output images to remove the need for a human tester to toggle the images. For example, test scene outputs could be augmented with additional graphical elements to pinpoint potential areas of

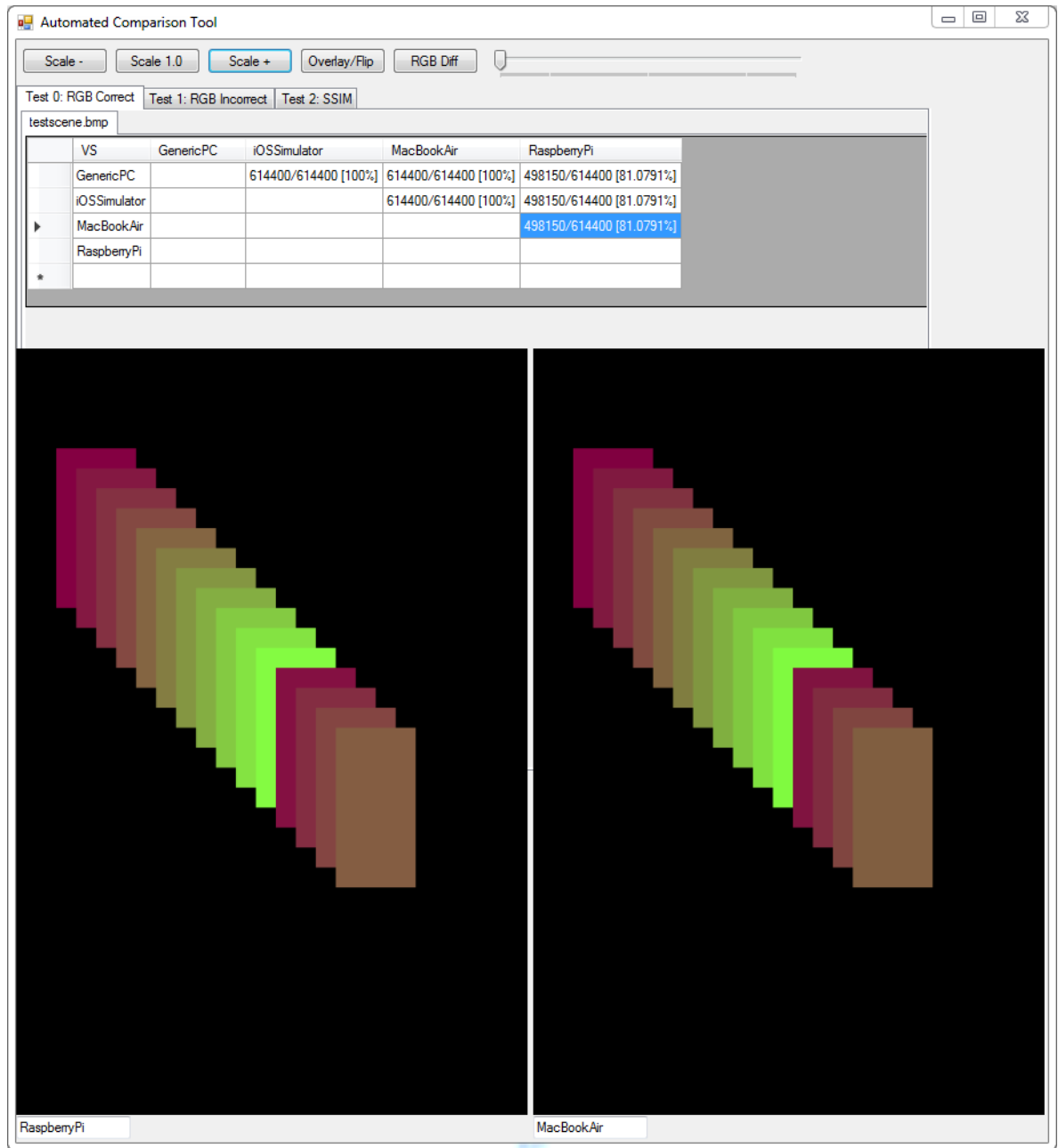


Figure 4.20: Iteration 5: Example of test results for cross-platform output generated by the Iteration 3 artefact.

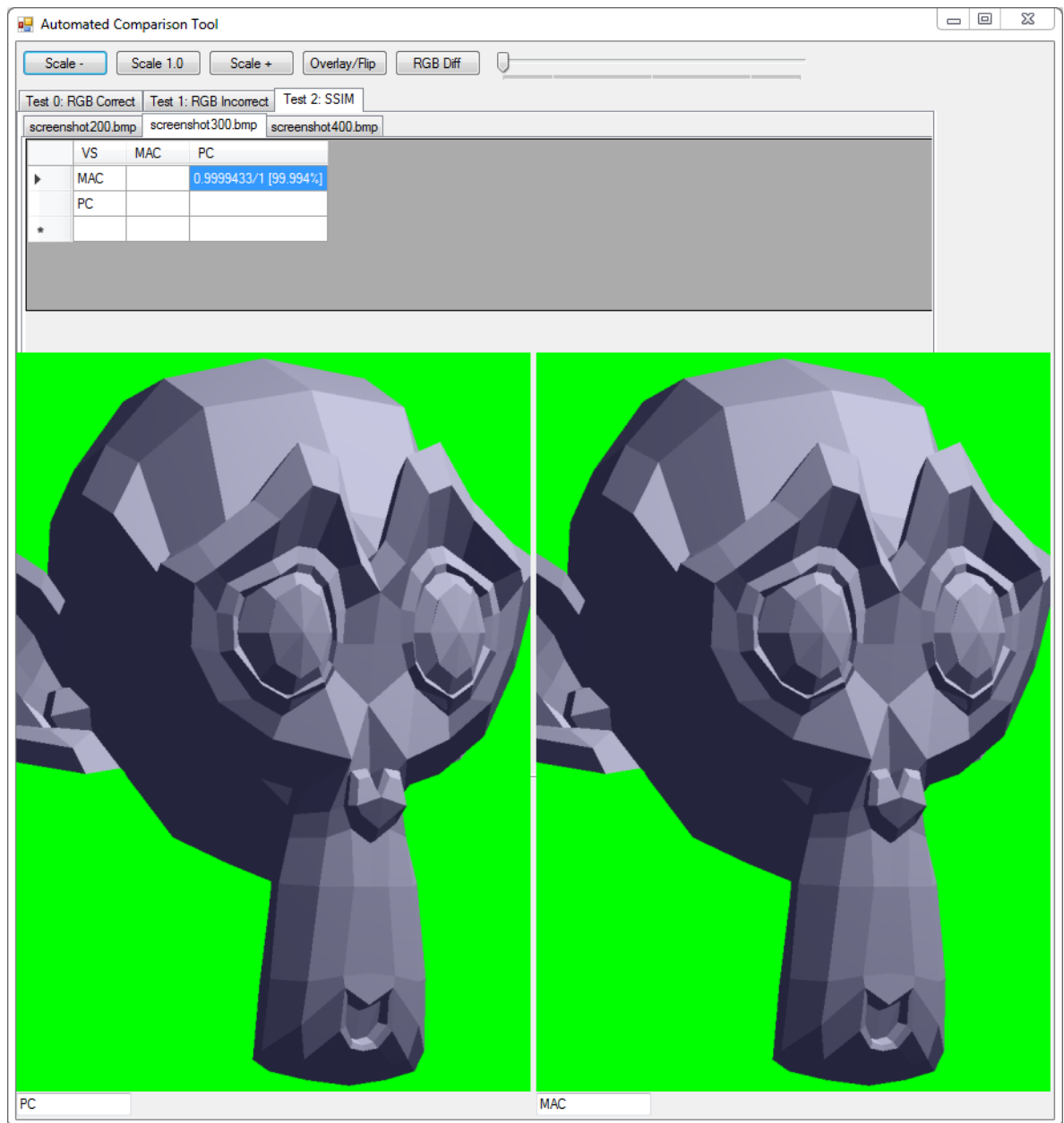


Figure 4.21: Iteration 5: Example of test results for cross-platform output generated by the Iteration 4 artefact.

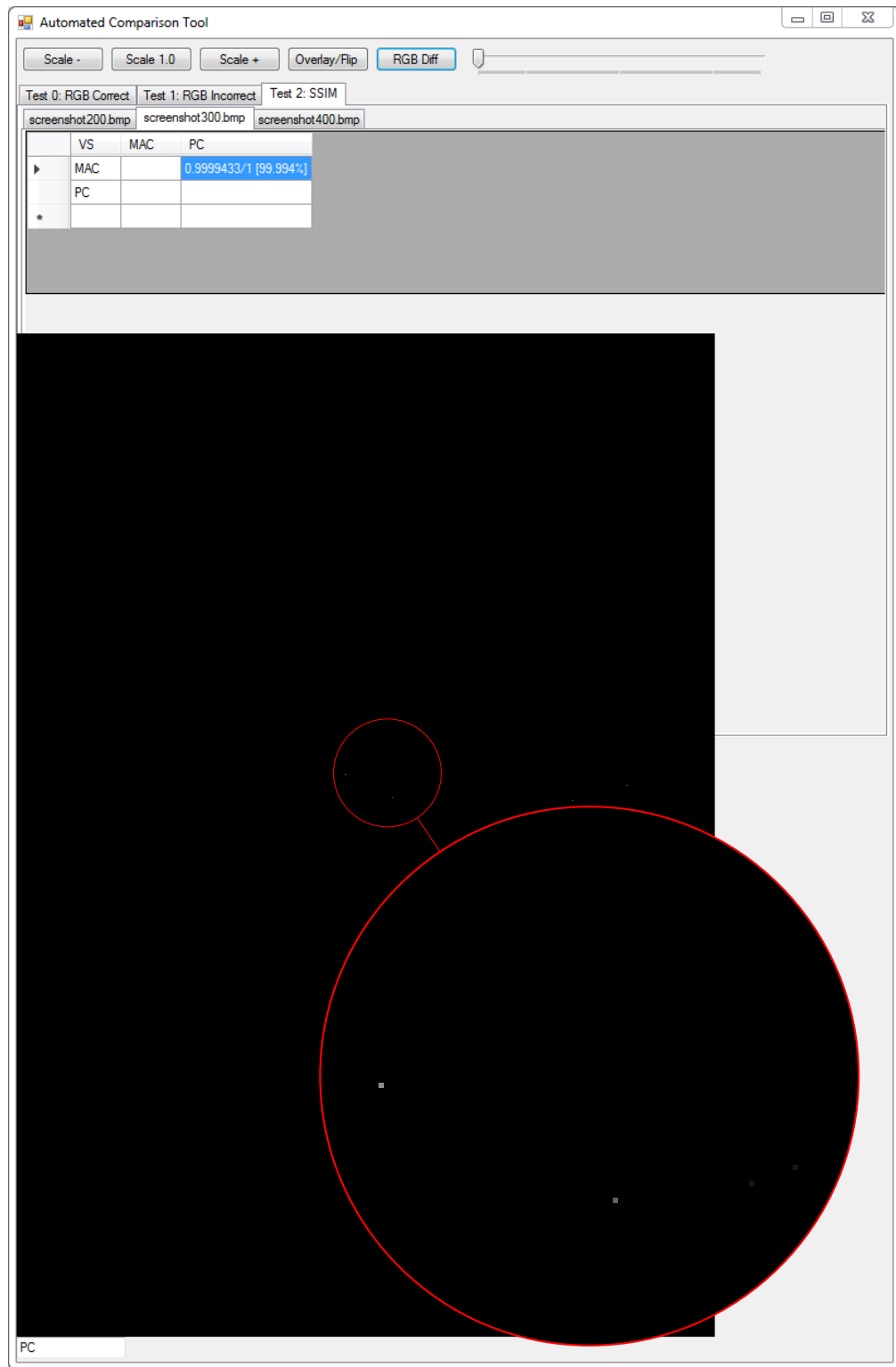


Figure 4.22: Iteration 5: Example of the difference image for cross-platform output generated by the Iteration 4 artefact.

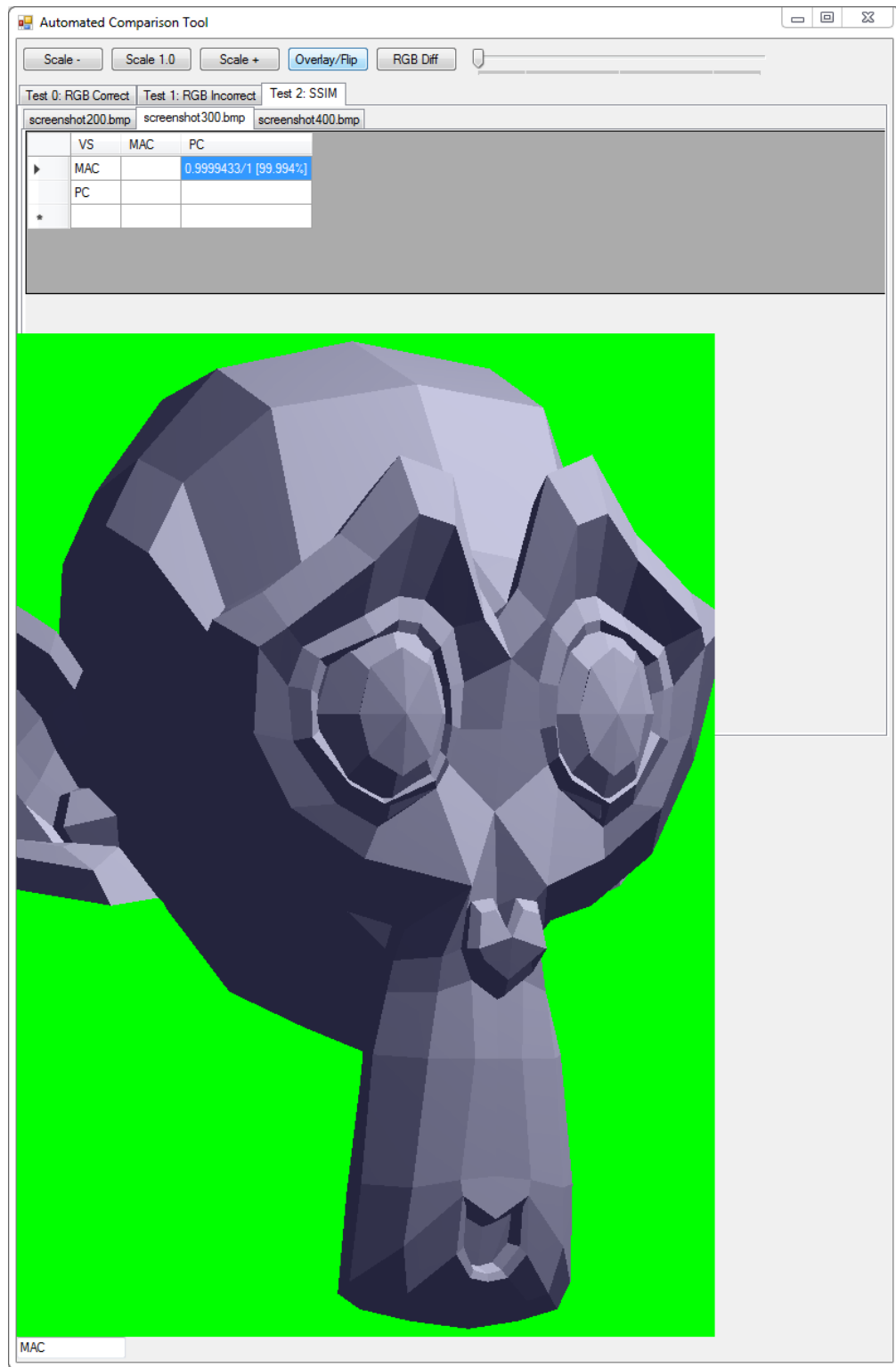


Figure 4.23: Iteration 5: Example of the Overlay/Flip feature for cross-platform output generated by the Iteration 4 artefact.

difference.

As the number of platforms and test scenes increased, the time to execute the comparison tests also increased. This was a reasonable amount of time for a small number of platforms and test scenes, but if more types of tests, target platforms or test scenes were added, then the time to execute could consume a significant amount of time.

#### 4.4.6 Iteration 6

The evaluation of Iteration 4 led me towards two possible further iterations. From this iteration, it would be possible to further develop the 3D capabilities of the 3D rendering artefact, adding, for example, texturing, alpha blending, character animation and physics simulation, which would further simulate complex 3D game features, as would be expected from a AAA game. Alternatively, it would be possible to develop more game-like scenarios in 2D, to explore the deterministic aspects of cross-platform game scenarios and simulations. I decided to concentrate on the 2D pathway to avoid a disproportionate focus on artistic requirements, such as creating 3D assets. Potential knowledge gained from further 2D work could then still be applied to 3D game development.

Iteration 6 targeted the *Generic PC*, *MacBook Air* and *Raspberry Pi* platforms used in Iteration 3 (see Table 4.4). To simulate cross-platform, game-like 2D rendering, as seen in *FTL: Faster Than Light* (Subset Games, 2012) and *N++* (Metanet Software, 2015), a C++ abstraction of the SDL middleware was further developed. This rendering abstraction (see Figure 4.24) was constructed to feature a line-based rendering system, known as *TurtleWorld*, and a sprite-based rendering system, known as *RobotWorld*. RobotWorld assets (Skene, 2014) can be seen in Figure B.1 in Appendix B. The rendering artefact supported texture loading, alpha blending and transparency, procedural animation, and text rendering which could be used to trace the state of the simulation as it executed. Both worlds could be driven by hardcoded C++ program code to create animated test scene simulations, which could then be played back on each target platform. A fixed time-step was used for animation. The rendering artefact developed in this iteration simulated a more complete feature-set than previous iterations, equivalent to the graphics rendering capabilities of a commercial cross-platform, 2D game.

As the artefact executed on the target platform, test scene output was generated for each fixed time-step, capturing every frame of animation. This differed from the test scene output in Iteration 4, where captures were taken every 100 frames. The framebuffer output captured each animation state change to file, which could then be retrieved for each platform. Examples of the test scene output can be seen in Figure 4.25 and Figure 4.26.

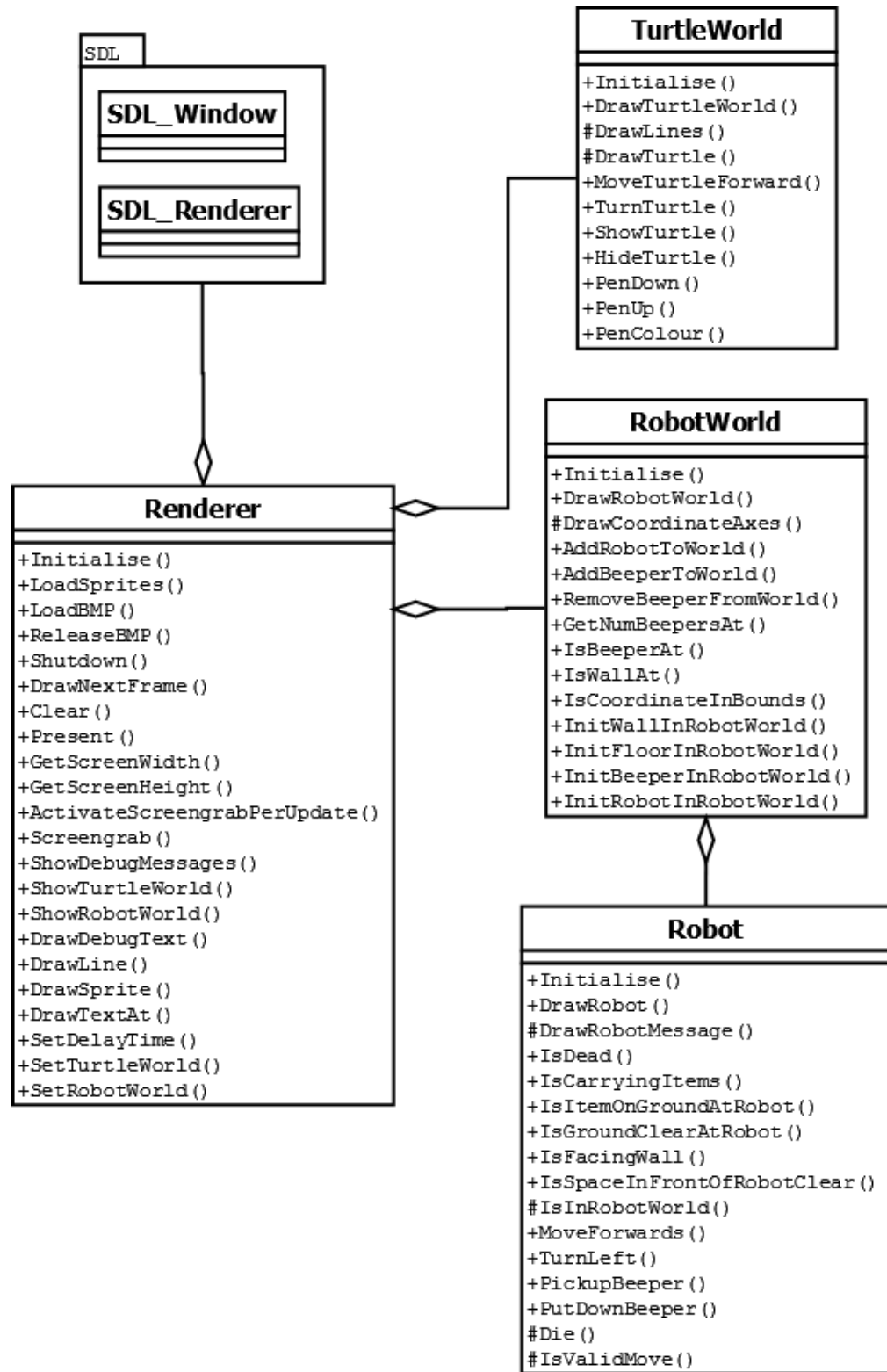


Figure 4.24: Iteration 6: Test scene generation: TurtleWorld and RobotWorld.



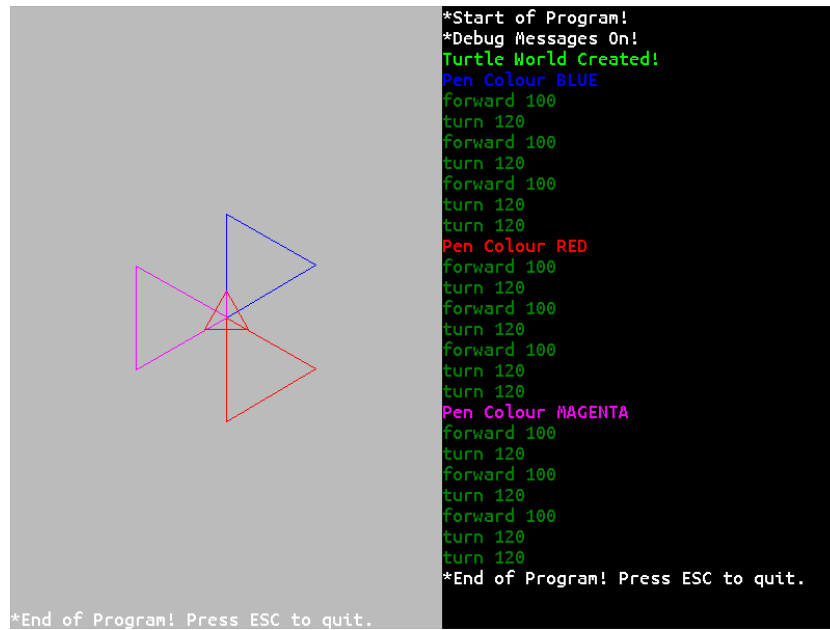


Figure 4.25: Iteration 6: Example of cross-platform output from the line-based TurtleWorld, with state tracing text rendering.

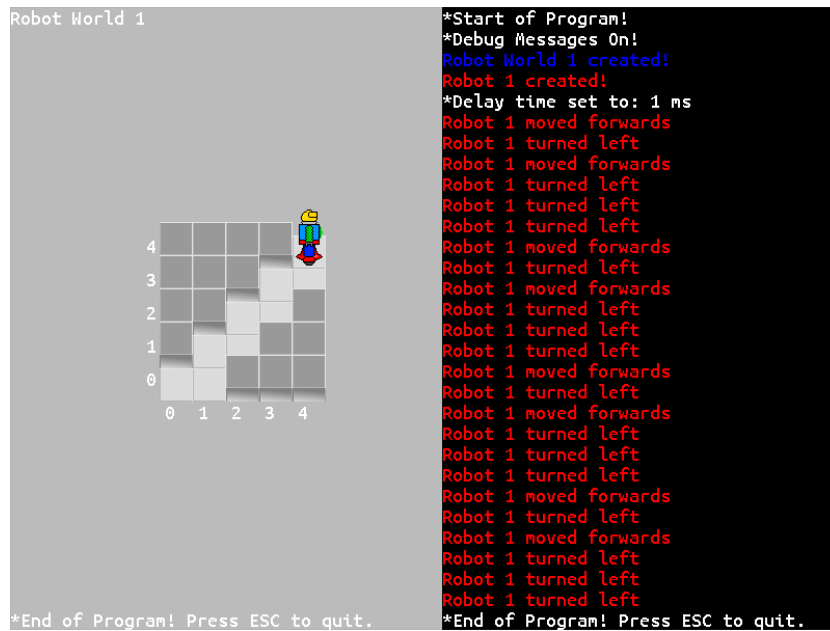


Figure 4.26: Iteration 6: Example of cross-platform output from the sprite-based RobotWorld, with state tracing text rendering.

## Evaluation

The cross-platform rendering artefact was able to generate a large number of procedurally animated test scenes on multiple platforms. This output data could be utilised by the automated comparison tool artefact. Iteration 6 demonstrated that adding 2D rendering features, similar to a commercial game, was feasible using SDL and the three target platforms. It was possible to have all graphics features successfully render on each target.

Developing using the SDL middleware on the *Raspberry Pi* Linux-based target was significantly slower than the *Generic PC* and *MacBook Air* targets. When compiling the SDL middleware, a rebuild on the *Raspberry Pi* would take in excess of 30 minutes. This meant that configuration of SDL for this Linux platform was time-consuming when compared to the other platforms, and it is therefore not a suitable platform for primary development. Retrieval of test scene output from the *Raspberry Pi* was achieved via USB memory stick transfer. As discussed for Iteration 3, the manual retrieval of this data could be automated through a network-based system.

The success of the hardcoded C++ procedural animation system led directly to the development of Iteration 7, which included a real-time game loop, whereby dynamic game scenarios with user input further simulated 2D game scenarios.

### 4.4.7 Iteration 7

In Iteration 7, a clone of the game *Space Invaders* (Taito Corporation, 1978) was designed and developed. This created a cross-platform, dynamic, real-time 2D game capable of executing on the *Generic PC* and *MacBook Air* target platforms (see Table 4.4). The *Raspberry Pi* target was eliminated in this iteration due to the time-consuming nature of development on the platform, as highlighted by Iteration 6. Additional C++ classes were developed to encapsulate aspects of the game. The *Game* class managed the real-time game loop, including the algorithms which controlled input handling, game simulation processing and world updating, and 2D scene rendering. Texture assets were able to be loaded from file and managed in memory by a resource manager class, the *TextureManager*, which interacted with the 2D renderer. An object hierarchy of game entities was created to simulate the desired gameplay features, including a human playable spaceship capable of shooting projectiles, enemy spaceship objects that had basic AI routines that reacted to collisions with player bullets, and explosion animations featuring alpha blending. An overview of the design can be seen in Figure 4.27.

The *Game* class was instrumented to capture framebuffer output from the game application at fixed-rate, one-second intervals. The rate of framebuffer capture was able to be configured in the C++ source code. Drawing from the prior works of Dunnett (2010), Grønbaek and Horn (2009) and Ostrowski and Aroudj (2013), and inspired by the Dickinson (2001) game engine replay system,

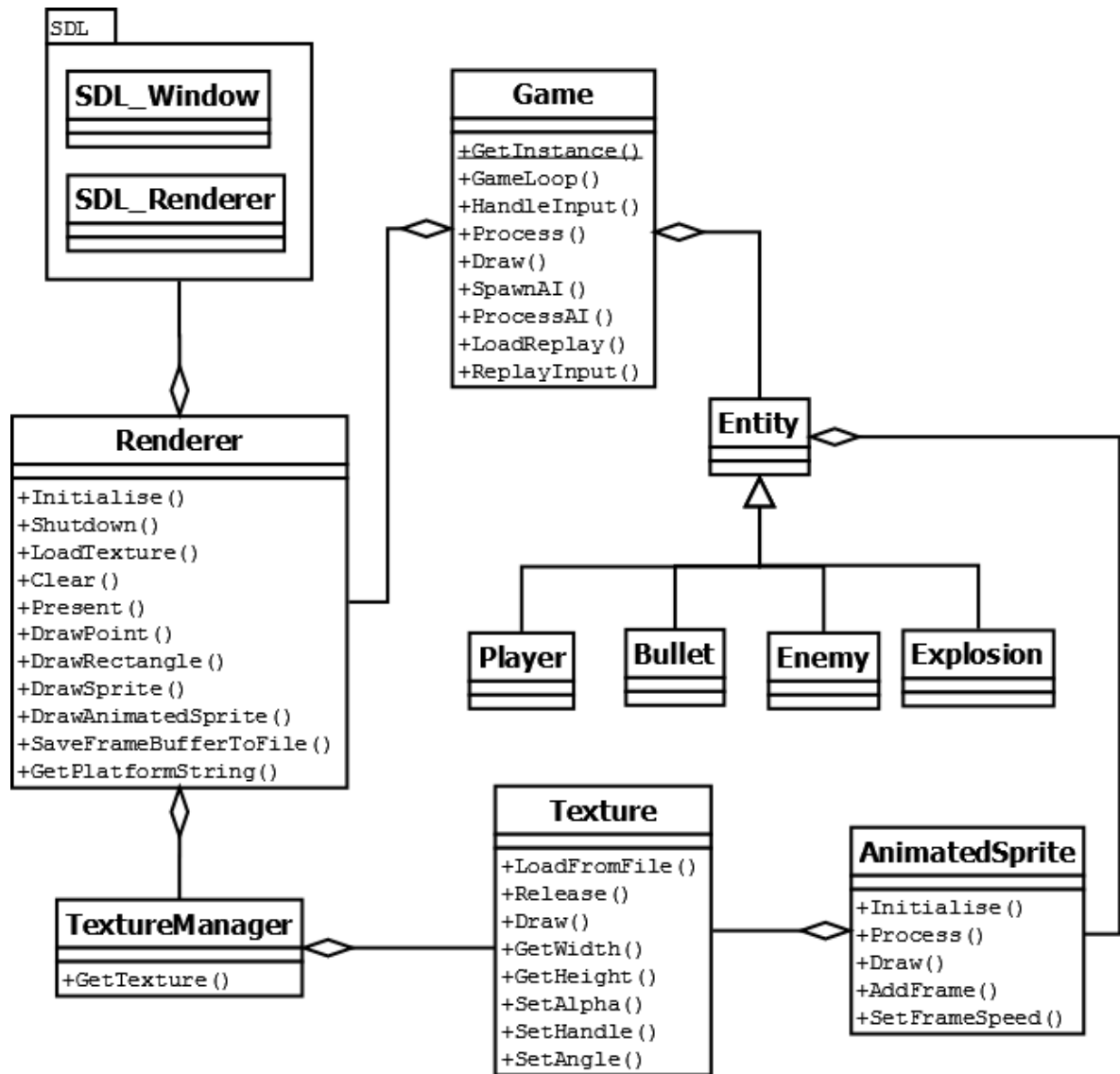


Figure 4.27: Iteration 7: Game clone with replay system.

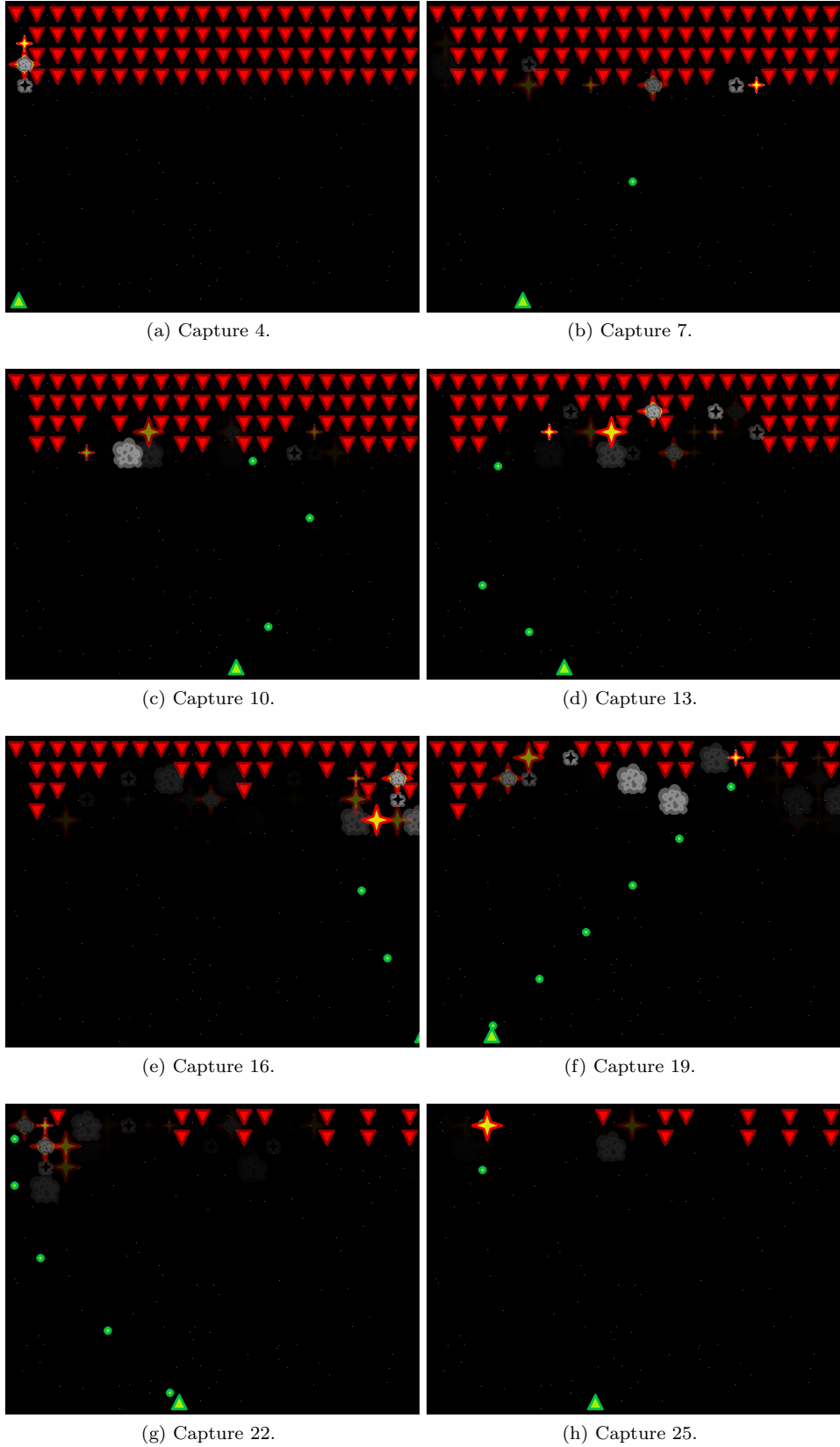


Figure 4.28: Iteration 7: Example replay captures, showing every third, one-second framebuffer capture from the *Generic PC* build target.

Iteration 6 also included a replay system, where the user's input could be recorded to system memory during gameplay, and subsequently saved to a file on disk. All player events, such as *move left*, *move right*, and *fire projectile*, were instrumented in the game's C++ source code to facilitate the recording functionality. Each instrumented event was time-stamped as it occurred. Pseudo-Random Number Generator (PRNG) seed values were also stored to disk, allowing for the same deterministic simulation to be re-run on subsequent replays.

Upon execution of the game, if a captured replay data file was found on disk, the game application would load the list of replay events into system memory. The manual player input system would be disabled and the game loop would then use the replay event list to trigger in-game player events based upon the time-stamped information for each event. A replay file generated on one platform could be loaded by the other cross-platform target, effectively replaying the previously saved playthrough, and ensuring that the same events occurred at precisely the same frames on both platforms. Example replay system output can be seen in Figure 4.28. Output was then loaded into the tool from Iteration 5.

## Evaluation

The Iteration 7 artefact allowed a human tester to play the game once and have the player events recorded. This meant that the game did not need to be played multiple times to generate test scene output for each target platform, and ensured consistency, since the same user events were executed on each platform. Framebuffer captures were collected at regular intervals, and could then be run through the automated comparison tool. This removed the human variable, showing that the type of human play testing described by Dunnett (2010) could be automated, reducing the reliance on human testers for each target platform.

In testing the initial implementation of Iteration 7 with the Iteration 5 automated comparison tool, a graphical defect was discovered (see Figure 4.29). When viewing the SSIM test results, it became apparent that the graphical output from the two platforms was not the same. While the RGB correctness test results showed a very high percentage of correct pixels, the SSIM results showed a more substantial difference of around 4% between the two target platforms. These results are shown in Table 4.6.

Using a combination of the Overlay/Flip feature and difference images it was possible to visually detect that the stars in the background were not the same on both target platforms (see Figure 4.30). RGB correct and incorrect results showed the images were almost identical because the stars were small when considered in the context of the entire scene. The SSIM results showed a 4% difference between the platforms, highlighting that there were structural differences — the stars were in different

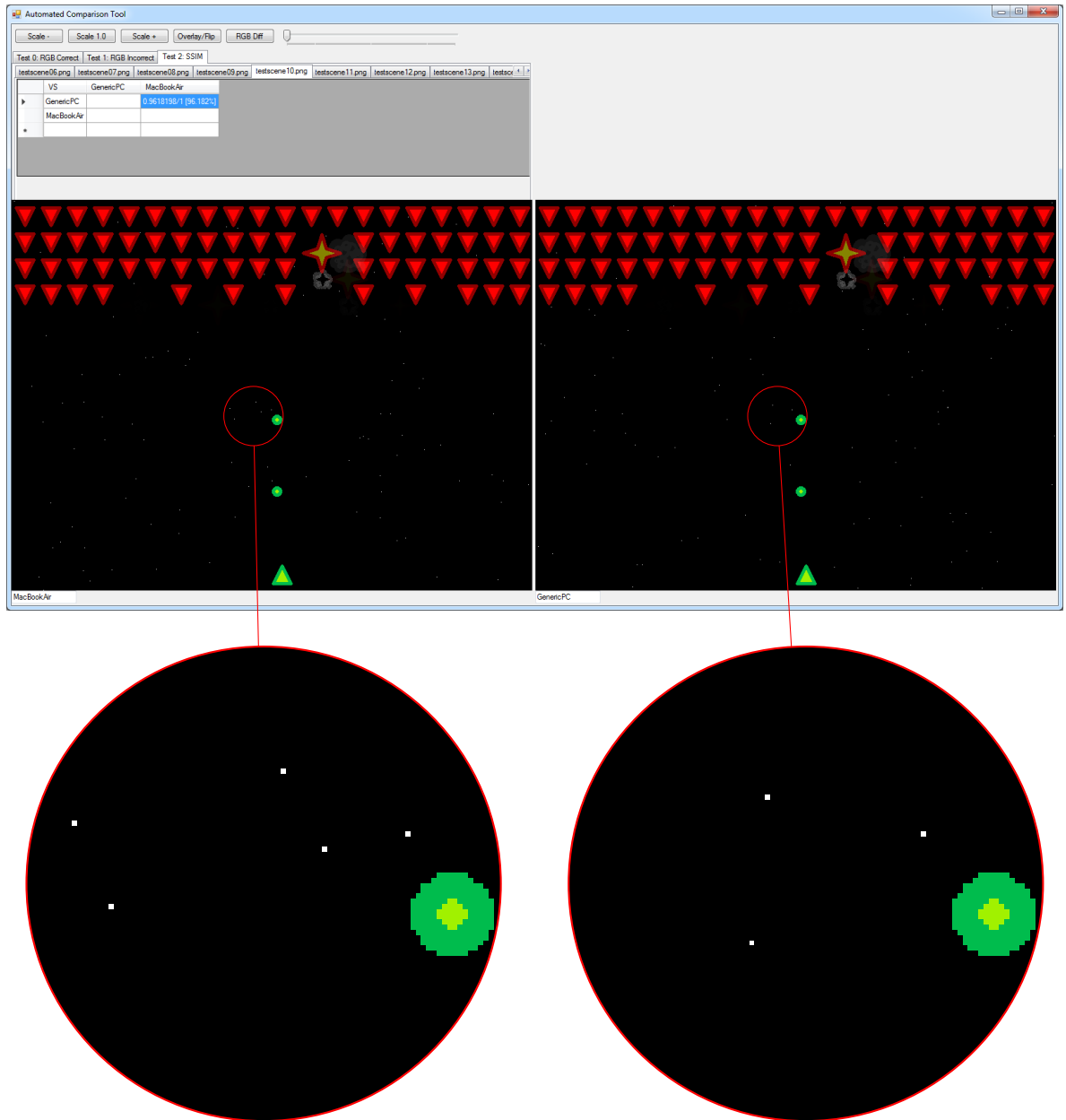


Figure 4.29: Iteration 7: Example of cross-platform output featuring detection of graphical defect.

<i>Generic PC vs MacBook Air</i>	RGB Correct	RGB Incorrect	SSIM
Test Scene 01	99.9625%	0.0375%	96.420%
Test Scene 02	99.96271%	0.03729167%	96.435%
Test Scene 03	99.96271%	0.03729167%	96.430%
Test Scene 04	99.96083%	0.03916667%	96.436%
Test Scene 05	99.9625%	0.0375%	96.386%
Test Scene 06	99.96188%	0.038125%	96.376%
Test Scene 07	99.96021%	0.03979167%	96.189%
Test Scene 08	99.93604%	0.06395833%	96.186%
Test Scene 09	99.96083%	0.03916667%	96.192%
Test Scene 10	99.96062%	0.039375%	96.182%
Test Scene 11	99.95729%	0.04270833%	96.207%
Test Scene 12	99.96%	0.04%	96.192%
Test Scene 13	99.96062%	0.039375%	96.165%
Test Scene 14	99.95937%	0.040625%	96.191%
Test Scene 15	99.95979%	0.04020833%	96.159%
Test Scene 16	99.95896%	0.04104167%	96.115%
Test Scene 17	99.96%	0.04%	96.068%
Test Scene 18	99.96021%	0.03979167%	96.091%

Table 4.6: Iteration 7: Automated comparison tool results.





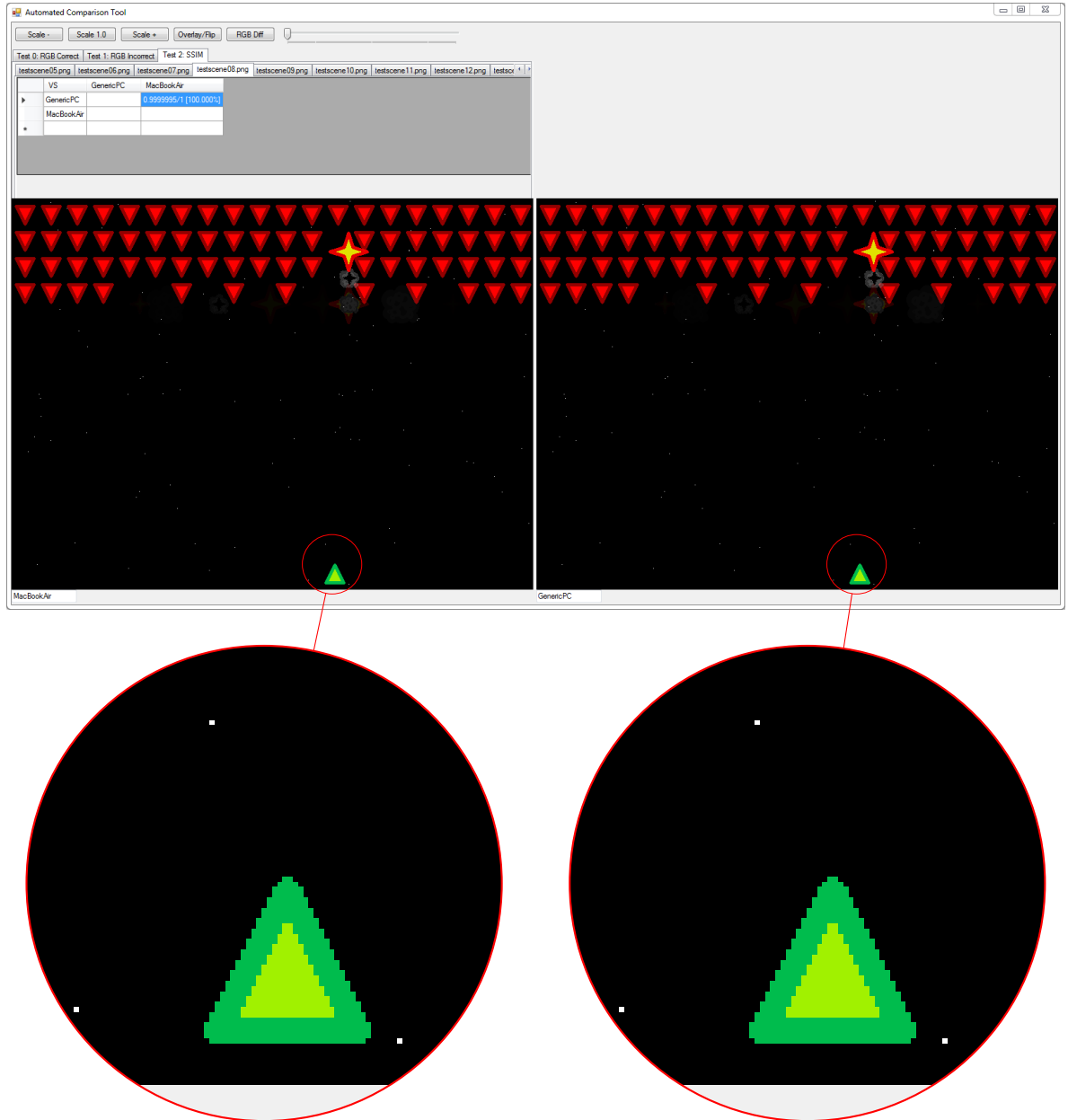


Figure 4.31: Iteration 7: Example of corrected cross-platform output, showing 100% SSIM result.

<i>Generic PC vs MacBook Air</i>	RGB Correct	RGB Incorrect	SSIM
Test Scene 01	100%	0%	100%
Test Scene 02	100%	0%	100%
Test Scene 03	100%	0%	100%
Test Scene 04	99.99854%	0.001458333%	100%
Test Scene 05	100%	0%	100%
Test Scene 06	100%	0%	100%
Test Scene 07	99.99958%	0.0004166667%	100%
Test Scene 08	99.97542%	0.02458333%	100%
Test Scene 09	100%	0%	100%
Test Scene 10	100%	0%	100%
Test Scene 11	99.99667%	0.003333333%	99.99998%
Test Scene 12	99.99937%	0.000625%	100%
Test Scene 13	100%	0%	100%
Test Scene 14	99.99875%	0.00125%	100%
Test Scene 15	99.99937%	0.000625%	100%
Test Scene 16	99.99854%	0.001458333%	100%
Test Scene 17	99.99958%	0.0004166667%	100%
Test Scene 18	99.99979%	0.0002083333%	100%

Table 4.7: Iteration 7: Automated comparison tool results, following defect repair.

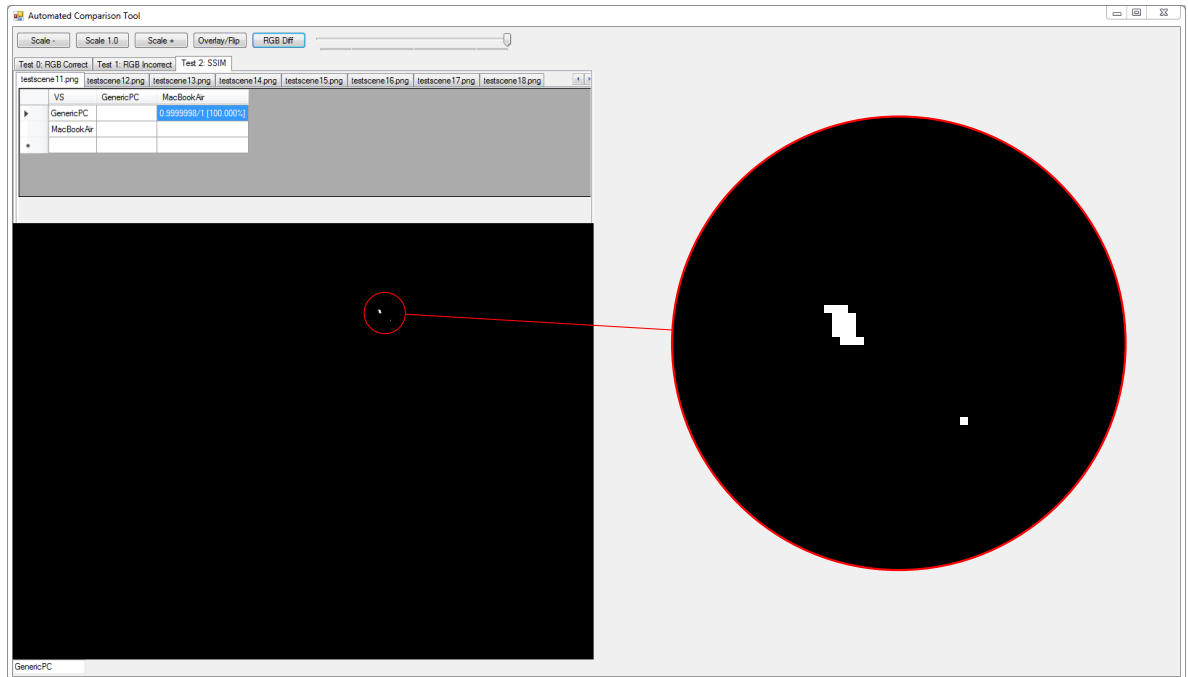


Figure 4.32: Iteration 7: Example of cross-platform output difference image for test scene 11.

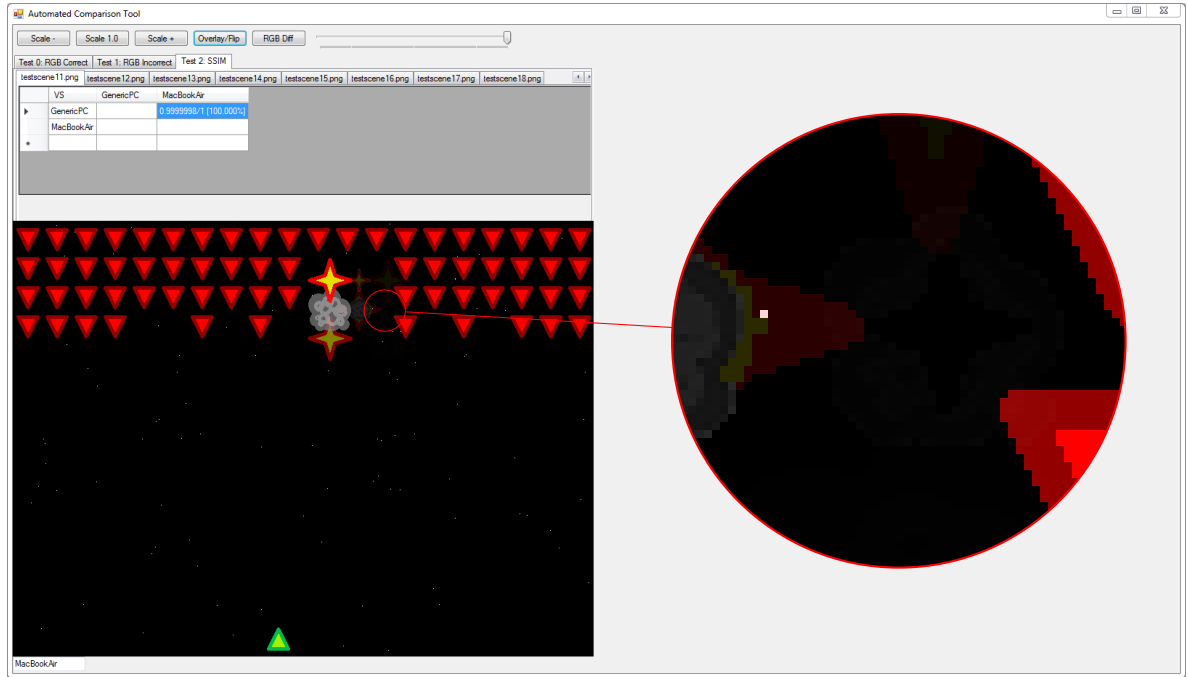


Figure 4.33: Iteration 7: Example of cross-platform output Overlay/Flip comparison for test scene 11.

the results graphically, showing the test results per scene in a graph which would highlight significant deviation between test scenes. The tool would then be able to highlight the affected tabs for quicker verification by a human taster.

### Knowledge Contribution

Following the completion of Iteration 7, I presented the research project to date at the *PikPok Developers' Conference* as an invited guest speaker (see Appendix D). PikPok is a Wellington-based game development studio, established in 1997, and the audience consisted of game development industry professionals. The presentation was followed by a question and answer session and discussion with PikPok developers, who were particularly interested in the issues I had encountered with deterministic behaviour and the replay system, as well as sharing their own experiences of cross-platform testing issues where UI elements appeared differently on different device resolutions. These interactions, along with the critique and feedback from industry professionals, indicated that I had reached a *good enough* DSR solution and could stop iterating and write up the research project results.

#### 4.4.8 Iteration 8

The knowledge gained from the iterative construction of the two artefacts allowed me to create a generalised model of a complete automated testing system, which could be applied to any cross-platform game development scenario, and shows not only the components necessary for the construction of such

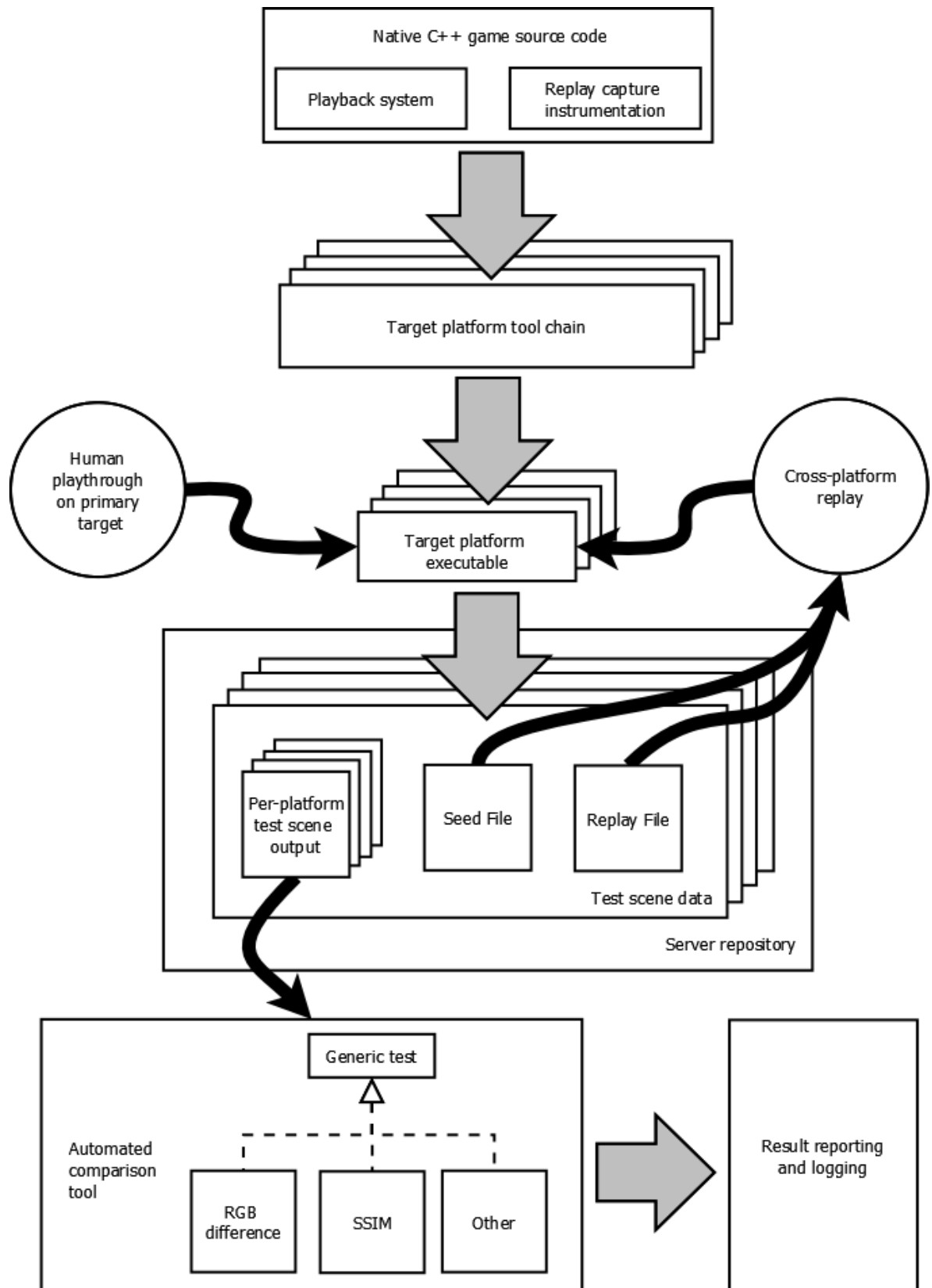


Figure 4.34: Iteration 8: Model: Automated testing and validation of computer graphics implementations for cross-platform game development.

a system, but the importance of the relationship between the two artefacts and how this contributes to the success of the automated test system for the verification of cross-platform computer graphics implementations. While the two artefacts are necessarily decoupled, to ensure consistency across all platforms, it is necessary to design and instrument the game implementation to facilitate automated testing. The model is shown in Figure 4.34.

## 4.5 Summary of Artefact Development and Evaluation

This chapter has detailed the DSR process undertaken in the iterative construction of two artefacts, and the corresponding results of the Development and Evaluation phases. A cross-platform test scene generation artefact was iteratively developed alongside an automated comparison tool artefact. Output from the test scene generation artefact at various iterations was able to be tested using the automated comparison tool, which was also iteratively developed throughout the research project. The combination of these building blocks and the various simulations led to an effective automated testing tool that was able to detect graphical defects. Finally, a model of a complete automated testing system was presented, based upon the knowledge gained through the DSR process. The following Discussion and Conclusion chapters map to the final Evaluation and Conclusion phases of the DSR project.

# Chapter 5

## Discussion

“It’s too risky to *not* do  
automated testing on a AAA  
game”

---

Francesco Carucci, AAA  
Automated Testing... For AAA  
Games (Carucci, 2009)

### 5.1 Overview

This chapter will present the major findings of the research project, summarising the results and contextualising them in relation to industry needs and related prior works. Limitations of the research project will then be discussed.

### 5.2 Summary of Results

The research project shows that the testing and validation of computer graphics implementations for cross-platform game development can be automated. Technological developments leading to the current state of video games are outlined in Section 1.5 and Section 2.2 provides examples of graphical defects in commercial video game software, highlighting the scale of the problem within the industry. Testing practices within industry are explored in Section 2.3 and further detailed in Appendix A. Technologies and techniques used to develop cross-platform games are discussed in Section 3.4 and are applied in the construction of the artefacts described in Chapter 4. Finally, the automation of quality assurance of visual defects in cross-platform video games is achieved through the DSR process, culminating in a generalised model, as described in Section 4.4.8.

The research project combines aspects of prior research and mimics testing techniques being utilised in the development of game engines and AAA games. The aspects existed as single elements in prior research, outlined in Chapter 2, however they were not combined or used in a cross-platform manner. The artefacts developed in this project feature game event instrumentation for record and playback, test scene generation and output capture, and SSIM and RGB per-pixel comparison methods, with a bespoke tool for reporting automated test results. The artefacts are able to successfully generate simulated cross-platform game output which can be tested to ensure correct implementation exists across all target platforms — the game plays the same no matter what the underlying architecture. This verifies that it is possible to build such a complete system, on a small scale, with limited resources. To be successful, not only does image comparison need to be utilised, but a game must be designed to allow for deterministic, and hence replayable, functionality to be instrumented. The automated comparison tool would not function without appropriate output from each target platform to test for correctness.

The resulting model is reusable and can be applied to any game scenario. This contrasts with systems such as those used at Unity Technologies and the VTK system, that are highly coupled to their implementations and therefore not easily transferable or reusable. The model could be integrated early in the game production cycle, for example well before the completion of the Alpha phase, allowing a cross-platform game to be thoroughly tested without needing to utilise a large number of human testers for multiple platforms. This could reduce development time and the necessity for crunch time, supporting the conclusions of Buhl and Gareeboo (2012) and Donat et al. (2014). Whereas Unity Technologies use small, isolated test scenes for different graphics features within their engine (Pranckevicius, 2007), this model allows a game scenario to be tested without the need to individualise particular graphics features, such as alpha blending, shadows or particles.

The nature of the automated test replay allows for regression testing to occur. Previously generated replays can be re-run on any target platform and the resulting outputs captured and compared. Again, this ensures that the game implementation continues to function as expected, despite further work being undertaken during game production. This is similar to the experiences of Yee and Newman (2004), who reported that automated testing of graphical output in film production helped ensure the quality of their product was not compromised as their development environment was upgraded with OS and tool chain changes. Using the model, the same can be applied to cross-platform game development, including changes to not only the tool chain, but also API, graphics driver and OS. Since the game industry must accommodate new and emerging technologies on a regular basis, it would therefore be less risky to develop for new platforms, integrate new graphics hardware or develop cross-generational games, since the system model would allow for the verification of target platform output.

A consideration when undertaking a research project of this type is that updates to the OS, drivers, APIs and tool chains on test target platforms may impact on the development environment. It is therefore important to consider turning auto-updates off to ensure the integrity of the test environment until the completion of the research project.

The results of the research project agree with Petersen (2012) and Andersen (2013), who both stated that testing across multiple platforms increases complexity. The need to set up unique tool chains for each target platform, and to debug platform-specific problems, adds to the overall complexity of a computer graphics implementation for game scenarios.

While a human tester is still required to initially play through the game under test on the primary target, subsequent playthroughs can be automated using the system described. Iteration 7 (see Section 4.4.7) shows that this is possible for cross-platform games. The human tester validates that the game works as expected on the primary target, and the automated playback captures output to test that it works on other targets. As the human plays, they are able to check that the fun aspects of the game are present. It is not possible to automate this aspect of testing. However, while the human tester is playing, they are able to decide that the game is behaving as intended, selecting the golden truth images by approving their playback scenario for testing. Because the replay system can be used on all targets, the human effort to find errors in cross-platform output is reduced. The reporting from the automated comparison tool can then aid developers in discovering defects within the cross-platform target implementations. Developers are able to easily recreate issues during the replay, without the need for the tester to communicate how an issue arose or what was happening during the game when it occurred — the developer can see this themselves in the replay. Automation can help to identify areas to look at, potential problems, and graphical defects. These issues might otherwise be unnoticeable to the human eye, but the tool has an unrelenting eye for detail.

In agreement with Grønbæk and Horn (2009), comparison metric results in isolation may be difficult to interpret. In some cases, a small deviation in SSIM result could be significant, while in others it may not be. Results presented over time can help to show deviations from the norm. Currently this has to be manually tabulated, however as discussed in the evaluation of Iteration 7, the tool could graph this information in a visual way, and alert to anomalous results by identifying affected test scene tabs. The results of testing using the automated comparison tool do not clearly indicate what constitutes a failed test — individual fine-tuning of the comparison method tolerances would be required to more accurately identify a definite fail or pass. This may be unique to particular graphic styles and would need to be calibrated depending on the type of game implementation being tested.



## 5.3 Limitations

I acknowledge that there are limitations to the research project. The DSR process simulates aspects of game development, however this does not cover the entirety of developing a commercial game, AAA or otherwise. The replayable game is limited to 2D, rather than 3D, on target platforms available to me as sole researcher, and does not include common commercial game target platforms, such as home consoles. This is due to these proprietary consoles only being available under non-disclosure agreements to registered commercial developers. Graphical rendering is limited to the fixed-function pipeline (FFP), which is quick to develop, but does not reflect the additional complexity in the use of the programmable shader pipeline in current generation graphics implementations. Each iteration of the test scene generation artefact renders output to a framebuffer that is the same size for each target platform. However, cross-platform games may not necessarily use the same sized framebuffer for all targets due to different device capabilities, a challenge also highlighted by Zioma (2012) at Unity Technologies.

To generate test scene output, the game implementation must be instrumented to capture events of interest, ensuring the replay occurs the same on subsequent playthroughs. This means that the game must be built in a deterministic manner to ensure the replay system functions correctly. The coverage provided by a test replay is only as good as the initial playthrough — not all aspects of the game may be utilised, and there may be hidden bugs not encountered by the tester.

The automated comparison tool utilises RGB per-pixel and SSIM comparison methods. There are other methods that could be used which may also be useful for image comparison and testing for graphical defects. These include, but are not limited to, MSE (Mean Squared Error), PSNR (Peak Signal-to-Noise Ratio) (Wang et al., 2004), SSIM variants, such as MS-SSIM (*Multiscale SSIM*) and IW-SSIM (*Information Content Weighted SSIM*) (Wang & Li, 2011; Whittle, Jones, & Mantiuk, 2017) and other bespoke techniques that could be used for detecting rendering errors (Amann, Chajdas, & Westermann, 2012).

The number of test cases generated throughout the research project is small in comparison to the scale of commercial testing at Unity Technologies. Similarly, the number of build configurations and target platforms used by Unity Technologies significantly outnumbers the target platforms I had available to build and test with.

Network transfer of test scene output is not implemented and therefore not tested, but theoretically it makes sense that this feature would simplify data transfer from target platforms. I acknowledge that the addition of this feature may introduce new issues with network connectivity and performance. However, automating this aspect is necessary to enhance the overall automation of the test system, and it is therefore deemed a significant feature to be included in the model.

## Chapter 6

# Conclusion

“Milestones are significant points in development marked by the completion of a certain amount of work (a *deliverable*).”

---

Charles Schultz, Robert Bryant,  
Tim Langdell, Game Testing All  
in One (Schultz et al., 2005, p.  
94)

In this thesis I have shown that it is possible to automate the testing and validation of computer graphics implementations for cross-platform game development. I have presented an overview of the game development industry and the industrial problem of graphical defects occurring in commercially released games, alongside a brief history of the technological developments that have shaped the industry. Examples of graphical defects from commercially released games were shown, highlighting the scale and severity of the issue, and industry insight from companies such as Unity Technologies and Electoral Arts was presented along with prior academic research into the automation of testing for graphics in related fields.

Using the Design Science Research methodology, an initial suggestion was proposed based upon the awareness of the problem. This led to the iterative construction of artefacts creating building blocks and simulations to investigate and ultimately solve the core research question. Two artefacts were constructed and seven development iterations undertaken using industry-relevant cross-platform development tools — one for the generation of cross-platform graphical test scenes and one for the automated comparison testing of these outputs. Knowledge generated throughout the research project was shared with both research and game industry communities, and feedback integrated into subse-

quent iterations. The eighth and final iteration presented a generalised model of a complete system for the automation of testing and validation of computer graphics implementations for cross-platform game development. Such a system could be applied in industrial settings to aid in the detection of graphical defects within games prior to their commercial release.

## 6.1 Future Work

Future research could build upon the foundations set out by the results of this research project. Instrumentation of an existing cross-platform game could be undertaken using an open source project to generate graphical output for use with the automated comparison tool. Further graphical testing of 2D as well as 3D output, could be undertaken, as commercial games have a large variety of graphical styles and visual effects. This would allow a researcher to uncover further knowledge regarding the complexity of cross-platform graphical output and the game simulations they represent, as well as enable a researcher to enhance the comparison capabilities of the automated tool to account for the added complexity of output from a feature-complete game. Testing could be undertaken at other layers within the layered architecture of a cross-platform game, such as swapping just driver implementation while keeping other layers the same, potentially allowing developers to discover a wider range of graphical defects that they could then provide support or resolution for prior to the release of a game. More tool chains and platforms could be utilised in the testing of such a system. This could include older console systems that are past their commercial viability and therefore more available to researchers for use in development.

The model could be applied to real world industrial settings to explore its effectiveness in a live production environment, including whether it has benefits in a team development scenario. This could include applying the model to the day to day developer workflow, which makes use of version control and continuous integration. The test system could be set up to execute each time a continuous integration build is created, with automated test results then reported back to developers for analysis.

Finally, research into the automated testing of graphical output for virtual reality and augmented reality systems should be explored, as this is an emerging area of technology and development within the game industry.

# References

- 19acomst. (2014). [FTL: Faster Than Light, in-game character graphical bugs, image screen capture]. Retrieved from [http://s23.postimg.org/48z356ex7/messed\\_up\\_crew.png](http://s23.postimg.org/48z356ex7/messed_up_crew.png).
- 2K Boston and 2K Australia. (2007). *BioShock* [Video game]. Novato, CA: 2K Games.
- 2K Czech. (2010). *Mafia II* [Video game]. Novato, CA: 2K Games.
- Abrash, M. (1997). *Michael Abrash's graphics programming black book special edition*. Scottsdale, AZ: The Coriolis Group, Inc.
- [Age of Wonders III image screen capture]. (n.d.). Retrieved from <http://images.akamai.steamusercontent.com/ugc/439447804167058449/6899C417CBEF876F4B859036C9DE7F9DA41C596C/>.
- Ahmed, M. D., & Sundaram, D. (2011). Design science research methodology: An artefact-centric creation and evaluation approach. In *ACIS 2011 proceedings*. ACIS. Retrieved from <http://aisel.aisnet.org/acis2011/79>
- Akenine-Moller, T., Haines, E., & Hoffman, N. (2008). *Real-time rendering* (3rd ed.). Wellesley, MA: A K Peters, Ltd.
- Alistar, E. (2014, July 15). Unity's test automation team [Web log post]. Retrieved from <http://blogs.unity3d.com/2014/07/15/unitys-test-automation-team/>.
- Amann, J., Chajdas, M. G., & Westermann, R. (2012). Error metrics for smart image refinement. *Journal of WSCG*, 20(2), 127–136.
- AMD. (2016). PerfStudio [Computer software]. Sunnyvale, CA: AMD.
- Andersen, K. (2013, June 2). Runtime tests - Unity's runtime API test framework [Web log post]. Retrieved from <http://blogs.unity3d.com/2013/06/02/runtime-tests-unitys-runtime-api-test-framework/>.
- Asif, A. (2013, November 20). *Assassins Creed 4 errors, crashes, graphics, launch problems and fixes*. Retrieved from <http://segmentnext.com/2013/11/20/assassins-creed-4-errors-crashes-graphics-launch-problems-and-fixes/>.
- [Assassin's Creed IV: Black Flag image screen capture]. (n.d.). Retrieved from <http://images.akamai.steamusercontent.com/ugc/3280048718378410511/B8AA69BD7AA1E04DB7318464A719E7A01F32A92D/>.
- [Assassin's Creed Unity image screen capture]. (n.d.). Retrieved from [http://static2.gamespot.com/uploads/scale\\_super/1544/15443861/2732028-289650\\_screenshots\\_2014-11-11\\_00007.jpg](http://static2.gamespot.com/uploads/scale_super/1544/15443861/2732028-289650_screenshots_2014-11-11_00007.jpg).
- Bast, E. (2012). [FTL: Faster Than Light, blocky text graphical bugs, image screen capture]. Retrieved from [https://d37wxxhohlp07s.cloudfront.net/s3\\_images/787760/ftl.JPG?1346870704](https://d37wxxhohlp07s.cloudfront.net/s3_images/787760/ftl.JPG?1346870704).
- Bates, B. (Ed.). (2003). *Game developer's market guide*. Boston, MA: Premier Press.
- [Batman: Arkham Knight image screen capture]. (n.d.). Retrieved from <http://i.imgur.com/zJ08dJN.jpg>.

- [Battlefield 3 image screen capture]. (n.d.). Retrieved from <http://bf4central.com/wp-content/uploads/2013/10/bf3-neck.jpg>.
- [Battlefield 4 image screen capture]. (n.d.). Retrieved from <http://www.gameranx.com/img/15-Feb/b8qm1npiuaaage3.png>.
- Bay, C. (2016, December 21). Graphics tests, the last line of automated testing [Web log post]. Retrieved from <https://blogs.unity3d.com/2016/12/21/graphics-tests-the-last-line-of-automated-testing/>.
- Bell, A. (2013, December 19). *EA sued by shareholders over Battlefield 4 glitches*. Retrieved from <http://nzgamer.com/news/7121/ea-sued-by-shareholders-over-battlefield-4-glitches.html>.
- Bethesda Game Studios. (2011). *The Elder Scrolls V: Skyrim* [Video game]. Rockville, MD: Bethesda Softworks.
- Bethesda Game Studios. (2015). *Fallout 4* [Video game]. Rockville, MD: Bethesda Softworks.
- Bethke, E. (2003, January 25). *Game development and production*. Plano, TX: Wordware Publishing Inc.
- Bierbaum, A., Hartling, P., & Cruz-Neira, C. (2003). Automated testing of virtual reality application interfaces. In *Proceedings of the workshop on virtual environments 2003* (pp. 107–114). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/769953.769966> doi: 10.1145/769953.769966
- Blizzard Entertainment. (2016). *Overwatch* [Video game]. Irvine, CA: Author.
- Blue Byte & Related Designs. (2012). *Anno 2070 - Deep Ocean* [Video game]. Rennes, France: Ubisoft.
- Blue Byte & Related Designs. (2014). *Might and Magic Heroes Online* [Video game]. Rennes, France: Ubisoft.
- Bourg, D. M., & Seemann, G. (2004). *AI for game developers*. O'Reilly Media, Inc.
- Brady, M., Dima, A., Gebase, L., Kass, M., Montanez-Rivera, C., & Rosenthal, L. (1999). VRML testing: Making VRML worlds look the same everywhere. In *IEEE computer graphics and applications* (Vol. 19, pp. 59–67). IEEE. doi: 10.1109/38.749124
- Buhl, C., & Gareeboo, F. (n.d.). *Automated testing: A key factor for success in video game development. case study and lessons learned*. Retrieved from [http://www.uploads.pnsrc.org/2012/slides/t-26.Gareeboo\\_slides.pdf](http://www.uploads.pnsrc.org/2012/slides/t-26.Gareeboo_slides.pdf).
- Buhl, C., & Gareeboo, F. (2012). Automated testing: A key factor for success in video game development. case study and lessons learned. In *Thirtieth annual Pacific Northwest software quality conference* (pp. 545–559). Retrieved from [http://www.uploads.pnsrc.org/proceedings/PNSQC\\_2012.Proceedings\\_9\\_25.pdf](http://www.uploads.pnsrc.org/proceedings/PNSQC_2012.Proceedings_9_25.pdf)
- Burnes, A. (2014, May 26). *Watch Dogs graphics, performance and tweaking guide*. Retrieved from <http://www.geforce.com/whats-new/guides/watch-dogs-graphics-performance-and-tweaking-guide>.
- [Call of Duty: Modern Warfare 3 image screen capture]. (n.d.). Retrieved from <http://i.crackedcdn.com/phpimages/article/2/2/9/151229-v1.jpg>.
- Carson, S., van Dam, A., Puk, D., & Henderson, L. R. (1998, February). The history of computer graphics standards development. *SIGGRAPH Comput. Graph.*, 32(1), 34–38. Retrieved from <http://doi.acm.org/10.1145/279389.279434> doi: 10.1145/279389.279434
- Carucci, F. (2009, August 16). *AAA automated testing... for AAA games* [PowerPoint slides]. Retrieved from [http://www.crytek.com/download/AAA.Automated\\_testing.ppt](http://www.crytek.com/download/AAA.Automated_testing.ppt).

- casparcg.com. (n.d.). *CasparCG: Pro graphics and video play-out system*. <http://www.casparcg.com/>.
- Chandler, H. M. (2014). *The game production handbook* (3rd ed.). Burlington, MA: Jones & Bartlett Learning.
- Crytek. (n.d.). *CryENGINE 3 — Crytek*. Retrieved from <http://www.crytek.com/cryengine/cryengine3/overview/>.
- Crytek. (2007). *Crysis* [Video game]. Redwood City, CA: Electronic Arts.
- Crytek GmbH. (n.d.). CryENGINE [Game engine]. Retrieved from <http://cryengine.com/>
- Cullinane, J. (2014). *Ubisoft stock slips following Assassin's Creed Unity's buggy launch*. Retrieved from <http://www.gameplanet.co.nz/playstation-4/news/g5463c53604c32/Ubisoft-stock-slips-following-Assassins-Creed-Unitys-buggy-launch/>.
- DeLoura, M. (2011, September 6). *Game engines and middleware in the west*. Retrieved from <http://www.slideshare.net/markdeloura/game-engines-and-middleware-2011>.
- Demerjian, C. (2012, March 6). Developers talk cross-platform mobile games [Web log post]. Retrieved from <http://semiaccurate.com/2012/03/06/developers-talk-cross-platform-mobile-games/>.
- Dickinson, P. (2001, July 13). *Instant replay: Building a game engine with reproducible behavior*. Retrieved from [http://www.gamasutra.com/view/feature/131466/instant\\_replay\\_building\\_a\\_game\\_.php](http://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php).
- Donat, M., Husain, J., & Coatta, T. (2014, July). Automated QA testing at Electronic Arts. *Commun. ACM*, 57(7), 50–57. Retrieved from <http://doi.acm.org/10.1145/2617754> doi: 10.1145/2617754
- Dunnett, G. (2010, January 12). On web player regression testing [Web log post]. Retrieved from <http://blogs.unity3d.com/2010/01/12/on-web-player-regression-testing/>.
- EA Canada. (2009). *FIFA 10* [Video game]. Redwood City, CA: EA Sports.
- EA Canada. (2010). *FIFA 11* [Video game]. Redwood City, CA: EA Sports.
- EA Digital Illusions CE. (2011). *Battlefield 3* [Video game]. Redwood City, CA: Electronic Arts.
- EA Digital Illusions CE. (2013). *Battlefield 4* [Video game]. Redwood City, CA: Electronic Arts.
- EA Tiburon. (2007). *NCAA Football 08* [Video game]. Redwood City, CA: EA Sports.
- EA Tiburon and EA Canada. (2010). *NCAA Football 11* [Video game]. Redwood City, CA: EA Sports.
- EA Tiburon and EA Canada. (2011). *NCAA Football 12* [Video game]. Redwood City, CA: EA Sports.
- Eberly, D. (2005). *3D game engine architecture: Engineering real-time applications with wild magic*. San Francisco, CA: Morgan Kaufmann.
- Epic Games. (n.d.). Unreal Engine [Game engine]. Retrieved from <https://www.unrealengine.com/what-is-unreal-engine-4>
- Exho. (2014). [Garry's Mod image screen capture]. Retrieved from <https://www.youtube.com/watch?v=QhN43Ubj5FY>.
- Facepunch Studios. (2004). *Garry's Mod* [Video game]. Bellevue, WA: Valve Corporation.
- Fahy, R., & Krewer, L. (2012). Using open source libraries in cross platform games development. In *2012 IEEE international games innovation conference (IGIC)*. Rochester, NY: IEEE. doi: 10.1109/IGIC.2012.6329835
- [Fallout 4 image screen capture]. (n.d.). Retrieved from <http://i.imgur.com/zYxPR33.jpg>.

- Fell, D. (2001, February 26). *Testing graphical applications*. Retrieved from <http://www.embedded.com/print/4023288>.
- Flynt, J., & Salem, O. (2005). *Software engineering for game developers*. Boston, MA: Thomson Course Technology PTR.
- Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1997). *Computer graphics principles and practice* (2nd ed.). Lebanon, IN: Addison-Wesley.
- Garney, B., & Preisz, E. (2011). *Video game optimization*. Boston, MA: Course Technology PTR.
- Goodwin, S. (2005). *Cross-platform game programming*. Rockland, MA: Charles River Media, Inc.
- [Grand Theft Auto: San Andreas image screen capture]. (n.d.). Retrieved from <http://i.crackedcdn.com/phpimages/article/2/2/8/151228.jpg?v=1>.
- Gregory, J. (2009). *Game engine architecture*. Boca Raton, FL: A. K. Peters/CRC Press.
- Grønbaek, B., & Horn, B. (2008). *Investigation of elements for an automated test tool for virtual environments*. Retrieved from <http://www.brianhorn.dk/reports/forkreport.pdf>.
- Grønbaek, B., & Horn, B. (2009). *Development of a graphical regression test tool for use in 3D virtual environments* (Master's thesis, Syddansk Universitet). Retrieved from <http://www.brianhorn.dk/reports/masterreport.pdf>.
- Harbour, J. (2004). *Game programming all in one*. Boston, MA: Thomson Course Technology PTR.
- Hardwidge, B. (2009, August 18). *The problem with porting games*. Retrieved from <http://www.bit-tech.net/gaming/2009/08/18/the-problem-with-porting-games/>.
- Hartel, N. (2011, September 30). *Fallout: New Vegas*. Retrieved from [http://games.highdefdigest.com/41/fallout\\_new-vegas.html](http://games.highdefdigest.com/41/fallout_new-vegas.html).
- Hearn, D. D., & Baker, M. P. (2003). *Computer graphics with OpenGL* (3rd ed.). Prentice Hall Professional Technical Reference.
- Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004, March). Design science in information systems research. *MIS Q.*, 28(1), 75–105. Retrieved from <http://dl.acm.org/citation.cfm?id=2017212>.
- Hight, J., & Novak, J. (2008). *Game development essentials: Project management*. Clifton Park, NY: Thomson Delmar Learning.
- Hook, B. (2005). *Write portable code: An introduction to developing software for multiple platforms*. San Francisco, CA: No Starch Press, Inc.
- Hunt, A., & Thomas, D. (2000). *The pragmatic programmer: From journeyman to master*. Reading, MA: Addison-Wesley.
- id Software. (2011). *Rage* [Video game]. Rockville, MD: Bethesda Softworks.
- Iftikhar, Z. (2010, August 24). *Mafia II errors, crashes, ATI fix, PhysX fix, and graphics bugs*. Retrieved from <http://segmentnext.com/2010/08/24/mafia-2-errors-crashes-ati-fix-physx-fix-graphics-bugs/>.
- Infinity Ward. (2011). *Call of Duty: Modern Warfare 3* [Video game]. Santa Monica, CA: Activision.
- Infinity Ward. (2013). *Call of Duty: Ghosts* [Video game]. Los Angeles, CA: Activision.
- Ion Storm Austin. (2004). *Thief: Deadly Shadows* [Video game]. London, UK: Eidos Interactive.
- Irish, D. (2005). *The game producer's handbook*. Boston, MA: Thomson Course Technology PTR.
- Kanode, C. M., & Haddad, H. M. (2009). Software engineering challenges in game development. In *Proceedings of the 2009 sixth international conference on information technology: New generations* (pp. 260–265). Washington, DC, USA: IEEE Computer Society. Retrieved from <http://dx.doi.org/10.1109/ITNG.2009.74> doi: 10.1109/ITNG.2009.74
- Keith, C. (2010). *Agile game development with scrum*. Boston, MA: Addison-Wesley.

- Kelly, C. (2012). *Programming 2D games*. Boca Raton: CRC Press.
- Kenwright, B. (2014, June). *Technical requirement certification (TRC) game development*. Retrieved from <http://games.soc.napier.ac.uk/resources/trc.pdf>.
- KiranKumar, M., & Sundaresasubramanian, G. (n.d.). *Explore: Testing the game*. Retrieved from <https://www.infosys.com/IT-services/independent-validation-testing-services/Documents/test-games-users-perspective.pdf>.
- Kuechler, B., & Vaishnavi, V. (2008). On theory development in design science research: Anatomy of a research project. *European Journal of Information Systems*, 17, 489–504. Retrieved from <http://dx.doi.org/10.1057/ejis.2008.40>
- LaMothe, A. (2003). *Tricks of the 3D game programming gurus*. Indianapolis, IN: Sams Publishing.
- Lapidous, E., & Jiao, G. (1999). Optimal depth buffer for low-cost graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware* (pp. 67–73). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/311534.311579> doi: 10.1145/311534.311579
- Levy, L., & Novak, J. (2010). *Game development essentials: Game QA & testing*. Clifton Park, NY: Delmar.
- Lewis, C., & Whitehead, J. (2011a, September). Repairing games at runtime or, how we learned to stop worrying and love emergence. *IEEE Softw.*, 28(5), 53–59. Retrieved from <http://dx.doi.org/10.1109/MS.2011.87> doi: 10.1109/MS.2011.87
- Lewis, C., & Whitehead, J. (2011b). The whats and the whys of games and software engineering. In *Proceedings of the 1st international workshop on games and software engineering* (pp. 1–4). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1984674.1984676> doi: 10.1145/1984674.1984676
- Lewis, C., Whitehead, J., & Wardrip-Fruin, N. (2010). What went wrong: A taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games* (pp. 108–115). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1822348.1822363> doi: 10.1145/1822348.1822363
- Lionhead Studios. (2001). *Black & White* [Video game]. Redwood City, CA: Electronic Arts.
- Llopis, N. (2003). *C++ for game programmers*. Hingham, MA: Charles River Media.
- Lorensen, B., & Miller, J. (2001, August). Visualization toolkit extreme testing. *Computer Graphics*, 35(3), 8–11. Retrieved from <http://www.siggraph.org/publications/newsletter/issues/v35/v35n3.pdf>
- [Mafia II image screen capture]. (n.d.). Retrieved from <https://i.ytimg.com/vi/KYfD-IWQ1EU/maxresdefault.jpg>.
- Manton, R., & Wilson, S. (n.d.). *Certification for console games*.
- March, S. T., & Smith, G. F. (1995, December). Design and natural science research on information technology. *Decis. Support Syst.*, 15(4), 251–266. Retrieved from [http://dx.doi.org/10.1016/0167-9236\(94\)00041-2](http://dx.doi.org/10.1016/0167-9236(94)00041-2) doi: 10.1016/0167-9236(94)00041-2
- McShaffry, M., & Graham, D. (2013). *Game coding complete* (4th ed.). Boston, MA: Course Technology PTR.
- Metanet Software. (2015). *N++* [Video game]. Toronto, Canada: Author.
- Metro. (2011, November 14). *Bethesda admits Skyrim graphics problems on Xbox 360*. Retrieved from <http://metro.co.uk/2011/11/14/bethesda-admits-skyrim-graphics-problems-on-xbox-360-218812/>.
- Microsoft. (2008). PIX [Computer software]. Redmond, Washington: Microsoft.



- Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014, June). Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In *Proceedings of the 36th international conference on software engineering (ICSE 2014)*. ACM. Retrieved from <http://research.microsoft.com/pubs/210047/murphyhill-icse-2014.pdf>.
- Myers, G., Badgett, T., & Sandler, C. (2012). *The art of software testing* (3rd ed.). John Wiley & Sons, Inc.: Hoboken, NJ.
- Nantes, A., Brown, R., & Maire, F. (2008). A framework for the semi-automatic testing of video games. In C. Darken & M. Mateas (Eds.), *Aiide*. The AAAI Press. Retrieved from <http://dblp.uni-trier.de/db/conf/aiide/aiide2008.html#NantesBM08>
- [NBA 2K13 image screen capture]. (n.d.). Retrieved from [http://www.nba-live.com/wp-content/uploads/2013/04/nba2k13\\_wade\\_clipping.png](http://www.nba-live.com/wp-content/uploads/2013/04/nba2k13_wade_clipping.png).
- NOWGamer.com. (2014). *Assassin's Creed: Unity framerate is Atrocious*. Retrieved from <http://www.nowgamer.com/assassins-creed-unity-framerate-is-atrocious/>.
- nVidia. (2016). Nsight [Computer software]. Santa Clara, CA: nVidia.
- Obsidian Entertainment. (2010). *Fallout: New Vegas* [Video game]. Rockville, MD: Bethesda Softworks.
- O'Leary, Z. (2004). *Essential guide to doing research*. London: SAGE Publications Ltd.
- Ostrowski, M., & Aroudj, S. (2013). Automated regression testing within video game development. *GSTF Journal on Computing (JoC)*, 3(2). Retrieved from <http://dx.doi.org/10.7603/s40601-013-0010-4> doi: 10.7603/s40601-013-0010-4
- [Overwatch, higher graphics setting, image screen capture]. (n.d.). Retrieved from <http://i.imgur.com/vk91MIq.png>.
- [Overwatch, low graphics setting, image screen capture]. (n.d.). Retrieved from <http://i.imgur.com/MKzMJwD.png>.
- Patton, R. (2001). *Software testing*. Indianapolis, IN: Sams Publishing.
- Peddie, J. (2013). *The history of visual magic in computers: How beautiful images are made in CAD, 3D, VR, and AR*. London, UK: Springer Verlag.
- Peffer, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007, December). A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, 24(3), 45–77. Retrieved from <http://dx.doi.org/10.2753/MIS0742-1222240302> doi: 10.2753/MIS0742-1222240302
- Petersen, T. (2012). Testing Unity [Web log post]. Retrieved from <http://blogs.unity3d.com/2012/05/08/testing-unity/>.
- Petrillo, F., Pimenta, M., Trindade, F., & Dietrich, C. (2008). Houston, we have a problem...: A survey of actual problems in computer games development. In *Proceedings of the 2008 ACM symposium on applied computing* (pp. 707–711). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1363686.1363854> doi: 10.1145/1363686.1363854
- Petrillo, F., Pimenta, M., Trindade, F., & Dietrich, C. (2009, February). What went wrong? A survey of problems in game development. *Comput. Entertain.*, 7(1), 13:1–13:22. Retrieved from <http://doi.acm.org/10.1145/1486508.1486521> doi: 10.1145/1486508.1486521
- Pfaff, G. (1984). Functional conformance testing of graphics software using a configurable reference system. In *Computers & graphics* (Vol. 8, pp. 29–37). Elsevier. doi: 10.1016/0097-8493(84)90021-9
- Phillips, L. (2002). *Game programming tricks of the trade*. Cincinnati, OH: Premier Press.

- Pranckevicius, A. (2007, July 31). Testing Graphics Code [Web log post]. Retrieved from <http://aras-p.info/blog/2007/07/31/testing-graphics-code/>.
- Pranckevicius, A. (2011, June 17). Testing Graphics Code, 4 Years Later [Web log post]. Retrieved from <http://aras-p.info/blog/2011/06/17/testing-graphics-code-4-years-later/>.
- Pyrarson. (2016). [N++ image screen capture]. Retrieved from <https://steamuserimages-a.akamaihd.net/ugc/275104943753088161/EE0FD7F459E32963AD18FB8D7205B97E8759AC16/>.
- Rabin, S. (2010). *Introduction to game development* (2nd ed.). Boston, MA: Course Technology.
- [Rage image screen capture]. (n.d.). Retrieved from <https://zeroharker.files.wordpress.com/2014/01/gen1.jpg>.
- Rage Software. (2002). *Rocky* [Video game]. Montreuil, France: Ubisoft.
- Razor Volare. (2016). [N++, graphical bugs, image screen capture]. Retrieved from <https://steamuserimages-a.akamaihd.net/ugc/266097744487927996/COA2B66AC8D4973F81186DBFDF98A0D2E9CA1B5B/>.
- Red Storm Entertainment. (1998). *Tom Clancy's Rainbow Six* [Video game]. Morrisville, NC: Author.
- Reddy, M. (2011). *API design for C++*. Burlington, MA: Morgan Kaufmann.
- Rees, E., & Fryer, L. (2003, April 22). *IGDA business committee - Best practices in quality assurance/testing*. Retrieved from [https://www.igda.org/resource/collection/FDB22FE1-269A-4EB8-B76A-7CD0BB88A008/IGDA\\_Best\\_Practices\\_QA\\_0.pdf](https://www.igda.org/resource/collection/FDB22FE1-269A-4EB8-B76A-7CD0BB88A008/IGDA_Best_Practices_QA_0.pdf).
- Reimer, J. (2005, November 8). *Cross-platform game development and the next generation of consoles*. Retrieved from <http://arstechnica.com/features/2005/11/crossplatform/>.
- Rockstar North. (2004). *Grand Theft Auto: San Andreas* [Video game]. New York City, NY: Rockstar Games.
- Rocksteady Studios. (2015). *Batman: Arkham Knight* [Video game]. Burbank, CA: Warner Bros. Interactive Entertainment.
- [Rocky, detached limbs, image screen capture]. (n.d.). Retrieved from [http://i.crackedcdn.com/phpimages/article/7/6/1/96761\\_v1.jpg](http://i.crackedcdn.com/phpimages/article/7/6/1/96761_v1.jpg).
- [Rocky, facial defects, image screen capture]. (n.d.). Retrieved from <http://i.crackedcdn.com/phpimages/article/7/5/7/96757.jpg?v=1>.
- Ruskin, E. (2008, February). *How to go from PC to cross platform development without killing your studio*. Retrieved from [http://www.valvesoftware.com/publications/2008/GDC2008\\_CrossPlatformDevelopment.pdf](http://www.valvesoftware.com/publications/2008/GDC2008_CrossPlatformDevelopment.pdf).
- Scacchi, W., & Cooper, K. M. (2015, June). Research challenges at the intersection of computer games and software engineering. In *Conference on foundations of digital games (FDG 2015)*. Pacific Grove, CA.
- Schertenleib, S. (2013). *Unlocking the potential of PlayStation 4: An in-depth developer guide*. <http://develop.scee.net/files/presentations/gceurope2013/ParisGC2013Final.pdf>.
- Schertenleib, S., & Hirani, K. (2009). *SCEE developer services*. [http://develop.scee.net/files/presentations/acgirussia/Case\\_Studies\\_ACGI.09.pdf](http://develop.scee.net/files/presentations/acgirussia/Case_Studies_ACGI.09.pdf).
- Schultz, C., Bryant, R., & Langdell, T. (2005). *Game testing all in one*. Boston, MA: Thomson Course Technology PTR.
- Segal, M., & Akeley, K. (1994). *The design of the OpenGL graphics interface*. [http://www.graphics.stanford.edu/courses/cs448a-01-fall/design\\_opengl.pdf](http://www.graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf).
- Sharke, M. (2011, October 5). *Rage PC launch marred by graphics issues*. Retrieved from <http://pc.gamespy.com/pc/id-tech-5-project/1198334p1.html>.
- Sherrod, A. (2008). *Game graphics programming*. Boston, MA: Course Technology.

- Sherrod, A., & Jones, W. (2012). *Beginning DirectX 11 game programming*. Boston, MA: Course Technology.
- Shirley, P., & Marschner, S. (2009). *Fundamentals of computer graphics*. Boca Raton, FL: CRC Press.
- Skene, J. (2014). *Java robotworld assets*. Unpublished internal document.
- Sony Computer Entertainment. (n.d.). PhyreEngine [Game engine]. Retrieved from <http://develop.scee.net/research-technology/phyreengine/>
- Staggs, M. (2013, November 28). *Battlefield 4 marred by connection issues and graphic glitches*. Retrieved from <http://suvudu.com/2013/11/battlefield-4-marred-by-connection-issues-and-graphic-glitches.html>.
- Starr, K. (2007, July 10). *Testing video games can't possibly be harder than an afternoon with Xbox, right?* Retrieved from <http://www.seattleweekly.com/2007-07-11/news/testing-video-games-can-t-possibly-be-harder-than-an-afternoon-with-xbox-right/>.
- Stone. (1999). *Color and shading fidelity*.
- Subset Games. (2012). *FTL: Faster Than Light* [Video game]. Shanghai, China: Author.
- Sutherland, B. (2013). *Beginning Android C++ game development*. New York, NY: Apress.
- Sutherland, B. (2014). *Learn C++ for game development*. New York, NY: Apress.
- Sutherland, I. E. (1963). Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference* (pp. 329-346). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1461551.1461591> doi: 10.1145/1461551.1461591
- Taito Corporation. (1978). *Space Invaders* [Video game]. Tokyo, Japan: Author.
- Tavakkoli, A. (2015). *Game development and simulation with Unreal Technology*. Boca Raton, FL: Taylor & Francis Group.
- The Creative Assembly. (2013). *Total War: Rome II* [Video game]. Tokyo, Japan: Sega.
- The Sims Studio. (2011). *The Sims Medieval* [Video game]. Redwood City, CA: Electronic Arts.
- Thier, D. (2014, November 13). *Ubisoft stock plummets after botched 'Assassin's Creed: Unity' launch*. Retrieved from <http://www.forbes.com/sites/davidthier/2014/11/13/ubisoft-stock-plummets-after-botched-assassins-creed-unity-launch/>.
- Thorn, A. (2008). *Cross-platform game development*. Plano, TX: Wordware Publishing, Inc.
- Thorn, A. (2011). *Game engine design and implementation*. Sudbury, MA: Jones & Bartlett Learning.
- Timofeitchik, A., & Nagy, R. (2012). *Verification of real time graphics systems* (Master's thesis, Chalmers University of Technology, Gothenburg, Sweden). Retrieved from <http://publications.lib.chalmers.se/records/fulltext/164580.pdf>.
- Traveller's Tales. (2013). *Lego Marvel Super Heroes* [Video game]. Burbank, CA: Warner Bros. Interactive Entertainment.
- Trebilco, D. (2014). GLIntercept [Computer software]. <https://github.com/dtrebilco/glinterscept>. GitHub.
- Treyarch. (2000). *Draconus: Cult of the Wyrms* [Video game]. Santa Monica, CA: Crave Entertainment.
- Treyarch. (2008). *Call of Duty: World at War* [Video game]. Los Angeles, CA: Activision.
- Triumph Studios. (2014). *Age of Wonders III* [Video game]. Delft, Netherlands: Author.
- TT Fusion. (2013). *Lego Marvel Super Heroes: Universe in Peril* [Video game]. Burbank, CA: Warner Bros. Interactive Entertainment.
- Ubisoft Montreal. (2008). *Tom Clancy's Rainbow Six: Vegas 2* [Video game]. Montreuil, France: Ubisoft.

- Ubisoft Montreal. (2013). *Assassin's Creed IV: Black Flag* [Video game]. Montreuil, France: Ubisoft.
- Ubisoft Montreal. (2014a). *Assassin's Creed Unity* [Video game]. Montreuil, France: Ubisoft.
- Ubisoft Montreal. (2014b). *Watch Dogs* [Video game]. Montreuil, France: Ubisoft.
- [Unity Graphics Tests Tool image]. (n.d.). Retrieved from <https://blogs.unity3d.com/wp-content/uploads/2016/12/image05-1.png>.
- [Unity reference image, test result image, difference image]. (n.d.). Retrieved from <https://blogs.unity3d.com/wp-content/uploads/2016/12/Screenshot-2016-12-21-11.18.54.png>.
- Unity Technologies. (n.d.). Unity [Game engine]. Retrieved from <http://unity3d.com/>
- Usher, W. (2014, May 28). *Watch Dogs on Xbox One has more screen-tearing, lower-res shadows than PS4*. Retrieved from <http://www.cinemablend.com/games/Watch-Dogs-Xbox-One-Has-More-Screen-Tearing-Lower-Res-Shadows-Than-PS4-64353.html>.
- Vaishnavi, V., & Kuechler, B. (2013). *Design science research in information systems*. Retrieved from <http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf>.
- Vaishnavi, V., & Kuechler, W. (2008). *Design science research methods and patterns: Innovating information and communication technology*. Boca Raton, FL: Auerbach Publications.
- Varvaressos, S., Lavoie, K., Massé, A. B., Gaboury, S., & Hallé, S. (2014). Automated bug finding in video games: A case study for runtime monitoring. In *Proceedings of the 2014 IEEE international conference on software testing, verification, and validation* (pp. 143–152). Washington, DC, USA: IEEE Computer Society. Retrieved from <http://dx.doi.org/10.1109/ICST.2014.27> doi: 10.1109/ICST.2014.27
- Visual Concepts. (2012). *NBA 2K13* [Video game]. Novato, CA: 2K Sports.
- Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004, April). Image quality assessment: From error visibility to structural similarity. *Trans. Img. Proc.*, 13(4), 600–612. Retrieved from <http://dx.doi.org/10.1109/TIP.2003.819861> doi: 10.1109/TIP.2003.819861
- Wang, Z., & Li, Q. (2011). Information content weighting for perceptual image quality assessment. *IEEE Transactions on Image Processing*, 20(5), 1185–1198.
- Warner, A. & Williams, J. & Lipman, D. (Producers) & Adamson, A. & Asbury, K. & Vernon, C. (Directors). (2004). *Shrek 2* [Motion picture]. Universal City, CA: DreamWorks Pictures.
- Whittle, J., Jones, M. W., & Mantiuk, R. (2017, Jun 01). Analysis of reported error in monte carlo rendered images. *The Visual Computer*, 33(6), 705–713. Retrieved from <https://doi.org/10.1007/s00371-017-1384-7> doi: 10.1007/s00371-017-1384-7
- Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. London: Springer Verlag. <http://eprints.eemcs.utwente.nl/25449/>.
- Wihlidal, G. (2006). *Game engine toolset development*. Boston, MA: Thomson Course Technology PTR.
- Xhane. (2014). All platforms - AC Unity current known issues [Online forum comment]. Retrieved from <http://forums.ubi.com/showthread.php/949543-PC-XB1-amp-PS4-AC-Unity-Current-Known-Issues>.
- Yee, Y. H., & Newman, A. (2004). A perceptual metric for production testing. In *ACM SIGGRAPH 2004 sketches* (pp. 121–). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/1186223.1186374> doi: 10.1145/1186223.1186374
- Yezpitelok, M., & Cantrell, M. (2011, October 25). *The 8 creepiest glitches hidden in popular video games*. Retrieved from [http://www.cracked.com/article\\_19507\\_the-8-creepiest-glitches-hidden-in-popular-video-games\\_p2.html](http://www.cracked.com/article_19507_the-8-creepiest-glitches-hidden-in-popular-video-games_p2.html).

- Yezpitelok, M., Cantrell, M., & Rio, C. (2012, October 28). *The 6 creepiest glitches in famous video games (part 2)*. Retrieved from [http://www.cracked.com/article\\_20125\\_the-6-creepiest-glitches-in-famous-video-games-part-2.html](http://www.cracked.com/article_20125_the-6-creepiest-glitches-in-famous-video-games-part-2.html).
- Younger, P. (2013, September 3). *Total War Rome 2 releases - Graphics issues, crashes and bugs abound*. Retrieved from <http://www.incgamers.com/2013/09/total-war-rome-2-releases-graphics-issues-crashes-and-bugs-abound>.
- Zerbst, S., & Duvel, O. (2004). *3D game engine programming*. Boston, MA: Course Technology PTR.
- Zioma, R. (2012). *Unity: iOS and Android - Cross-platform challenges and solutions*. Retrieved from [http://www.realtimerendering.com/downloads/MobileCrossPlatformChallenges\\_siggraph.pdf](http://www.realtimerendering.com/downloads/MobileCrossPlatformChallenges_siggraph.pdf).
- Zioma, R., & Pranckevicius, A. (2012). *Unity: iOS and Android: Cross platform challenges and solutions*.
- Zobel, J. (2004). *Writing for computer science*. London: Springer.

# Glossary

**2D** Two-dimensional.

**3D** Three-dimensional.

**3DR** 3DRender.

**AAA** Triple-A.

**ANGLE** Almost Native Graphics Layer Engine.

**API** Application programming interface.

**ARB** Architecture Review Board.

**AUT** Auckland University of Technology.

**CAD** Computer-aided design.

**CG** Computer graphics.

**Cg** C for Graphics.

**CG API** Computer graphics API.

**CPU** Central processing unit.

**DSR** Design science research.

**EA** Electronic Arts.

**FFP** Fixed-function graphics pipeline.

**GKS** Graphical Kernal System.

**GL** Graphics Library.

**GLEW** OpenGL Extension Wrangler Library.

**GLFW** OpenGL Framework.

**GLSL** OpenGL Shading Language.

**GLSL ES** OpenGL Shading Language Embedded Systems.

**GLUT** OpenGL Utility Toolkit.

**GPU** Graphics processing unit.

**GUI** Graphical user interface.

**HLSL** High-level Shader Language.

**HOOPS** Hierarchical Object-Oriented Programming System.

**IGDA** International Game Developers Association.

**IPL98** Image Processing Library 98.

**IRIS GL** Integrated Raster Imaging System Graphics Library.

**IW SSIM** Information Content Weighted SSIM.

**MS SSIM** Multiscale SSIM.

**MSE** Mean Squared Error.

**MVVM** Model-view-viewmodel.

**NDA** Non-disclosure agreement.

**NDK** Android Native Development Kit.

**NIST** National Institute of Standards and Technology.

**OpenGL ES** OpenGL Embedded Systems.

**OS** Operating system.

**PC** Personal computer.

**PHIGS** Programmer's Hierarchical Interactive Graphics Standard.

**PRNG** Pseudo-Random Number Generator.

**PSNR** Peak Signal-to-Noise Ratio.

**PSSL** PlayStation Shader Language.

**QA** Quality assurance.

**QD3D** QuickDraw 3D.

**RAM** Random access memory.

**RGB** Red-green-blue.

**RUGVEF** Runtime Graphics Verification Framework.

**SDET** Software Development Engineers in Test.

**SDK** Software development kit.

**SDL** Simple DirectMedia Layer.

**SERL** AUT Software Engineering Research Laboratory.

**SGI** Silicon Graphics.

**SIGGRAPH** ACM Special Interest Group on Computer Graphics and Interactive Techniques.

**SIMD** Single instruction, multiple data.

**SKU** Stock keeping unit.

**SSIM** Structural Similarity Index.

**STL** C++ Standard Template Library.

**UI** User interface.

**UML** Unified Modeling Language.

**USB** Universal serial bus.

**VAGI** VESA Advanced Graphics Interface.

**VDP** Visible differences predictor.

**VESA** Video Electronics Standards Association.

**VRML** Virtual Reality Modeling Language.

**VTK** Visualization Toolkit.

**VTs** VRML Test Suite.



# Appendices

## Appendix A

# Game Industry Testing

The terms *test* and *quality assurance* are used interchangeably in the video game industry (Schultz et al., 2005). Publishers provide quality assurance (QA) services to developers, and large development studios may have an internal QA department (Hight & Novak, 2008). The International Game Developers Association (IGDA) Business Committee issued a report summarising best practices in QA and testing (Rees & Fryer, 2003), the aim of which was to share best practice knowledge in the hope of improving game development practices. Testing was defined as overall quality assurance, which includes functionality, playability, and compatibility, while QA was defined as the process of finding and reporting bugs in games. The report highlighted testing as a time consuming and resource intensive activity, but one that helps to ship games that are bug-free.

Due to the costs associated with QA, smaller developers can choose to outsource QA activities. Testing is an entry-level job, and small companies may be unable to afford a full time testing department. As such, external companies provide game testing services to meet the need for QA testing (Bates, 2003). A downside to outsourcing is that external testers may be unfamiliar with console or publisher standards. Developers may also have internal staff that act as core testers. QA is essential if companies want to compete for consumers and testing is a critical component of the development cycle (Rees & Fryer, 2003).

Game developers can utilise specialised graphics debuggers to analyse complex rendering (Garney & Preisz, 2011). Some examples of graphics debugging tools are (Garney & Preisz, 2011; McShaffry & Graham, 2013; Nantes et al., 2008):

- *PIX* (Microsoft, 2008): used for DirectX, this tool allows capture of a single frame and viewing of API calls made to the graphics pipeline;
- *GLIntercept* (Trebilco, 2014): an open-source OpenGL function call interceptor;
- *Nsight* (nVidia, 2016): nVidia’s tool for graphics debugging and profiling;

- *GPU PerfStudio* (AMD, 2016): used for AMD graphics cards, this tool supports both DirectX and OpenGL.

These sorts of development tools enable a developer to capture rendering output, and to step through the events that created the rendered scene, hence assisting with debugging, but not with initial defect detection. Therefore, these tools provide little assistance for high-level testing of game environment and entertainment value (Nantes et al., 2008).

## A.1 Types of Testing

There are two broad categories of testing:

- *Whitebox Testing* takes an internal perspective to game testing, and source code and multimedia assets can be tested in isolation for functional compliance (Keith, 2010). Programmers can write unit tests to test modules of source code (Myers et al., 2012; Schultz et al., 2005), and artists can conduct asset validation, such as checking for polygon count budgets (Keith, 2010). For this reason, whitebox testing requires technical skill and experience within the respective game development disciplines (Keith, 2010). Testing with only whitebox testing is difficult since it does not account for the complexity introduced by the player feedback loop (Schultz et al., 2005).
- *Blackbox Testing* refers to testing conducted from the user's perspective, with testers required to think and act like players (Keith, 2010; Schultz et al., 2005). This type of testing can be unstructured, but allows for the discovery of bugs not found in whitebox testing since it is conducted from the player's perspective (Bethke, 2003). Testers can be given lists of expected behaviours and the inputs required to ensure they occur (Keith, 2010), however, there is a degree of randomness introduced to the testing since the player and game react to each, creating a feedback loop (Schultz et al., 2005). Testers must adapt their input based on what is happening in a game, and player reactions can be unpredictable (Schultz et al., 2005). A blackbox testing approach can require long hours and be frustrating for the tester (Bethke, 2003; Keith, 2010).

Game programmers do not fully test their own games (Schultz et al., 2005). Developers can overlook defects since they work closely with a game on a daily basis (Keith, 2010). Manual testing is common within the game industry with human testers hired to find, replicate and report on bugs that could negatively impact the consumer's experience of the game (Keith, 2010; Levy & Novak, 2010; Lewis & Whitehead, 2011b; Murphy-Hill et al., 2014). Game testers undertake many different types of testing (Chandler, 2014; Hight & Novak, 2008; Keith, 2010; Levy & Novak, 2010; Schultz et al., 2005):

- *Quality Assurance Testing*: Experienced testers, independent of the development team, find and

classify bugs;

- *Smoke Testing*: Ensures that a build will run, and that the game loads and starts on a target platform without any crashes;
- *Regression Testing*: Conducted by the QA team to ensure old defects have not resurfaced;
- *Play Testing*: Focuses on testing the game to make sure it includes *fun*;
- *Focus Testing*: Conducted by marketing using a broader audience to find the game’s appeal;
- *Balance Testing*: Ensures that game play is not too easy or too difficult and that any AI characters are of reasonable strength when compared to the human player;
- *Compatibility Testing*: Exclusive to the PC platform, it ensures that a game works on different hardware configurations;
- *Compliance Testing*: A certification process conducted by console manufacturers;
- *Localisation Testing*: Ensures the translation into local languages is correct for global target markets;
- *Usability Testing*: Tests for intuitive human interaction with a game.

## A.2 Compliance Testing for Certification

Console manufacturers control the quality of the games released on their platforms by restricting access to the proprietary development kits which developers must pay licensing fees to obtain (Manton & Wilson, n.d.). They also have standards for content verification and conduct compliance testing to enforce that these standards are met (Manton & Wilson, n.d.; KiranKumar & Sundaresasubramanian, n.d.). The quality control standards include a focus on video, graphics and user interface elements, however, the specifics are protected by NDA (Kenwright, 2014; Ruskin, 2008). Ruskin (2008) states that console manufacturer certification processes must be met otherwise a game will not be allowed to ship on a target platform. This compliance testing is known by different terminology depending on the manufacturer, for example, *LotCheck* (Nintendo), *Technical Certification Requirements* (Microsoft), and *Technical Requirements Checklist* (Sony) (Levy & Novak, 2010; Schultz et al., 2005). Manton and Wilson (n.d.) note that Microsoft split the process into two stages: *Compliance Testing*, which deals with the consistency and security of the user experience; and, *Functional Testing*, which deals with game play issues. As console technology has improved, the number of test cases has also increased (Manton & Wilson, n.d.). Specialised game console testing hardware, where a console’s memory footprint can be retrieved for debugging purposes, may be utilised by developers and publishers to help with compliance (Kenwright, 2014).

Examples of certification test cases include (Kenwright, 2014; Levy & Novak, 2010; Manton & Wilson, n.d.):

- Splash screen logos must be displayed for at least two seconds, and the player should not be able to click through before the timer expires;
- In-game text must fit within the screen safe (or guard band) area;
- Naming of functionality and console specific features must be consistent and adhere to manufacturer naming conventions;
- If a game is paused for more than five seconds, it must present an on-screen animation, ensuring that the player realises the game has not crashed;
- Warnings must be displayed when dealing with external storage or data corruption;
- When a controller is unplugged the game must pause;
- Overall game performance must ensure robustness and the game must be playable without suffering memory-related crashes.

Test cases are also created to suit the features of the game under test. Results from Microsoft’s game play testing are collated in the forms of: issues of note; standard reporting issues; and, conditions of resubmission. Manton and Wilson (n.d.) state that 46% of *Xbox 360* games are rejected at least once during certification testing, and as such console manufacturers offer pre-submission testing to developers as a paid optional extra. Kenwright (2014) highlights that certification testing does not focus on finding bugs in video games, but on ensuring that the game software functions correctly within platform-specific constraints.

### A.3 Industry Insight: Testing

To gain insight into testing in the game industry, Starr (2007) worked as a tester at a video game testing outsource company, and reported on her experiences. Her first experience was testing an *Xbox360* update that Microsoft had outsourced to the company. Testers were paid minimum wage to check that the update could be installed and used without the system freezing or crashing. The testers performed tasks repeatedly with different hardware variations, as different regions of the world have different versions of the console. The author observed that the process seemed “grueling and ... ineffectual” (Starr, 2007), and that the industry seemed to prefer testers who love games or are video game addicts. This supports KiranKumar and Sundaresasubramanian (n.d.) who stated that human testers are used to experience the game from the player’s perspective and are expected to have prior gaming experience.

Starr (2007) moved on to spend two weeks working for Volt, another independent video game testing company that hires hundreds of testers for large companies such as Microsoft and Nintendo, but also for small game development studios. Testing at Volt was conducted in secure labs and testers had to sign an NDA. Cameras recorded the testers and cellphones were not allowed in the testing labs. This high level of security and secrecy reinforces the description of the industry as *blackbox* and highly competitive (Lewis & Whitehead, 2011b). A tester who breaches an NDA by posting pictures or video online, for example, could be prosecuted (Levy & Novak, 2010). Testers playtested for eight-hour work days and needed to be articulate, as they had to report crashes and major errors discovered as they played. Part of the reporting included detailing the specific steps, no matter how minor, that would allow a game developer to reproduce the situation that led to the discovery of the bug — the more accurate the report, the easier it is for a developer to reproduce the bug (McShaffry & Graham, 2013). Starr (2007) noted that the lead tester did not care how well the testers played the game, only that it was stable enough to be played, in this case, at the same time as a movie was downloaded to the console. Testers could use documentation that describes the desired flow of testing and manually verify the game step-by-step, or alternatively, they could use their own intuition (KiranKumar & Sundaresasubramanian, n.d.; Schultz et al., 2005). They could also have access to game design documentation to help them determine whether a perceived defect is an actual defect (Lewis et al., 2010). Testers check that there is sufficient challenge in a game, as well as how long is spent playing a given level (KiranKumar & Sundaresasubramanian, n.d.). There are times when games undergo manufacturer submission testing and testers were required to undertake 24-hour shifts (Starr, 2007). Starr (2007) noted that there was high staff turnover and that QA roles in game development were paid less than in general software development.

Despite the low pay, there is a high cost to employing adequate numbers of human testers to ensure a game is of a high enough quality for release (Nantes et al., 2008). When a reduction in spending is required, the game industry executives who control production budgets cut quality assurance first to save money (Starr, 2007).

## Appendix B

# Iteration 6: RobotWorld Assets

### B.1 Graphical Assets

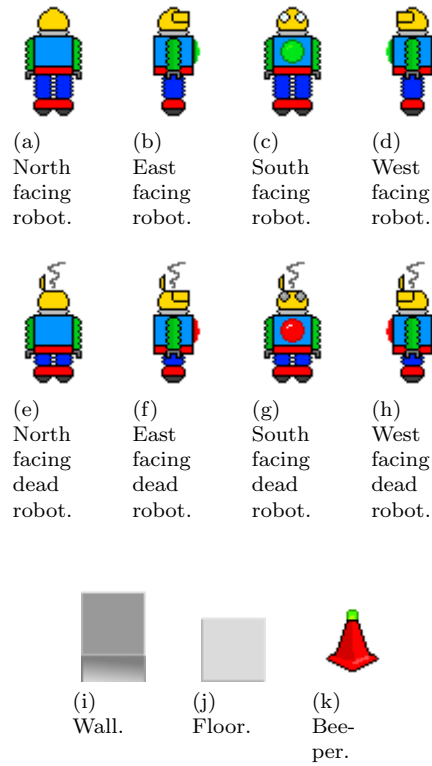


Figure B.1: Iteration 6: RobotWorld sprite assets (Skene, 2014).

## Appendix C

# Software Engineering Research Laboratory (SERL) Seminar 2014

### C.1 Seminar Presentation Slides



# SERL Seminar

Steffan Hooper  
9 December 2014

Primary Supervisor: Stefan Marks  
Secondary Supervisor: Anne Philpott

# Introduction

- Steffan Hooper
  - BSc(CompSci), GradDip(GameDev)
  - School of Computer and Mathematical Sciences
  - AUT University
- Currently: Master of Philosophy
  - Research Topic: *Automated testing and validation of computer graphics implementations for cross-platform game development.*
  - Primary Supervisor: Stefan Marks, CoLab
  - Secondary Supervisor: Anne Philpott, SCMS

# What is the Problem?

This image has been removed for copyright reasons.

Graphical Defect!

Assassin's Creed Unity (Ubisoft Montreal, 2014) [Xbox One, PlayStation 4, Windows PC]

Image retrieved from: [http://ibatt2.gamespot.com/uploads/wallpaper/1544/15443861/2710208\\_389600\\_screenshots\\_2014-11-11\\_00007.jpg](http://ibatt2.gamespot.com/uploads/wallpaper/1544/15443861/2710208_389600_screenshots_2014-11-11_00007.jpg)

# What is the Problem?

- Rocky (Rage Software, 2002):
  - Defects (Yezpitelok & Cantrell, 2011):
    - Character's facial features.
    - Character's limbs detached from their bodies.

This image has been removed for copyright reasons.

Graphical Defects!

This image has been removed for copyright reasons.

Images retrieved from: [http://www.cracked.com/article\\_18507\\_the-6-craziest-glitches-hidden-in-popular-video-games\\_33.html](http://www.cracked.com/article_18507_the-6-craziest-glitches-hidden-in-popular-video-games_33.html)

# What is the Problem?

- Grand Theft Auto: San Andreas (Rockstar North, 2004):
  - Defects (Yezpitelok, Cantrell & Rio, 2012):
    - In-game character's head not displayed.

This image has been removed for copyright reasons.

Graphical Defect!

Image retrieved from: [http://www.cracked.com/article\\_20125\\_the-6-craziest-glitches-in-famous-video-games-part-2\\_g2.html](http://www.cracked.com/article_20125_the-6-craziest-glitches-in-famous-video-games-part-2_g2.html)

# What is the Problem?

- Mafia II (2K Czech, 2010):
  - Defects (Iftikhar, 2010):
    - Black lines rendered across game scenes.
    - Lighting/shadow rendering problems.

This image has been removed for copyright reasons.

Graphical Defect!

Image retrieved from: <http://steamcommunity.com/app/50130/discussions/0/649593366808813/>

Figure C.1: SERL Seminar (slide set 1 of 11).

### What is the Problem?

- **Fallout: New Vegas** (Obsidian Entertainment, 2010):
  - Defects (Hartel, 2011):
    - Character animation problems.
    - Texture loading issues.
    - Shadow rendering issues.

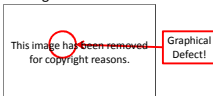


Image retrieved from: [http://www.crashed.com/article\\_20125\\_the-6-craziest-glitches-in-famous-video-games-part-2.html](http://www.crashed.com/article_20125_the-6-craziest-glitches-in-famous-video-games-part-2.html)

### What is the Problem?

- **Call of Duty: Modern Warfare 3** (Infinity Ward, 2011):
  - Defects (Yezpitelok, Cantrell & Rio, 2012):
    - Playable character's head not displayed.




Image retrieved from: [http://www.crashed.com/article\\_20125\\_the-6-craziest-glitches-in-famous-video-games-part-2\\_p2.html](http://www.crashed.com/article_20125_the-6-craziest-glitches-in-famous-video-games-part-2_p2.html)

### What is the Problem?

- **Battlefield 3** (EA Digital Illusions CE, 2011)
  - Defects (Morris, 2014):
    - In-game characters exhibit long necks.

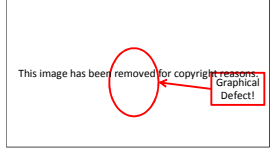


Image retrieved from: <http://BMcrafter.com/wp-content/uploads/2013/10/B3-neck.jpg>

### What is the Problem?

- **Rage** (id Software, 2011)
  - Defects (Sharkey, 2011):
    - Screen tearing.
    - Texture issues.




Image retrieved from: <http://DotaBul.com/2011/02/why-was-the-pc-launch-of-rage-not-a-cluster>

### What is the Problem?

- **Battlefield 4** (EA Digital Illusions CE, 2013)
  - Defects (Staggs, 2013):
    - Graphical problems hinder game play.
      - “Laser Pointer”-style red spots rendered...
    - World geometry stops being rendered when the player's viewing angle changes in game...

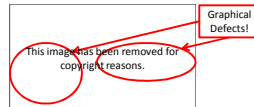


Image retrieved from: <https://www.youtube.com/watch?v=5m0B834Uw>

### What is the Problem?

- **Watch Dogs** (Ubisoft Montreal, 2014):
  - Defects (Cristea, 2014):
    - Anti-aliasing setting caused rendering issues.




Image retrieved from: <https://www.youtube.com/watch?v=4U05u0P4Uw>

Figure C.2: SERL Seminar (slide set 2 of 11).

### What is the Problem?

- Assassin's Creed Unity (Ubisoft Montreal, 2014):
  - Defects (NOWGamer.com, 2014):
    - Delays in texture loading.
    - Low frame rate issues (< 30fps).

This image has been removed for copyright reasons.

Image retrieved from: [http://static2.gamewatch.com/uploads/cale\\_supern/1544/15443863/2732075\\_309050\\_unitywhats\\_2014-11-11\\_00034.jpg](http://static2.gamewatch.com/uploads/cale_supern/1544/15443863/2732075_309050_unitywhats_2014-11-11_00034.jpg)

### What is the Problem?

Published Video Game	Game Engine	Target Platforms
Assassin's Creed IV: Black Flag	AnvilNext	Win, PS3, PS4, WiiU, 360, XB1
Assassin's Creed Unity	AnvilNext	Win, PS4, XB1
Battlefield 4	Frostbite 3 Engine	Win, PS3, PS4, 360, XB1
Call of Duty: Modern Warfare 3	MW3 Engine	Win, OSX, PS3, 360, Wii
Fallout: New Vegas	Gamebryo	Win, PS3, 360
Grand Theft Auto: San Andreas	RenderWare	Win, PS2, Xbox, OSX, PS3, IOS, Android, Kindle, Windows Phone, Fire TV
The Elder Scrolls V: Skyrim	Creation Engine	Win, PS3, 360
Mafia II	Illusion Engine	Win, OSX, PS3, 360
Rage	id Tech 5	Win, OSX, PS3, 360
Rocky	-	GCM, PS2, Xbox, GBA
Watch Dogs	Disrupt Engine	Win, PS3, PS4, 360, XB1, WiiU

### What is the Problem?

- Commercially released video games can contain graphical defects...
  - Defects can:
    - Hinder game play or player experience.
      - Range: Humorous? Unplayable?
    - Impact portions of the game's target platforms.
      - Platform parity...
    - Influence the commercial success.
      - Poor critical reviews... Free DLC compensation...
    - Require post-release bug-fixing and patching.
      - Additional development costs...

### What is the Problem?

- How can testing and validation of computer graphics implementations for cross-platform game development be automated?
  - Testing: obtain information about the product.
  - Validation: Checking and testing the real product. Is the right product being built?
  - Cross-Platform: Product can run identically on different platforms.
  - Automated: Automatic operation.

### Motivation and Goals

- Game Developers Target Multiple Platforms:
  - Commercial viability (Irish, 2005):
    - Wider consumer base.
    - Film tie-in – cross promotion.
  - Development risk reduction (Hook, 2005):
    - Commercial success.
    - Robust software – platform development tool variety.
    - Adaptable to technology changes.
  - Reduced marketing effort and budget (Demerjian, 2012).

### Motivation and Goals

- Issues:
  - Finding errors in the presentation layer of an application is labour intensive (Myers, Badgett & Sandler, 2012).
- Research Project Goals:
  - Increase video game software reliability.
  - Detect graphics problems before commercial release.
  - Reduce human testing burden.

Figure C.3: SERL Seminar (slide set 3 of 11).

Existing Research

- Timofeitchik, A., & Nagy, R. (2012). *Verification of real time graphics systems*.
  - Real-time broadcast television overlay graphics defect detection.
    - Detected previously unknown defects.
  - Structural Similarity Index (SSIM):
    - Perceived change in structural information...
    - Optimised SSIM implementation.
  - Future work identified: video game industry...
    - Dynamic video output... Regression testing...

Existing Research

- Zioma, R., & Pranckevicius, A. (2012). *Unity: ios and android: cross platform challenges and solutions*.

This image has been removed for copyright reasons.

Existing Research

- Zioma, R., & Pranckevicius, A. (2012). *Unity: ios and android: cross platform challenges and solutions*.
  - Unity (Unity Technologies, 2013):
    - Proprietary game engine, closed-source.
  - 238 test scenes.
  - Capture and compare screenshots...
    - Per-pixel comparison.
  - Issues: Device Crashes, Graphics Shader Variants...

This image has been removed for copyright reasons.  
 Unity logo.

Relevant State of the Art

- Current Testing in the Video Game Industry:
  - Testing game software is difficult to automate, humans conduct the majority of testing (Pedriana, 2007).
    - Visual inspections, human resource required...
  - Technical Director of video game developer Crytek suggests future development of automated testing to save money and gain a competitive advantage (Carucci, 2009).
    - Commercial viability...

Relevant State of the Art

- Current Testing in the Video Game Industry:
  - Test Lead at THQ, Inc. states automated testing will be essential in the future for verifying visual appearance in games (Levy & Novak, 2010).
    - Automation need...
  - Sony (SCEE Research and Development) provide conformance, regression and functional testing for the PlayStation 4 (Schertenleib, 2013).
    - Proprietary, licensed technology...
    - Graphics systems? Not cross-platform...

Relevant State of the Art

- Unreal Engine 4 (Epic Games, 2014):
  - Licensed subscription model.
  - Automated Systems:
    - Unit Testing: API tests...
    - Feature Testing:
      - Verify systems work...
    - Content Stress Testing
      - Thorough testing of systems... Load all content...
    - Screenshot comparison:
      - QA comparison of screenshots for rendering issues...

This image has been removed for copyright reasons.  
 Unreal Engine 4 logo.

Figure C.4: SERL Seminar (slide set 4 of 11).

Methodology

- Design Science Research:
  - Create an artefact to solve a problem (Vaishnaivi & Kuechler, 2008).
- DSR Phases – Iterative Research Activity:
  - Awareness of a problem
  - Suggestion
  - Development (with Micro-Evaluations)
  - Evaluation (with Patterns)
  - Conclusion

Methodology

- Design Science Research Methods and Patterns (Vaishnaivi & Kuechler, 2008)
- Applying Patterns:
  - Suggestion and Development:
    - Simulation and Exploration
  - Evaluation and Validation:
    - Simulation
    - Demonstration
    - Benchmarking

This image has been removed for copyright reasons.

Activity Framework for Design Science Research. Adapted from "On theory development in design science research: anatomy of a research project," by B. Kuechler and V. Vaishnavi, European Journal of Information Systems, Volume 17, Issue 5, pp. 489-504. Copyright 2008 by Palgrave Macmillan. Adapted with permission.

Problem Scope

- Commercial Target Variety:
  - Desktop + Current-Gen Console + Next-Gen Console:
    - Windows, OS X, PlayStation 4, PlayStation 3, Xbox One, Xbox 360, Wii U
  - Mobile (Portable + Dedicated):
    - iOS, Android, Windows Phone, Fire TV, Kindle
    - PlayStation Vita, Nintendo 3DS
  - Consoles: Proprietary, closed source systems...
    - Require third-party developer licensing...

Architecture: Graphics Layers

Layer	Technology Stack
Target Exe:	Build Target's (Debug / Release) Output Executable
Software:	Cross-Platform Video Game's Native Source Code
Game Engine:	Game Engine (Reusable Framework / Abstraction)
API:	Graphics Application Programming Interface (API)
Driver:	Graphics Processing Unit Device Driver
OS:	Operating System / Kernel
Hardware GPU:	Video Card Hardware: Graphical Processing Unit
Hardware CPU:	Computer Hardware: Central Processing Unit

Game Developers

Graphics Vendors

Hardware Vendors

Architecture: Graphics Technology

Layer	Win	OS X	PS4	PS3	Xbox	360	Wii U	iOS
High-level API:	DirectX 9/10/11 OpenGL	OpenGL	GNMX	LibGCM	DirectX 11.1	DirectX 9+	GK2	OpenGL ES
Low-level API:	Mantle DirectX12		GNM	-		-	-	Metal
Driver:	Various	Various				None		
OS:	Windows Mac OS X...	Mac OS X...	Orbis OS	CellOS	Xbox OS	...	System Software	iOS ...
GPU:	Various	Various	AMD GCN Radeon	Nvidia/ SCEI RSX	AMD GCN Radeon	ATI Xenos	AMD Radeon Llante	Various
CPU:	Various	Various	8-core AMD x86-64 Jaguar	Cell BE (1 PPE, 7 SPEs)	8-core AMD APU Jaguar	PowerP C Tri- Core Xenon	Tri-Core IBM PowerPC Espresso	Various

Problem Scope

- Research Project Scope:
  - Available targets:
    - Windows, OS X, iOS, Android, Raspberry PI
    - Still great variety!
      - GPUs, Drivers, Low-level APIs, High-level APIs...
  - Cross-platform approach applicable to proprietary consoles...
  - Ever-expanding growth of potential graphics features... And hence possible defects!
    - One developer/researcher, start with a small feature set...

Figure C.5: SERL Seminar (slide set 5 of 11).

### Methodology

- Design Science Research:
  - Awareness of a problem (Research proposal)
  - Suggestion (Research proposal, literature review)
  - Development (Research project to date...)
    - Iteration 1:
      - Tentative Design, Simulation and Exploration, Micro-eval...
    - Iteration 2:
      - Simulation and Exploration, Artefact, Micro-evaluation...
    - Iteration 3:
      - Simulation and Exploration, Artefact, Micro-evaluation...

### Iteration 1: Tentative Design

- Cross Platform Development Strategies:
  - Commercial Game Development in C++ (Kelly, 2012).
  - C++ is platform independent, supports low-level system access (Llopis, 2003).
  - Cross-platform foundations can be built from common framework of classes and functions (Thorn, 2011).
  - Abstraction for multiple platforms (Llopis, 2003).

### Iteration 1: Tentative Design

- Two Target Platforms:
  - C++ rendering abstraction...
  - Graphics implementations for:
    - Windows 7 (32bit), Open GL (GLEW 1.10.0, GLFW 3.0.4)
    - Mac OS X (32bit), Open GL (GLUT)
  - Common/shared cross-platform test scenes
- Automated Comparison Tool:
  - RGB Difference Per Pixel, SSIM Comparison.
  - Windows, C#.

### Iteration 1: Graphics Layers

Layer	Technology Stack
Target Exe:	Build Target's (Debug / Release) Output Executable
Software:	Cross-Platform Video Game's Native Source Code
Game Engine:	Game Engine (Reusable Framework / Abstraction)
API:	Graphics Application Programming Interface (API)
Driver:	Graphics Processing Unit Device Driver
OS:	Operating System / Kernel
Hardware GPU:	Video Card Hardware: Graphical Processing Unit
Hardware CPU:	Computer Hardware: Central Processing Unit

C++ Debug Target, Bespoke Engine / Rendering Framework

### Iteration 1: Artefacts

Build and Test Automation Pipeline:

```

graph TD
    A[Cross Platform Native Code (*.cpp / *.h)] --> B[Target Platform's Build Chain (Compiler/Linker)]
    A --> C[Target Platform's Build Chain (Compiler/Linker)]
    B --> D[Mac OS X Executable]
    C --> E[Windows 7 Executable]
    D --> F[Test Scene Output]
    E --> G[Test Scene Output]
    F --> H[Platform Output Repository]
    G --> H
    H --> I[Comparison Tool Processing]
    I --> J[Test Result Reporting]
  
```

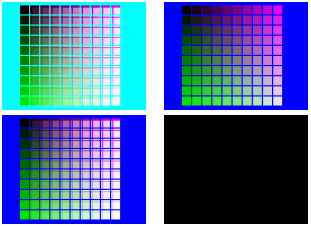
### Iteration 1: Artefact Output

- Graphical Test Scene Output (Mac OS X):

Figure C.6: SERL Seminar (slide set 6 of 11).

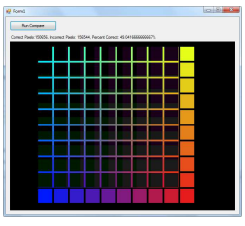
### Iteration 1: Artefact Output

- Graphical Test Scene Output (Windows 7):



### Iteration 1: Artefact

- Automation Tool Results Output:



### Iteration 1: Test Data

- Watch Dogs (Ubisoft, 2014) [Windows, PlayStation 3, PlayStation 4, Xbox 360, Xbox One, Wii U]

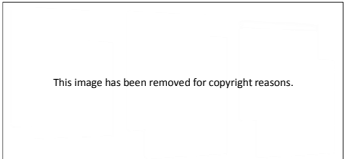


Image retrieved from: [http://watch.dub.uib.no/ressources/15/games/watchdogs/watchdogs\\_screenshot\\_145387.png](http://watch.dub.uib.no/ressources/15/games/watchdogs/watchdogs_screenshot_145387.png)

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison  
– Graphics Quality Level Setting: Low

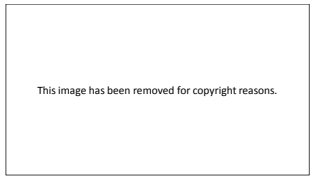


Image retrieved from: <http://www.geforce.com/what-new/guide/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison  
– Graphics Quality Level Setting: Medium

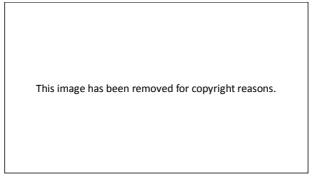


Image retrieved from: <http://www.geforce.com/what-new/guide/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison  
– Graphics Quality Level Setting: High

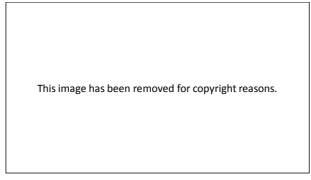
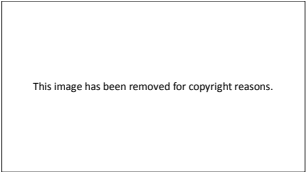


Image retrieved from: <http://www.geforce.com/what-new/guide/watch-dogs-graphics-performance-and-tweaking-guide>

Figure C.7: SERL Seminar (slide set 7 of 11).

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison
  - Graphics Quality Level Setting: Ultra




This image has been removed for copyright reasons.

Image retrieved from: <http://www.gforce.com/whats-new/guide/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison
  - Graphics Quality Level Setting: Spliced...



This image has been removed for copyright reasons.

This image has been removed for copyright reasons.

This image has been removed for copyright reasons.

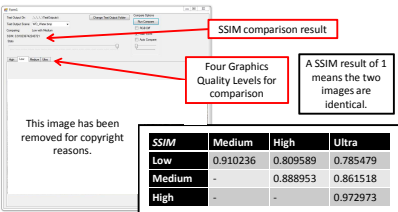
This image has been removed for copyright reasons.

Low Medium High Ultra

Image retrieved from: <http://www.gforce.com/whats-new/guide/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison



SSIM comparison result

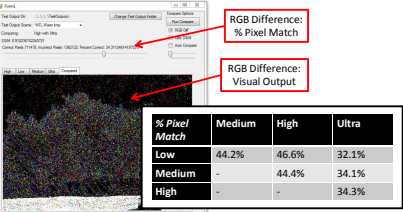
Four Graphics Quality Levels for comparison

A SSIM result of 1 means the two images are identical.

SSIM	Medium	High	Ultra
Low	0.910236	0.809589	0.785479
Medium	-	0.888953	0.861518
High	-	-	0.972973

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: RGB Difference



RGB Difference: % Pixel Match

RGB Difference: Visual Output


% Pixel Match	Medium	High	Ultra
Low	44.2%	46.6%	32.1%
Medium	-	44.4%	34.1%
High	-	-	34.3%

### Iteration 1: Micro-Evaluations

- DSR Suggestion Revisited:
  - RGB Difference may not be enough for image comparison...
    - Low percent of similar pixels...
    - Yet different rendered scenes do look similar...
  - SSIM comparisons results are promising...
    - Further types of comparison to be explored...
  - Cross-platform graphics abstractions...
    - Limited target platforms...
    - Perhaps use a third-party cross-platform abstraction...

### Iteration 2: Artefact

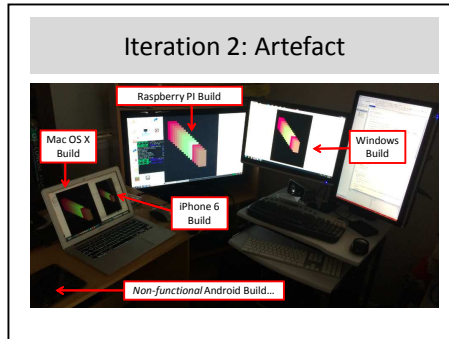
- Five Target Platforms
  - Graphics Abstraction: Simple DirectMedia Layer
    - SDL2.0.3: C based, commercially used in video games.
  - Targets:
    - Windows 7
    - Mac OS X
    - iOS, iPhone 6 Simulator
    - Raspberry Pi
    - Android
  - C++ Cross-Platform Test Scene



Example Test Scene

Figure C.8: SERL Seminar (slide set 8 of 11).





### Iteration 2: Micro-Evaluations

- DSR Suggestion Revisited:
  - SDL2.0.3's 2D rendering is cross-platform...
    - But 3D requires direct calls to the Graphics API.
  - Output scene data file locations:
    - Windows & OS X: Easy file system access...
    - iOS: Applications are sandboxed...
    - Raspberry Pi: Unsure...
    - Possibly send outputs via a network connection.
  - Android tool chain and debugging difficulty:
    - Build crashes in simulators, and on hardware device...

### Iteration 3: Artefact

- Two Target Platforms
  - Graphics Abstraction:
    - Simple DirectMedia Layer (SDL 2.0.3)
      - C based. Commercially used in video games.
  - Targets:
    - Windows 7 (SDL2.0.3 – OpenGL 2.0)
    - Mac OS X (SDL2.0.3 – OpenGL 2.0)
  - C++ Cross-Platform Test Scene
    - 3D mesh loading, rendering with procedural animation
    - Fixed function pipeline, with light

### Iteration 3: Artefact

- Graphical Test Scene 1 (Mac OS X vs Win PC):

### Iteration 3: Artefact

- Graphical Test Scene 2 (Mac OS X vs Win PC):

### Iteration 3: Micro-Evaluations

- DSR Suggestion Revisited:
  - Further test scenes to be added:
    - Texturing, Blending, Alphing.
    - Character animation, Physics simulation.
  - Graphics abstraction:
    - OpenGL (Desktop) vs OpenGL ES (Mobile).
    - Iteration 1 style abstraction for 3D scenes...
  - Replay system:
    - Deterministic game simulation.
    - Replay saved game state on multiple platforms.

Figure C.9: SERL Seminar (slide set 9 of 11).

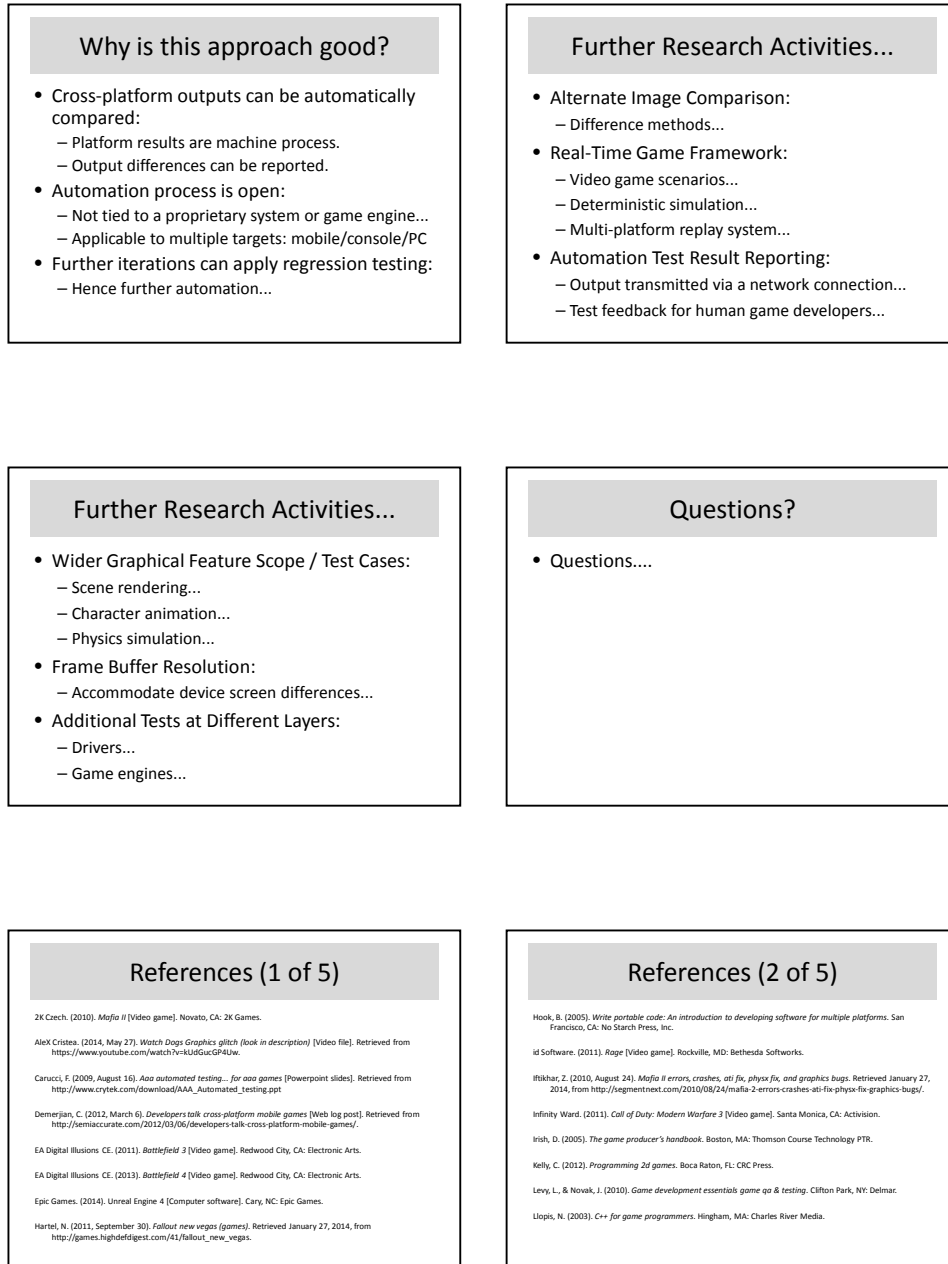


Figure C.10: SERL Seminar (slide set 10 of 11).

References (3 of 5)

Morris, C. (2014). *How to become a game tester – and why you may not want to*. Retrieved December 7, 2014 from <https://games.yahoo.com/blogs/plugged-in/how-to-become-a-game-tester-9e2b809604-and-why-you-may-not-want-to-202954900.html>.

Myers, G., Badgett, T., & Sandler, C. (2012). *The art of software testing* (3rd ed.). John Wiley & Sons, Inc.: Hoboken, NJ.

NOWGamer.com. (2014, November 11). *Assassin's Creed: Unity Framework is "Atrocious"*. Retrieved November 13, 2014 from <http://www.nowgamer.com/assassins-creed-unity-framework-is-atrocious/>.

Obsidian Entertainment . (2010). *Fallout: New Vegas* [Video game]. Rockville, MD: Bethesda Softworks.

Pedriana, P. (2007). *Eastl – electronic arts standard template library*. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html>.

Rage Software. (2002). *Rocky* [Video game]. Montreuil, France: Ubisoft.

Rockstar North. (2004). *Grand Theft Auto: San Andreas* [Video game]. New York, NY: Rockstar Games.

References (4 of 5)

Schertenleib, S. (2013). *Unlocking the potential of PlayStation 4: An in-depth developer guide*. <http://develop.sce.net/files/presentations/gceurope2013/ParisGC2013Final.pdf>.

Sharkey, M. (2011, October 5). *Rage PC launch marred by graphics issues*. Retrieved January 27, 2014 from <http://pc.gamespy.com/pc/t6-tech-5-project/1198534pt.html>.

Staggs, M. (2013, November 28). *Battlefield 4 marred by connection issues and graphic glitches*. Retrieved December 20, 2013, from <http://uvuudu.com/2013/11/battlefield-4-marred-by-connection-issues-and-graphic-glitches.html>.

Thorn, A. (2011). *Game engine design and implementation*. Sudbury, MA: Jones & Bartlett Learning.

Timofteitchik, A., & Nagy, R. (2012). *Verification of real time graphics systems* (Master's thesis, Chalmers University of Technology, Gothenburg, Sweden). Retrieved from <http://publications.lib.chalmers.se/records/fulltext/164580.pdf>.

Ubisoft Montreal. (2014). *Assassin's Creed Unity* [Video game]. Montreuil, France: Ubisoft.

Ubisoft Montreal. (2014). *Watch Dogs* [Video game]. Montreuil, France: Ubisoft.

References (5 of 5)

Unity Technologies. (2013). *Unity Pro* [Computer software]. Retrieved November 27, 2013, from [https://store.unity3d.com/products/v18/feature\\_3\\_pro\\_en.html](https://store.unity3d.com/products/v18/feature_3_pro_en.html).

Vaishtai, V., & Kuechler, W. (2008). *Design Science Research Methods and Patterns*. Boca Raton, FL: Auerbach Publications.

Yezpikelok, M., & Cantrell, M. (2011, October 25). *The 8 creepiest glitches hidden in popular video games*. Retrieved January 24, 2014, from <http://www.cracked.com/article-19507-the-8-creepiest-glitches-hidden-in-popular-video-games-p2.html>.

Yezpikelok, M., Cantrell, M., & Rio, C. (2012, October 28). *The 6 creepiest glitches in famous video games (part 2)*. Retrieved January 24, 2014, from <http://www.cracked.com/article-20125-the-6-creepiest-glitches-in-famous-video-games-part-2.html>.

Zioma, R., & Prankevicius, A. (2012). *Unity: ios and android: cross platform challenges and solutions*.

Figure C.11: SERL Seminar (slide set 11 of 11).

## Appendix D

# PikPok Developers' Conference 2015

### D.1 Conference Presentation Slides

Research Project:  
Automated testing of computer graphics  
for cross-platform game development

Steffan Hooper  
5 November 2015

Primary Supervisor: Stefan Marks  
Secondary Supervisor: Anne Philpott

Introduction

- Steffan Hooper
  - BSc(CompSci)
  - GradDipGameDev(GameProg)
  - AUT University: Lecturer and Co-Director of PiGSty
    - School of Engineering, Computer and Mathematical Sciences
  - Currently: Master of Philosophy
    - Research Topic: Automated testing and validation of computer graphics implementations for cross-platform game development.
    - Primary Supervisor: Stefan Marks, CoLab, AUT
    - Secondary Supervisor: Anne Philpott, SECMS, AUT

What is the Problem?

This image has been removed for copyright reasons.

Assassin's Creed Unity (Ubisoft Montreal, 2014) [Xbox One, PlayStation 4, Windows PC]

Image retrieved from: [http://ibatt2.gamespot.com/uploads/wallpaper/1544/15443861/2710208\\_389601\\_screenshots\\_2014-11-11\\_00007.jpg](http://ibatt2.gamespot.com/uploads/wallpaper/1544/15443861/2710208_389601_screenshots_2014-11-11_00007.jpg)

What is the Problem?

- Rocky (Rage Software, 2002):
  - Defects (Yezpitelok & Cantrell, 2011):
    - Character's facial features.
    - Character's limbs detached from their bodies.

Images retrieved from: [http://www.cracked.com/article\\_19507\\_the-6-craziest-glitches-hidden-in-popular-video-games\\_33.html](http://www.cracked.com/article_19507_the-6-craziest-glitches-hidden-in-popular-video-games_33.html)

What is the Problem?

- Grand Theft Auto: San Andreas (Rockstar North, 2004):
  - Defects (Yezpitelok, Cantrell & Rio, 2012):
    - In-game character's head not displayed.

Image retrieved from: [http://www.cracked.com/article\\_20125\\_the-6-craziest-glitches-in-famous-video-games-part-2\\_32.html](http://www.cracked.com/article_20125_the-6-craziest-glitches-in-famous-video-games-part-2_32.html)

What is the Problem?

- Mafia II (2K Czech, 2010):
  - Defects (Iftikhar, 2010):
    - Black lines rendered across game scenes.
    - Lighting/shadow rendering problems.

Image retrieved from: <http://steamcommunity.com/app/50130/discussions/0/649593366809813/>

Figure D.1: PikPok Developers' Conference (slide set 1 of 21).

### What is the Problem?

- Fallout: New Vegas (Obsidian Entertainment, 2010):
  - Defects (Hartel, 2011):
    - Character animation problems.
    - Texture loading issues.
    - Shadow rendering issues.

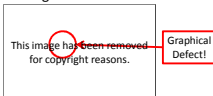


Image retrieved from: [http://www.crashed.com/article\\_20125\\_the-6-most-notable-glitches-in-famous-video-games-part-2.html](http://www.crashed.com/article_20125_the-6-most-notable-glitches-in-famous-video-games-part-2.html)

### What is the Problem?

- Call of Duty: Modern Warfare 3 (Infinity Ward, 2011):
  - Defects (Yezpitelok, Cantrell & Rio, 2012):
    - Playable character's head not displayed.




Image retrieved from: [http://www.crashed.com/article\\_20125\\_the-6-most-notable-glitches-in-famous-video-games-part-2\\_p2.html](http://www.crashed.com/article_20125_the-6-most-notable-glitches-in-famous-video-games-part-2_p2.html)

### What is the Problem?

- Battlefield 3 (EA Digital Illusions CE, 2011)
  - Defects (Morris, 2014):
    - In-game characters exhibit long necks.

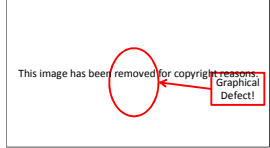


Image retrieved from: <http://BMcrafter.com/wp-content/uploads/2013/10/B3-neck.jpg>

### What is the Problem?

- Rage (id Software, 2011)
  - Defects (Sharkey, 2011):
    - Screen tearing.
    - Texture issues.




Image retrieved from: <http://DotaDota.com/5847762/why-was-the-pc-launch-of-rage-not-a-cluster>

### What is the Problem?

- Battlefield 4 (EA Digital Illusions CE, 2013)
  - Defects (Staggs, 2013):
    - Graphical problems hinder game play.
      - “Laser Pointer”-style red spots rendered...
    - World geometry stops being rendered when the player's viewing angle changes in game...

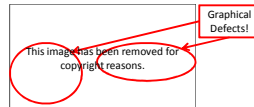


Image retrieved from: <https://www.youtube.com/watch?v=5m0B834Uw>

### What is the Problem?

- Watch Dogs (Ubisoft Montreal, 2014):
  - Defects (Cristea, 2014):
    - Anti-aliasing setting caused rendering issues.

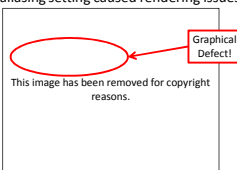


Image retrieved from: <https://www.youtube.com/watch?v=4U05u0P4Uw>

Figure D.2: PikPok Developers' Conference (slide set 2 of 21).

### What is the Problem?

- Assassin's Creed Unity (Ubisoft Montreal, 2014):
  - Defects (NOWGamer.com, 2014):
    - Delays in texture loading.
    - Low frame rate issues (< 30fps).




Image retrieved from: [http://static2.gamewatch.com/uploads/cale\\_super/2544/25443862/2732075\\_309050\\_unitywhats\\_2014-11-11\\_00034.jpg](http://static2.gamewatch.com/uploads/cale_super/2544/25443862/2732075_309050_unitywhats_2014-11-11_00034.jpg)

### What is the Problem?

- Batman: Arkham Knight (Rocksteady Studio, 2015):
  - Defects (Lawler, 2015):
    - Low quality textures and glitches...

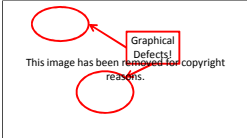


Image retrieved from: <http://imgur.com/9t4e8b>

### What is the Problem?

Published Video Game	Game Engine	Target Platforms
Assassin's Creed IV: Black Flag	AnvilNext	Win, PS3, PS4, WiiU, 360, XB1
Assassin's Creed Unity	AnvilNext	Win, PS4, XB1
Batman: Arkham Knight	Unreal Engine	Win, OSX, PS4, XB1, Linux
Battlefield 4	Frostbite 3 Engine	Win, PS3, PS4, 360, XB1
Call of Duty: Modern Warfare 3	MW3 Engine	Win, OSX, PS3, 360, Wii
Fallout: New Vegas	Gamebryo	Win, PS3, 360
Grand Theft Auto: San Andreas	RenderWare	Win, PS2, Xbox, OSX, PS3, iOS, Android, Kindle, Windows Phone, Fire TV
The Elder Scrolls V: Skyrim	Creation Engine	Win, PS3, 360
Mafia II	Illusion Engine	Win, OSX, PS3, 360
Rage	id Tech 5	Win, OSX, PS3, 360
Rocky	-	GCN, PS2, Xbox, GBA
Watch Dogs	Disrupt Engine	Win, PS3, PS4, 360, XB1, WiiU

### What is the Problem?

- Commercially released video games can contain graphical defects...
  - Defects can:
    - Hinder game play or player experience.
      - Range: Humorous? Unplayable?
    - Impact portions of the game's target platforms.
      - Platform parity...
    - Influence the commercial success.
      - Poor critical reviews... Free DLC compensation...
    - Require post-release bug-fixing and patching.
      - Additional development costs...

### What is the Problem?

- How can testing and validation of computer graphics implementations for cross-platform game development be automated?
  - Testing: obtain information about the product.
  - Validation: Checking and testing the real product. Is the right product being built?
  - Cross-Platform: Product can run identically on different platforms.
  - Automated: Automatic operation.

### Motivation and Goals

- Game Developers Target Multiple Platforms:
  - Commercial viability (Irish, 2005):
    - Wider consumer base.
    - Film tie-in – cross promotion.
  - Development risk reduction (Hook, 2005):
    - Commercial success.
    - Robust software – platform development tool variety.
    - Adaptable to technology changes.
  - Reduced marketing effort and budget (Demerjian, 2012).

Figure D.3: PikPok Developers' Conference (slide set 3 of 21).

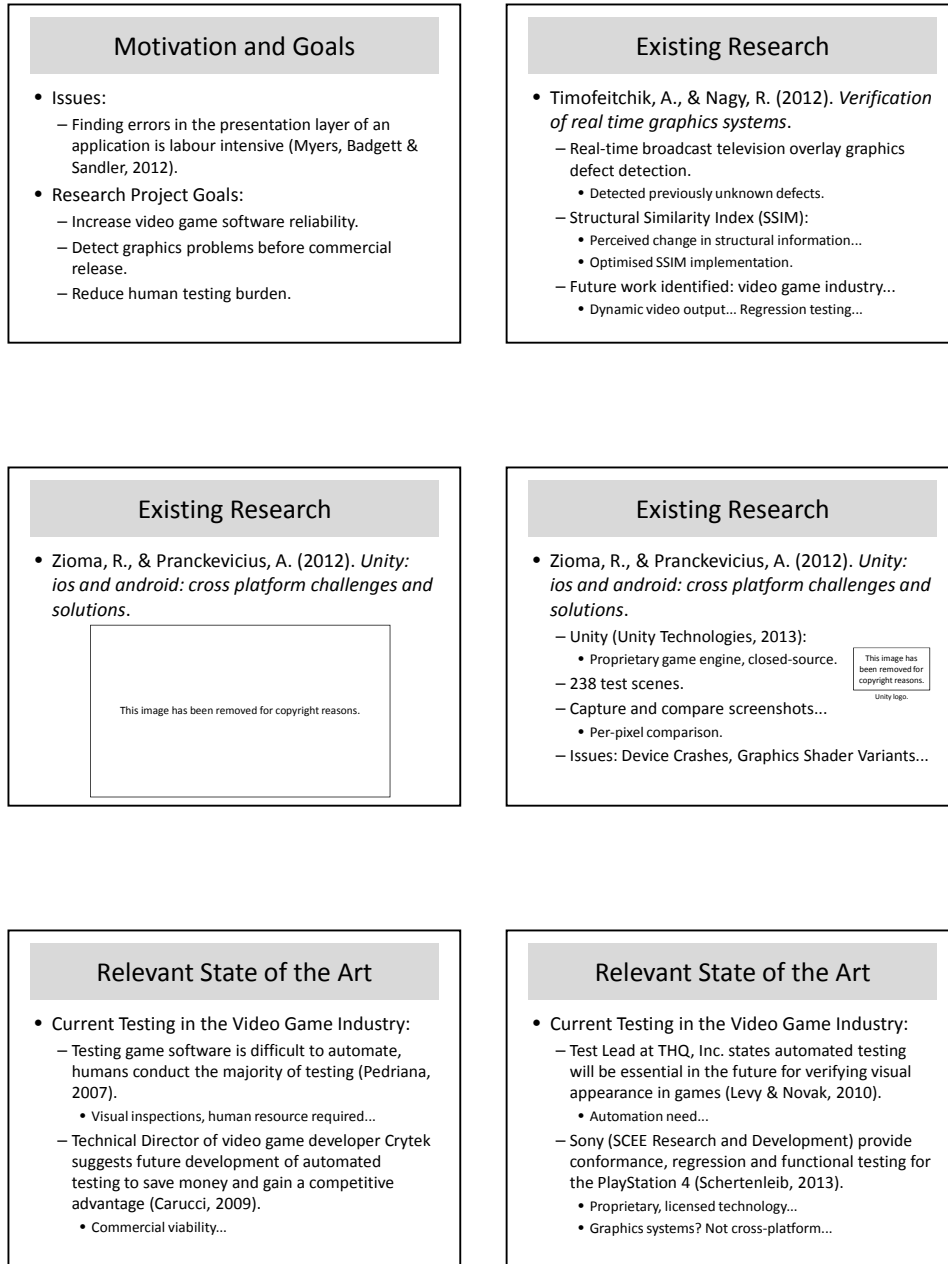


Figure D.4: PikPok Developers' Conference (slide set 4 of 21).



Relevant State of the Art

- Unreal Engine 4 (Epic Games, 2014):
  - Licensed subscription model.
  - Automated Systems:
    - Unit Testing: API tests...
    - Feature Testing:
      - Verify systems work...
    - Content Stress Testing
      - Thorough testing of systems... Load all content...
    - Screenshot comparison:
      - QA comparison of screenshots for rendering issues...

This image has been removed for copyright reasons.

Unreal Engine 4 logo.

Methodology

- Design Science Research:
  - Create an artefact to solve a problem (Vaishnaivi & Kuechler, 2008).
- DSR Phases – Iterative Research Activity:
  - Awareness of a problem
  - Suggestion
  - Development (with Micro-Evaluations)
  - Evaluation (with Patterns)
  - Conclusion

Methodology

- Design Science Research Methods and Patterns (Vaishnaivi & Kuechler, 2008)
- Applying Patterns:
  - Suggestion and Development:
    - Simulation and Exploration
  - Evaluation and Validation:
    - Simulation
    - Demonstration
    - Benchmarking

This image has been removed for copyright reasons.

Activity Framework for Design Science Research. Adapted from "On theory development in design science research: anatomy of a research project," by B. Kuechler and V. Vaishnaivi, European Journal of Information Systems, Volume 17, Issue 5, pp. 489-504. Copyright 2008 by Palgrave Macmillan. Adapted with permission.

Problem Scope

- Commercial Target Variety:
  - Desktop + Current-Gen Console + Next-Gen Console:
    - Windows, OS X, PlayStation 4, PlayStation 3, Xbox One, Xbox 360, Wii U
  - Mobile (Portable + Dedicated):
    - iOS, Android, Windows Phone, Fire TV, Kindle
    - PlayStation Vita, Nintendo 3DS
  - Consoles: Proprietary, closed source systems...
    - Require third-party developer licensing...

Architecture: Graphics Layers

Layer	Technology Stack
Target Exe:	Build Target's (Debug / Release) Output Executable
Software:	Cross-Platform Video Game's Native Source Code
Game Engine:	Game Engine (Reusable Framework / Abstraction)
API:	Graphics Application Programming Interface (API)
Driver:	Graphics Processing Unit Device Driver
OS:	Operating System / Kernel
Hardware GPU:	Video Card Hardware: Graphical Processing Unit
Hardware CPU:	Computer Hardware: Central Processing Unit

Game Developers

Graphics Vendors

Hardware Vendors

Architecture: Graphics Technology

Layer	Win	OS X	PS4	PS3	X81	360	Wii U	iOS
High-level API:	DirectX 9/10/11 OpenGL	OpenGL	GNMx	LibGCM	DirectX 11.1	DirectX 9+	GK2	OpenGL ES
Low-level API:	Mantle DirectX12		GNM	-		-	-	Metal
Driver:	Various	Various				None		
OS:	Windows ...	Mac OS X ...	Orbis OS	CellOS	Xbox OS	...	System Software	iOS ...
GPU:	Various	Various	AMD GCN Radeon	Nvidia/ SCEI RSX	AMD GCN Radeon	ATI Xenos	AMD Radeon Latte	Various
CPU:	Various	Various	8-core AMD x86-64 Jaguar	Cell BE (1 PPE, 7 SPEs)	8-core AMD APU Jaguar	PowerPC Tri-Core Xenon	Tri-Core IBM PowerPC Espresso	Various

Figure D.5: PikPok Developers' Conference (slide set 5 of 21).

### Problem Scope

- Research Project Scope:
  - Available targets:
    - Windows, OS X, iOS, Android, Raspberry PI
    - Still great variety!
      - GPUs, Drivers, Low-level APIs, High-level APIs...
  - Cross-platform approach applicable to proprietary consoles...
  - Ever-expanding growth of potential graphics features... And hence possible defects!
    - One developer/researcher, start with a small feature set...

### Methodology

- Design Science Research:
  - Awareness of a problem (Research proposal)
  - Suggestion (Research proposal, literature review)
  - Development (Research project to date...)
    - Iteration 1:
      - Tentative Design, Simulation and Exploration, Micro-eval...
    - Iterations 2, 3, 4 and 5:
      - Simulation and Exploration, Artefact, Micro-evaluation...
  - Evaluation and Validation (Research thesis...)
  - Conclusion (Research thesis...)

### Iteration 1: Tentative Design

- Cross Platform Development Strategies:
  - Commercial Game Development in C++ (Kelly, 2012).
  - C++ is platform independent, supports low-level system access (Llopis, 2003).
  - Cross-platform foundations can be built from common framework of classes and functions (Thorn, 2011).
  - Abstraction for multiple platforms (Llopis, 2003).

### Iteration 1: Tentative Design

- Two Target Platforms:
  - C++ rendering abstraction...
  - Graphics implementations for:
    - Windows 7 (32bit), Open GL (GLEW 1.10.0, GLFW 3.0.4)
    - Mac OS X (32bit), Open GL (GLUT)
  - Common/shared cross-platform test scenes
- Automated Comparison Tool:
  - RGB Difference Per Pixel, SSIM Comparison.
  - Windows, C#.

### Iteration 1: Graphics Layers

Layer	Technology Stack
Target Exe:	Build Target's (Debug / Release) Output Executable
Software:	Cross-Platform Video Game's Native Source Code
Game Engine:	Game Engine (Reusable Framework / Abstraction)
API:	Graphics Application Programming Interface (API)
Driver:	Graphics Processing Unit Device Driver
OS:	Operating System / Kernel
Hardware GPU:	Video Card Hardware: Graphical Processing Unit
Hardware CPU:	Computer Hardware: Central Processing Unit

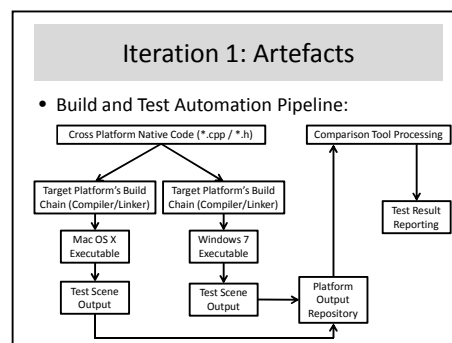
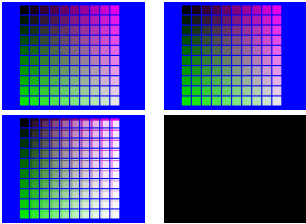


Figure D.6: PikPok Developers' Conference (slide set 6 of 21).

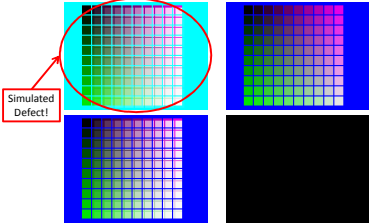
### Iteration 1: Artefact Output

- Graphical Test Scene Output (Mac OS X):



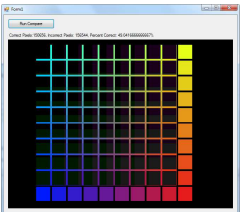
### Iteration 1: Artefact Output

- Graphical Test Scene Output (Windows 7):



### Iteration 1: Artefact

- Automation Tool Results Output:



### Iteration 1: Test Data

- Watch Dogs (Ubisoft, 2014) [Windows, PlayStation 3, PlayStation 4, Xbox 360, Xbox One, Wii U]

This image has been removed for copyright reasons.

Image retrieved from: [http://static.cdn.ubi.com/resources/en-US/games/watchdogs/watchdogs\\_hqplay\\_boxart\\_140587.png](http://static.cdn.ubi.com/resources/en-US/games/watchdogs/watchdogs_hqplay_boxart_140587.png)

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison – Graphics Quality Level Setting: Low

This image has been removed for copyright reasons.

Image retrieved from: <http://www.geforce.com/watchdogs/guides/watchdogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison – Graphics Quality Level Setting: Medium

This image has been removed for copyright reasons.

Image retrieved from: <http://www.geforce.com/watchdogs/guides/watchdogs-graphics-performance-and-tweaking-guide>

Figure D.7: PikPok Developers' Conference (slide set 7 of 21).

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison
  - Graphics Quality Level Setting: High

This image has been removed for copyright reasons.

Image retrieved from: <http://www.geforce.com/whats-new/guides/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison
  - Graphics Quality Level Setting: Ultra

This image has been removed for copyright reasons.

Image retrieved from: <http://www.geforce.com/whats-new/guides/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison
  - Graphics Quality Level Setting: Spliced...

This image has been removed for copyright reasons.

Low

This image has been removed for copyright reasons.

Medium

This image has been removed for copyright reasons.

High

This image has been removed for copyright reasons.

Ultra

Image retrieved from: <http://www.geforce.com/whats-new/guides/watch-dogs-graphics-performance-and-tweaking-guide>

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: SSIM comparison

SSIM comparison result

Four Graphics Quality Levels for comparison

A SSIM result of 1 means the two images are identical.

SSIM	Medium	High	Ultra
Low	0.910236	0.809589	0.785479
Medium	-	0.888953	0.861518
High	-	-	0.972973

This image has been removed for copyright reasons.

### Iteration 1: Artefact

- Watch Dogs [Windows PC]: RGB Difference

RGB Difference: % Pixel Match

RGB Difference: Visual Output

% Pixel Match	Medium	High	Ultra
Low	44.2%	46.6%	32.1%
Medium	-	44.4%	34.1%
High	-	-	34.3%


### Iteration 1: Micro-Evaluations

- DSR Suggestion Revisited:
  - RGB Difference may not be enough for image comparison...
    - Low percent of similar pixels...
    - Yet different rendered scenes do look similar...
  - SSIM comparisons results are promising...
    - Further types of comparison to be explored...
  - Cross-platform graphics abstractions...
    - Limited target platforms...
    - Perhaps use a third-party cross-platform abstraction...

Figure D.8: PikPok Developers' Conference (slide set 8 of 21).

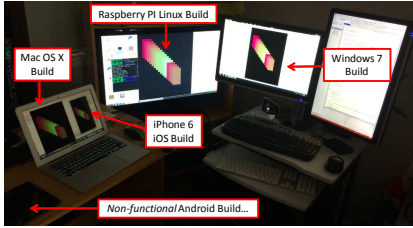
### Iteration 2: Artefact

- Five Target Platforms
  - Graphics Abstraction: Simple DirectMedia Layer
    - SDL2.0.3: C based, commercially used in video games.
  - Targets:
    - Windows 7
    - Mac OS X
    - iOS, iPhone 6 Simulator
    - Linux, Raspberry Pi
    - Android
  - C++ Cross-Platform Test Scene



Example Test Scene

### Iteration 2: Artefact



### Iteration 2: Micro-Evaluations

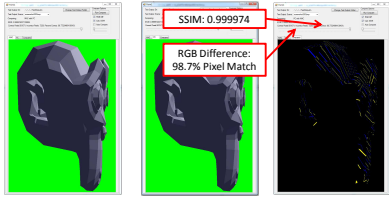
- DSR Suggestion Revisited:
  - SDL2.0.3's 2D rendering is cross-platform...
    - But 3D requires direct calls to the Graphics API.
  - Output scene data file locations:
    - Windows & OS X: Easy file system access...
    - iOS: Applications are sandboxed...
    - Raspberry Pi: Linux file system...
    - Possibly send outputs via a network connection.
  - Android tool chain and debugging difficulty:
    - Build crashes in simulators, and on hardware device...

### Iteration 3: Artefact

- Two Target Platforms
  - Graphics Abstraction:
    - Simple DirectMedia Layer (SDL 2.0.3)
      - C based. Commercially used in video games.
  - Targets:
    - Windows 7 (SDL2.0.3 – OpenGL 2.0)
    - Mac OS X (SDL2.0.3 – OpenGL 2.0)
  - C++ Cross-Platform Test Scene:
    - 3D mesh loading, rendering with procedural animation
    - Fixed function pipeline, with light

### Iteration 3: Artefact

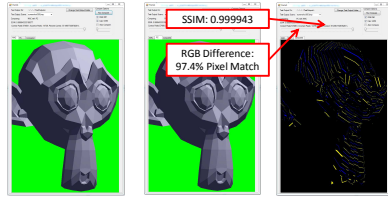
- Graphical Test Scene 1 (Mac OS X vs Win PC):



Mac OS X Output    Windows PC Output    Automated Tool Results

### Iteration 3: Artefact

- Graphical Test Scene 2 (Mac OS X vs Win PC):



Mac OS X Output    Windows PC Output    Automated Tool Results

Figure D.9: PikPok Developers' Conference (slide set 9 of 21).

Iteration 3: Micro-Evaluations

- DSR Suggestion Revisited:
  - Further test scenes to be added, more test data:
    - Texturing, Alpha Blending.
    - Character animation, Physics simulation.
  - Graphics abstraction:
    - OpenGL (Desktop) vs OpenGL ES (Mobile).
    - Iteration 1 style abstraction for 3D scenes...
  - Replay system:
    - Deterministic game simulation.
    - Replay saved game state on multiple platforms.

Iteration 4: Artefact

- Three Target Platforms
  - Targets:
    - Windows 7 (SDL 2.0.3 – OpenGL 2.0)
    - Mac OS X (SDL 2.0.3 – OpenGL 2.0)
    - Raspbian (SDL 2.0.3 – OpenGL)
- Programmable C++ Animated Test Scenes:
  - Re-useable C++ test scene interfaces...
    - Turtle Graphics and Robot World
  - Utilised to generate more test scene data...
    - Animations, texturing, alpha blending...

Iteration 4: Artefact

- Programmable C++ Animated Test Scenes:
  - Turtle Graphics example:

```

int main()
{
    for (int k = 0; k < 3; ++k)
    {
        pen_colour(NEXT);

        for (int i = 0; i < 3; ++i)
        {
            forward(100);
            turn(360/3);
        }

        turn(360/3);
    }
}

```

```

*Start of Program!
*Debug Messages On!

```

```

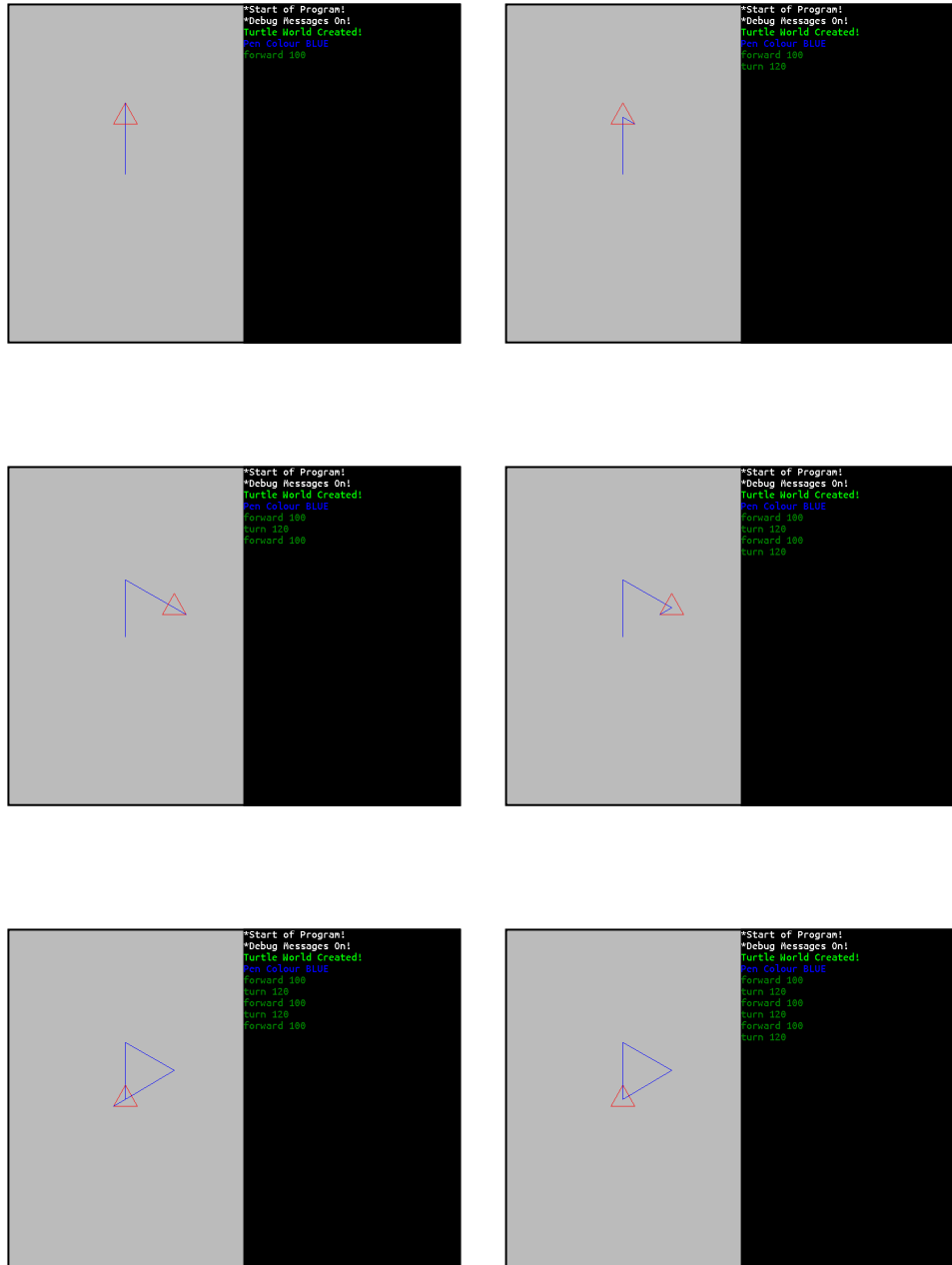
*Start of Program!
*Debug Messages On!
Turtle World Created!

```

```

*Start of Program!
*Debug Messages On!
Turtle World Created!
Pen Colour BLUE

```



11

Figure D.11: PikPok Developers' Conference (slide set 11 of 21).

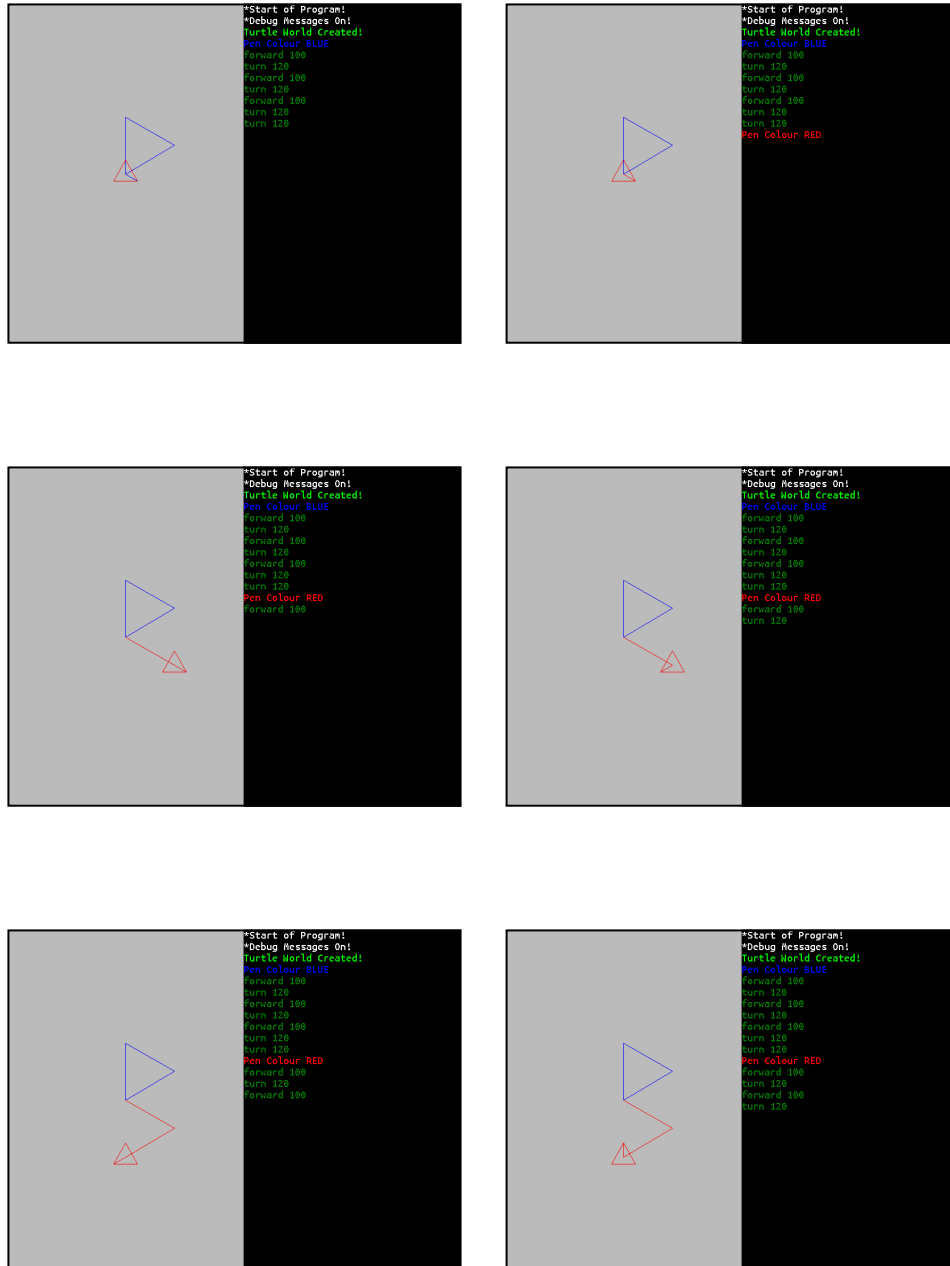


Figure D.12: PikPok Developers' Conference (slide set 12 of 21).



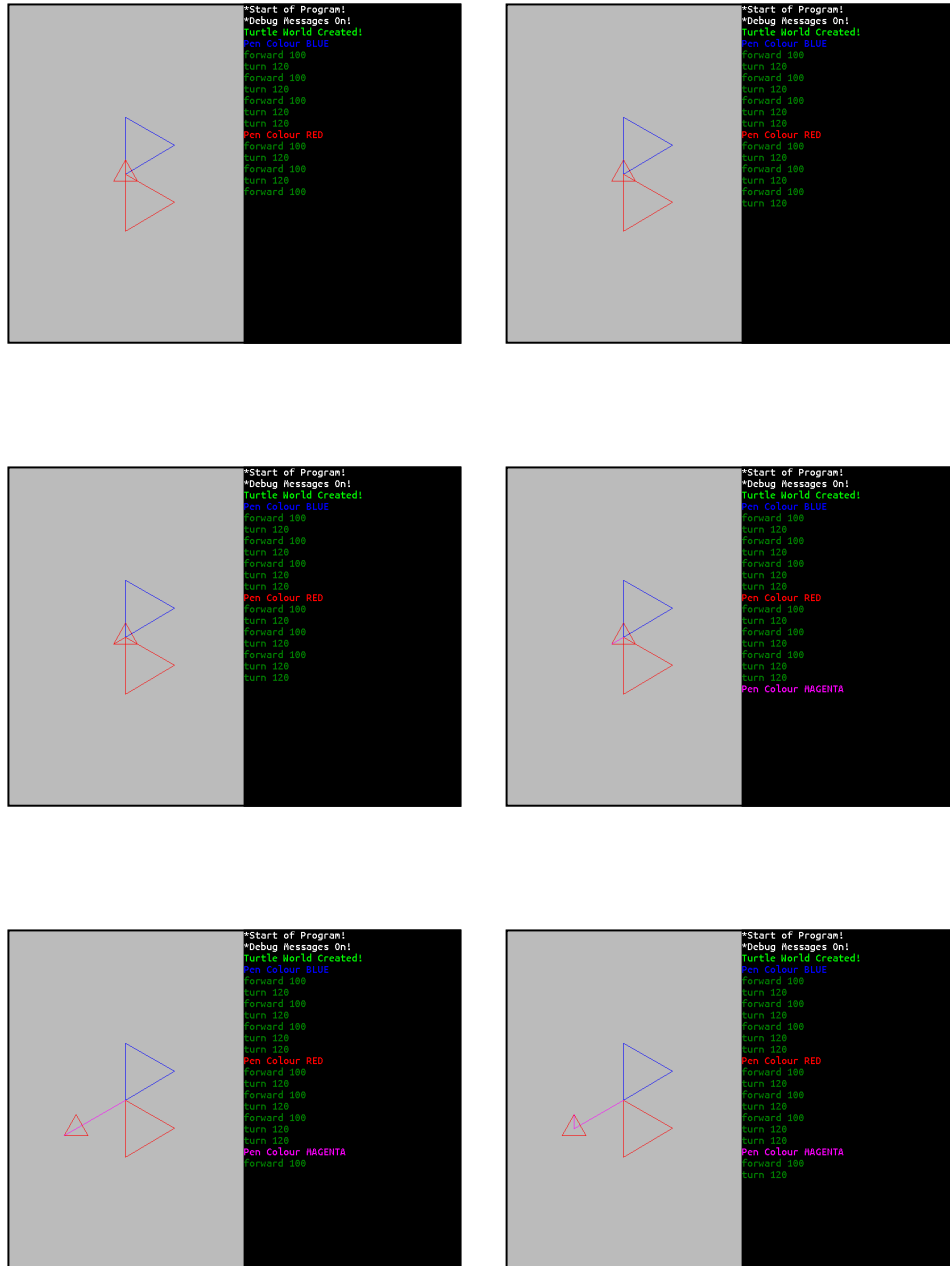


Figure D.13: PikPok Developers' Conference (slide set 13 of 21).

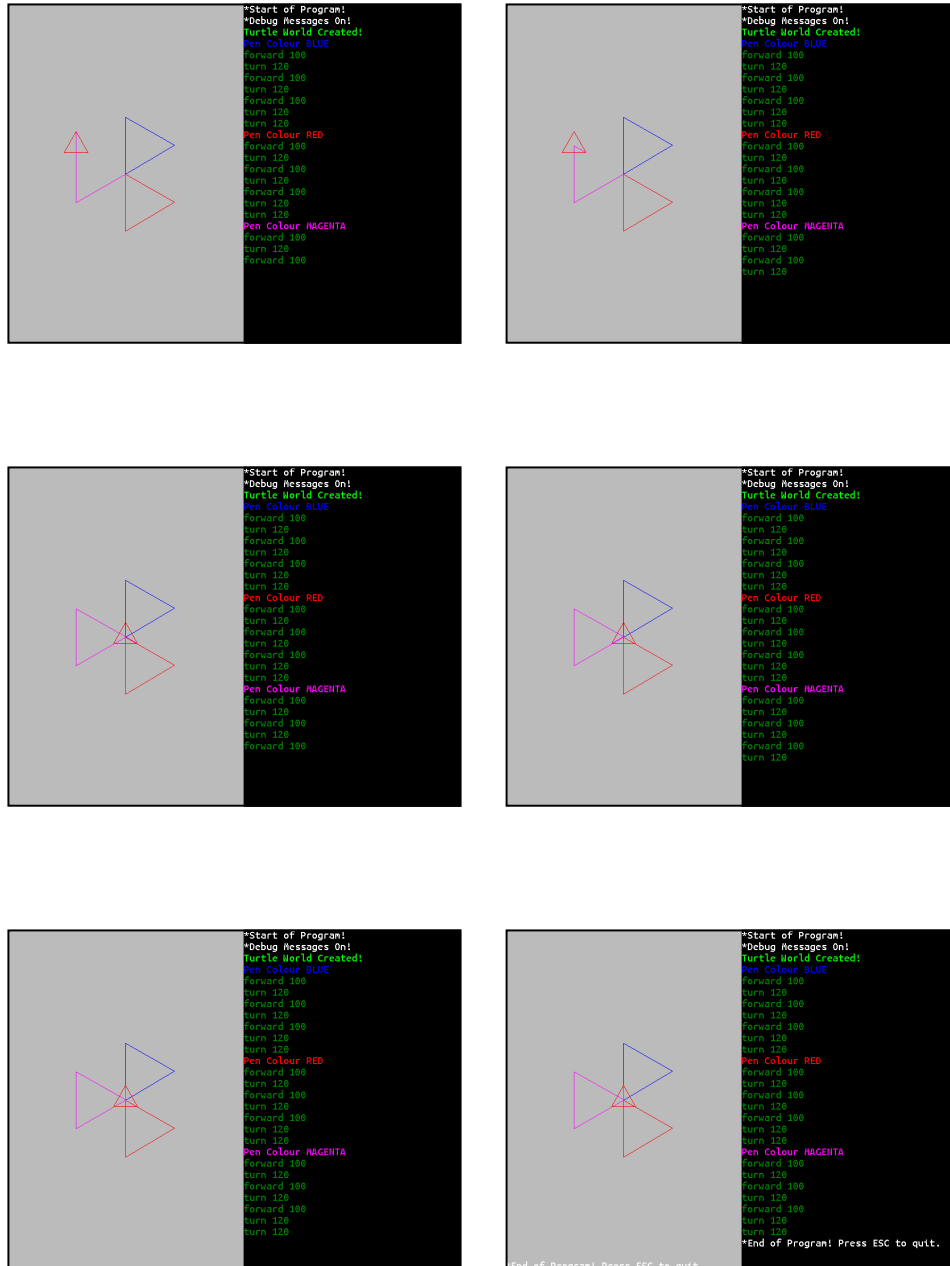


Figure D.14: PikPok Developers' Conference (slide set 14 of 21).

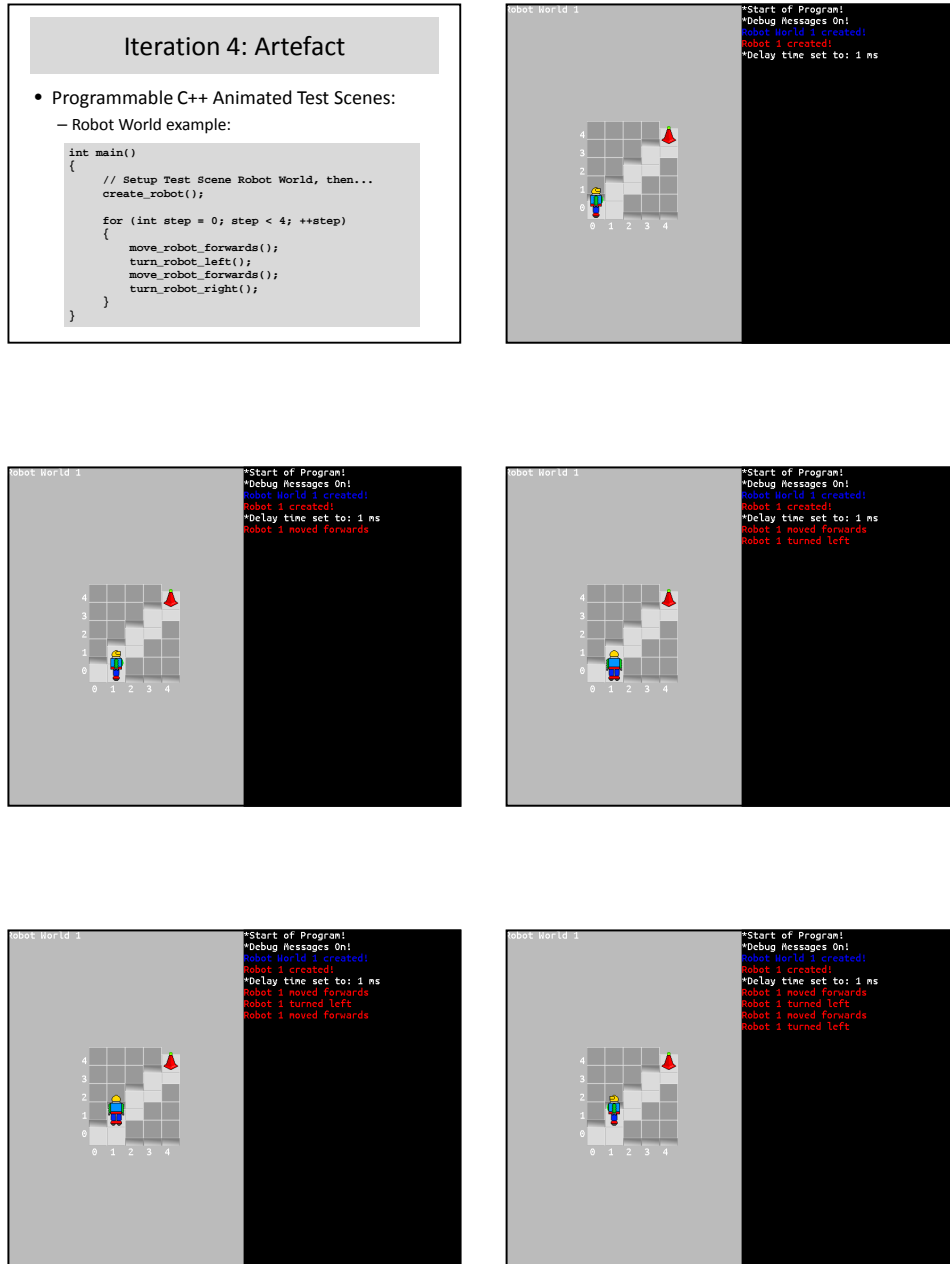


Figure D.15: PikPok Developers' Conference (slide set 15 of 21).

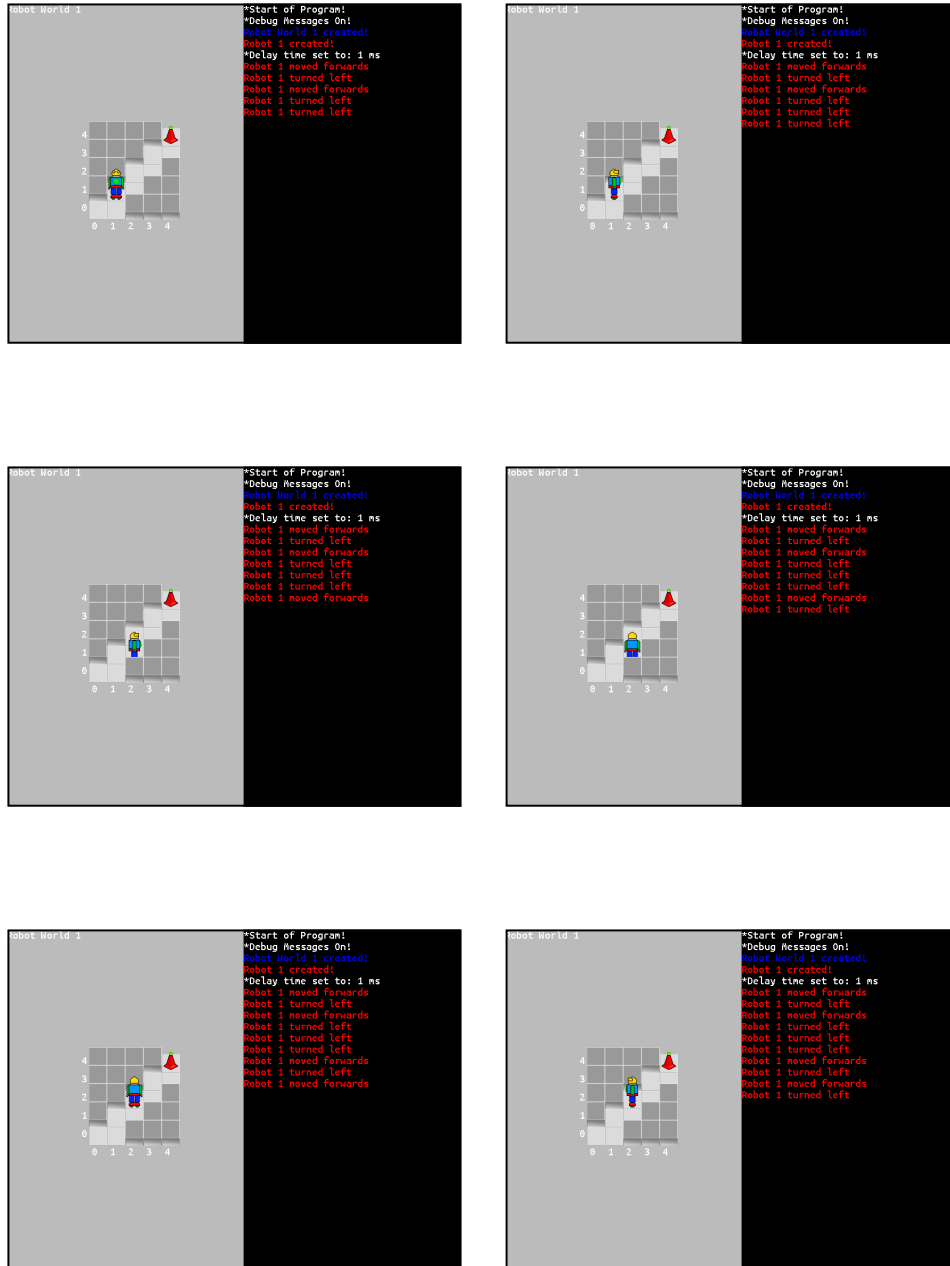


Figure D.16: PikPok Developers' Conference (slide set 16 of 21).

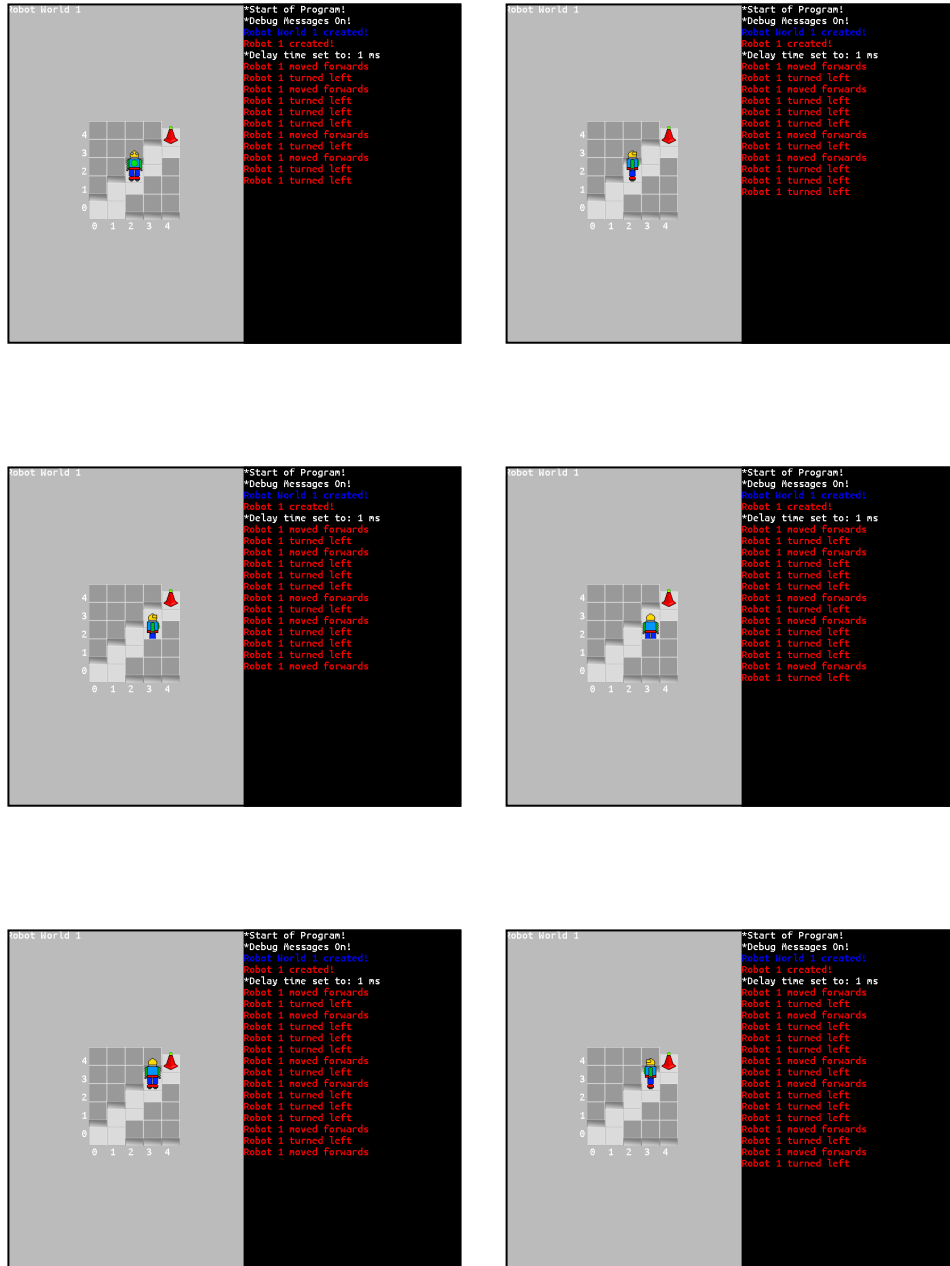


Figure D.17: PikPok Developers' Conference (slide set 17 of 21).

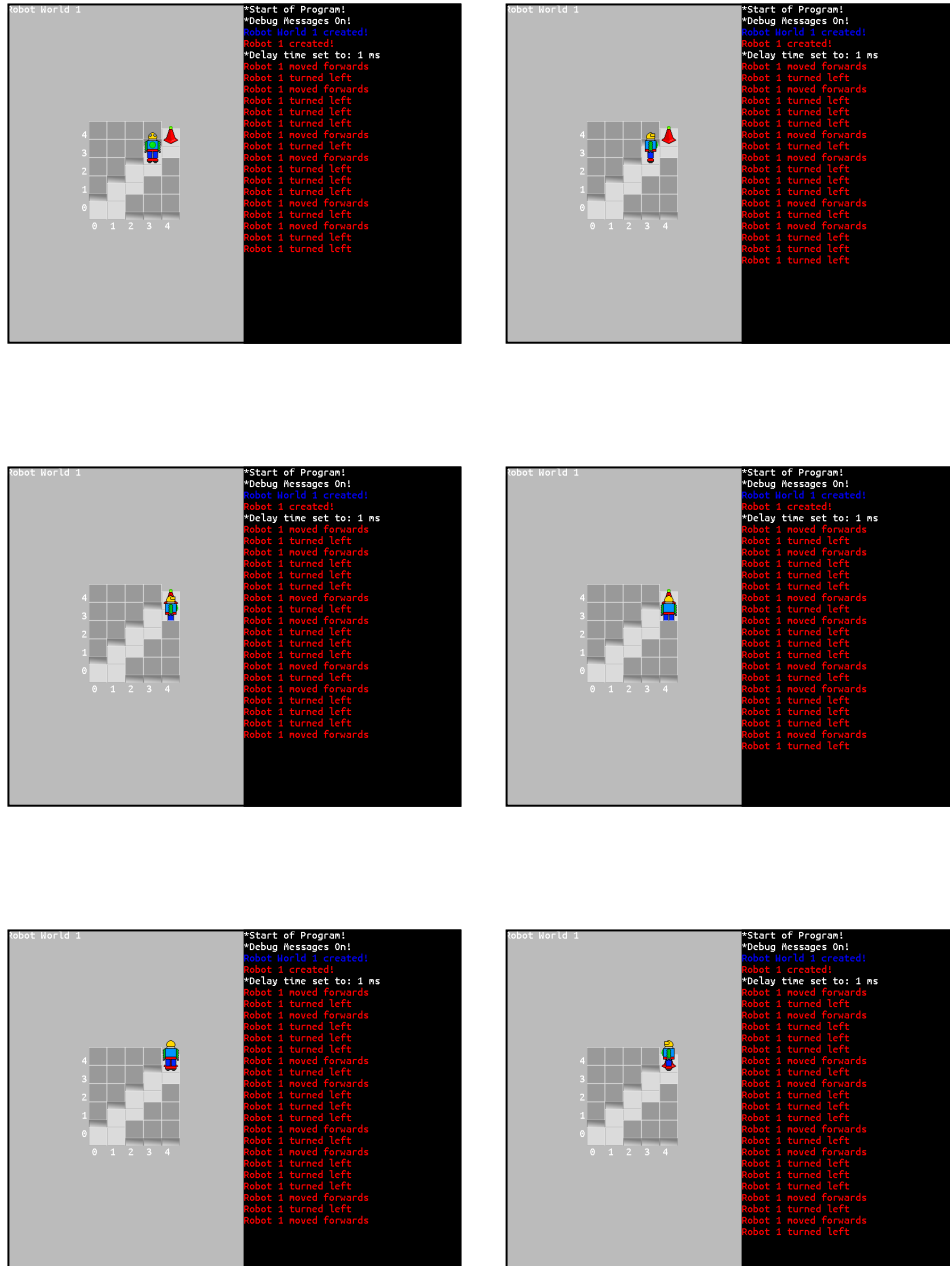


Figure D.18: PikPok Developers' Conference (slide set 18 of 21).



## Iteration 4: Micro-Evaluations

- DSR Suggestion Revisited:
  - Fixed time step animated test scene captures:
    - Iterate towards real-time game captures...
  - Multiple cross-platform project setups:
    - Platform build times:
      - SDL2.0.3 configuration on the Raspbian...
      - » Primary development target: Mac OS X.
  - Playable real-time game implementation:
    - Dynamic real-time input via human user...
    - Automated scene captures...

## Iteration 5: Artefact

- Two Target Platforms
  - Targets:
    - Windows 7 (SDL 2.0.3)
    - Mac OS X (SDL 2.0.3)
  - C++ Cross-Platform Real-Time Game Simulation Framework:
    - Space Invaders (Taito Corporation, 1978) clone.
  - Test Scene Replay System:
    - Deterministic game simulation.
    - Replay saved game state on multiple platforms.

## Iteration 5: Artefact

- Test Scene Replay System:
  - Record in-game events...
  - Generates a binary replay save file...
    - Platform independent...
    - Replay file is reloadable on either target platform...
  - Simulated real-time playback...
    - Deterministic time steps...

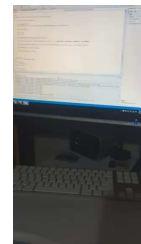


Figure D.19: PikPok Developers' Conference (slide set 19 of 21).

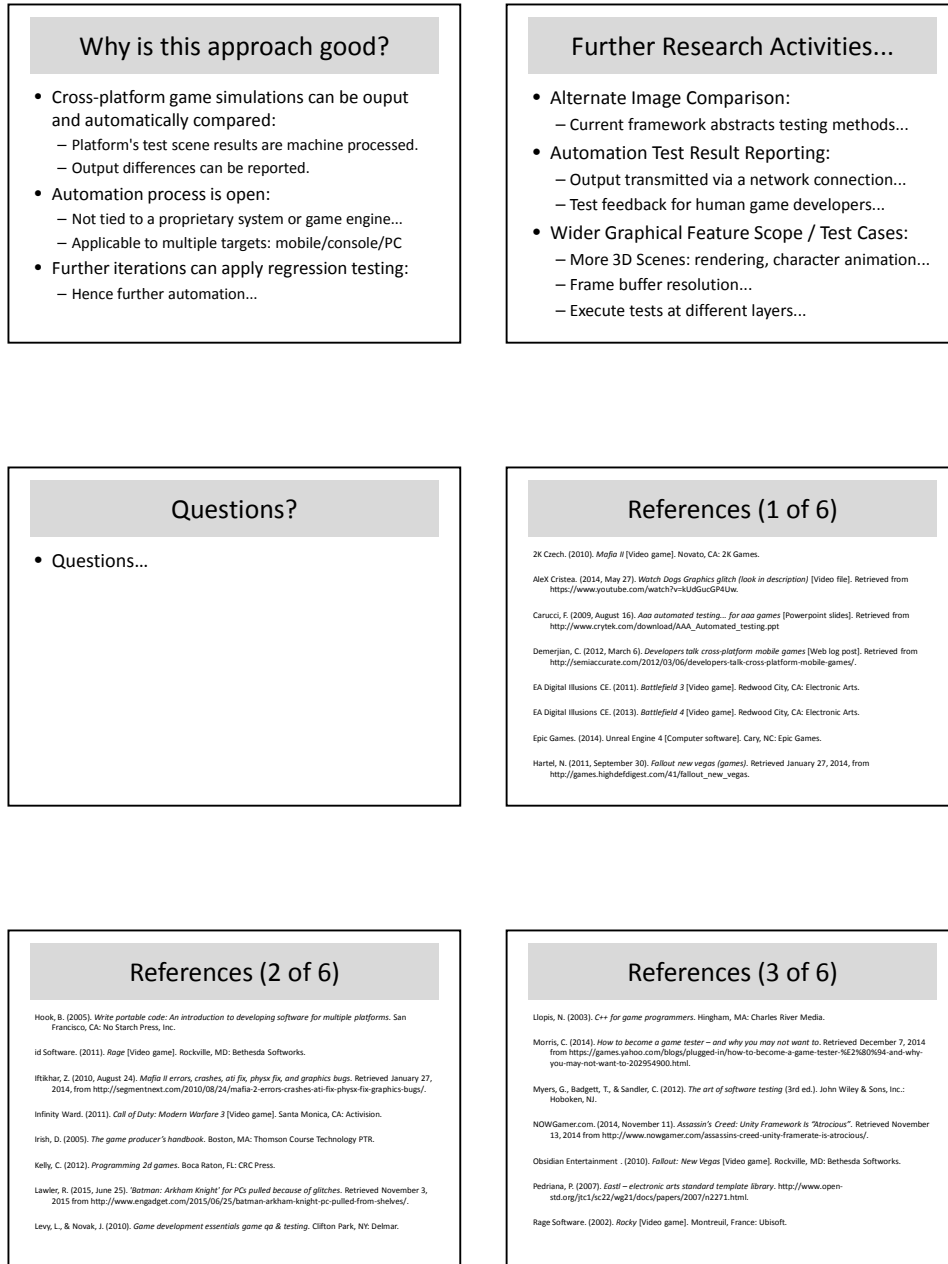


Figure D.20: PikPok Developers' Conference (slide set 20 of 21).



References (4 of 6)

Rockstar North. (2004). *Grand Theft Auto: San Andreas* [Video game]. New York, NY: Rockstar Games.

Rocksteady Studios. (2015). *Batman: Arkham Knight* [Video game]. London, England: Rocksteady Studios.

Scherstenleib, S. (2013). *Unlocking the potential of PlayStation 4: An in-depth developer guide*. <http://develop.scee.net/files/presentations/geunp2013/PaisGC2013Final.pdf>.

Sharkey, M. (2011, October 5). *Rage PC launch marred by graphics issues*. Retrieved January 27, 2014 from <http://pc.gamespy.com/pc/t6-tech-5-project/1198314p1.html>.

Staggs, M. (2013, November 28). *Battlefield 4 marred by connection issues and graphic glitches*. Retrieved December 20, 2013, from <http://uvuadu.com/2013/11/battlefield-4-marred-by-connection-issues-and-graphic-glitches.html>.

Thorn, A. (2011). *Game engine design and implementation*. Sudbury, MA: Jones & Bartlett Learning.

Timofteichik, A., & Nagy, R. (2012). *Verification of real time graphics systems* (Master's thesis, Chalmers University of Technology, Gothenburg, Sweden). Retrieved from <http://publications.lib.chalmers.se/records/fulltext/164580.pdf>

References (5 of 6)

Taito Corporation. (1978). *Space Invaders* [Video game]. Tokyo, Japan: Taito Corporation.

Ubisoft Montreal. (2014). *Assassin's Creed Unity* [Video game]. Montreal, France: Ubisoft.

Ubisoft Montreal. (2014). *Watch Dogs* [Video game]. Montreuil, France: Ubisoft.

Unity Technologies. (2013). *Unity Pro* [Computer software]. Retrieved November 27, 2013, from [https://store.unity3d.com/products/v18/feature\\_1\\_pro.en.html](https://store.unity3d.com/products/v18/feature_1_pro.en.html)

Vaishtal, V., & Kuechler, W. (2008). *Design Science Research Methods and Patterns*. Boca Raton, FL: Auerbach Publications.

Yezhovok, M., & Cantrell, M. (2011, October 25). *The 8 creepiest glitches hidden in popular video games*. Retrieved January 24, 2014, from <http://www.cracked.com/article/19507-the-8-creepiest-glitches-hidden-in-popular-video-games-p2.html>.

Yezhovok, M., Cantrell, M., & Rio, C. (2012, October 28). *The 6 creepiest glitches in famous video games (part 2)*. Retrieved January 24, 2014, from <http://www.cracked.com/article/20125-the-6-creepiest-glitches-in-famous-video-games-part-2.html>.

References (6 of 6)

Zioma, R., & Prankevicius, A. (2012). *Unity: ios and android: cross platform challenges and solutions*.

Figure D.21: PikPok Developers' Conference (slide set 21 of 21).