University of Rhode Island

DigitalCommons@URI

Open Access Master's Theses

2013

System for Video Game Enhancement Using Console Emulator State Information and Scene Recognition

Christopher J. Thibeaulty

University of Rhode Island, cjthibeault@teebo.com

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Recommended Citation

Thibeaulty, Christopher J., "System for Video Game Enhancement Using Console Emulator State Information and Scene Recognition" (2013). *Open Access Master's Theses*. Paper 37.
<https://digitalcommons.uri.edu/theses/37>

This Thesis is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu.

SYSTEM FOR VIDEO GAME ENHANCEMENT USING CONSOLE EMULATOR STATE
INFORMATION AND SCENE RECOGNITION

BY

CHRISTOPHER J. THIBEAULT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2013

MASTER OF SCIENCE THESIS
OF
CHRISTOPHER J. THIBEAULT

APPROVED:

Thesis Committee:

Major Professor Jean-Yves Herve

Gerard Baudet

Ron Hutt

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2013

ABSTRACT

Video game console emulators have long had a need to improve the quality of their graphical output to accommodate higher video resolutions and increased consumer expectations. Previous attempts at providing improved visual quality have relied on interpolation-based filters and, more recently, fullscreen raster-to-vector graphics conversion. Interpolation filters produce low-quality results at low computational cost. Conversion to vector graphics can produce high-quality results, but at a high computational cost. In all cases, these algorithms work on the full frame of graphical data at once, which can introduce errors in the result due to the inability of these algorithms to separate foreground graphics from background graphics.

This thesis investigates the modification of an existing emulator to achieve the goals of high-performance and higher-quality enhancement than what has previously been available. Two approaches were implemented and compared. The first uses traditional image processing techniques to perform object detection. The second uses a modified emulator to access virtual hardware state information to perform object detection. The results of object detection are used to perform replacement of all graphics with high-resolution replacements and to implement a form of scene recognition to add additional graphical and audio effects.

The system developed as part of this research uses a modified version of the FCEUX emulator for the Nintendo Entertainment System. The performance of the using virtual hardware state information as a means of object detection and localization has been found to be more efficient and accurate than using image processing algorithms. Using this system, a substantial audio-visual upgrade was successfully developed for a commercially-released game.

ACKNOWLEDGMENTS

I would like to thank Dr. Jean-Yves Hervé, my major professor, for his continuous guidance and advice throughout my research, as well as his willingness to assist with obscure typesetting issues. Additionally, I would like to thank the other members of my core committee, including Dr. Gérard Baudet, Professor Ron Hutt, and committee chair Dr. Mustafa Kulenović. Their time and contributions to this work are deeply appreciated.

Thank you to the Department of Computer Science and Statistics for providing me with funding and an office to serve as my home-away-from-home. I would not have been able to complete this work without their generous support. I would also like to thank the staff of the department, especially Lorraine Berube. She has cleared the fog on many, many occasions.

I would also like to acknowledge the work of Dr. Timothy Toolan of the Department of Electrical, Computer, and Biomedical Engineering. If it were not for his development and maintenance of the URI thesis class template for $\text{\LaTeX}$ I would surely have had to write all of this using Microsoft Word.

Finally, I would like to thank all of the members of my family for their patience and thick skin during the last couple of years. My parents, Alphonse and Deborah, for providing me with the opportunity to explore my many interests. My wife, Catherine, who has supported and tolerated me during all of this, and my two daughters, Kate and Addison, whose interest in old video games rekindled my own enjoyment of them and led me to pursue this topic.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGMENTS	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
1 Introduction	1
1.1 Motivation	2
1.1.1 Commercial Applications	2
1.1.2 Image Processing and Computer Vision Algorithms	3
1.1.3 Cultural Preservation	5
1.2 Problem Definition	6
1.3 Structure of the Thesis	6
2 Related Work	8
2.1 Filter Algorithms	8
2.1.1 Analog Simulation Filters	9
2.1.2 Traditional Scaling Filters	10
2.1.3 Vectorization Filters	14
2.1.4 The Limits of Interpolation	17
2.2 Object Detection	17

	Page
2.2.1 Detection	17
2.3 Image Hashing	22
2.3.1 General Hash Functions	23
2.3.2 Transformation-Invariant Image Hashing Algorithms	23
2.4 Histograms	25
2.5 Color Coherence Vectors	28
2.6 Related Work Summary	30
3 Hardware and Emulation	31
3.1 History	31
3.2 Description of the Hardware	31
3.2.1 Color Palettes	32
3.2.2 Display	33
3.2.3 Sprites	33
3.2.4 Background Graphics	34
3.2.5 Audio Channels	35
3.3 Description of the Emulator	36
3.3.1 Visual Enhancements	36
3.3.2 Audio Enhancements	37
3.3.3 Debugging Facilities	37
3.3.4 Random Access Memory	39
3.3.5 Summary	39
4 Description of the System	40
4.1 Software Components	40

	Page
4.1.1 Emulator	41
4.1.2 Game Binary Image	41
4.1.3 Configuration Data	42
4.1.4 Art Assets	42
4.1.5 Audio Assets	43
4.1.6 Redistributable Package	43
4.2 Development Roles	43
4.2.1 Zero-Effort Roles	44
4.2.2 Fixed-Effort Roles	44
4.2.3 Variable-Effort Roles	44
4.2.4 Workflow	45
5 Object Detection Implementation	46
5.1 Modifying the Emulator for Game Enhancement	46
5.1.1 Initial Analysis of the Emulator	46
5.1.2 Information Available to the System	47
5.1.3 User Configurable Behavior	48
5.2 Sprite Detection	49
5.3 Background Detection	51
5.3.1 Video Output Buffer	51
5.3.2 Preparation	53
5.3.3 Tile Recognition	53
5.4 Post-detection Filtering	59
5.5 Scene Detection	62

	Page
6 Scene Recognition and Event Detection	63
6.1 Events	63
6.1.1 Triggers	64
6.2 Scenes	64
6.2.1 Marking Procedure	65
6.2.2 Representation and Recognition	65
6.3 Non-interfering Reaction	68
6.3.1 Sound Events	69
7 Discussion of the Results	71
7.1 System Feasibility	71
7.1.1 Modifying an Emulator	71
7.1.2 Specific Features Implemented	72
7.2 Object Detection	72
7.2.1 Sprite Detection Results	73
7.2.2 Applicability to Other Emulators	74
7.2.3 Tile Detection Results	75
7.2.4 Weaknesses of the Image Processing Approach	76
7.3 Event Detection Results	81
7.4 Scene Recognition Results	81
7.5 Discussion Summary	86
8 Conclusions and Future Work	87
8.1 Summary	87
8.2 Suggestions for Future Work	88

	Page
8.2.1 Further Development on FCEUX	88
8.2.2 Other Hardware Sprite Engine Emulators	89
8.2.3 Semi-Automated Image Retrieval	89
8.2.4 Non-Raster Image Replacement	90
LIST OF REFERENCES	91
BIBLIOGRAPHY	96

LIST OF TABLES

Table		Page
1	The number of pixels belonging to each component is counted. Components C and D each contain less than $\tau = 4$ pixels, and so those pixels are considered incoherent.	29
2	The color coherence vector for the image in Figure 9, presented in table form. For each color, α represents the number of coherent pixels, and β represents the number of incoherent pixels.	30
3	Memory layout of an entry in the Object Attribute Memory. One entry exists for each sprite that is on-screen during a frame	50
4	Required configuration values for FCEUX 2.14a. The choices for these values are restricted to ensure that the current implementation of the system works as expected	52
5	Comparison of hardware sprite engines available in popular video game consoles. This details the total number of hardware sprites available, how many sprites can be rendered on a single scanline, the number of colors per-sprite, and possible sprite sizes	74
6	Comparison background layer bitmap support in popular video game consoles. This details the number of background layers, limitations per layer, and comments about additional features or limitations, such as whether the hardware supports scaling and rotation of those layers	80

LIST OF FIGURES

Figure		Page
1	(a) Example of aliasing in pixel art. This image has had a 3×3 nearest-neighbor upscaling filter applied. Notice that diagonal lines have a step-like appearance, which is undesired. (b) The same image scaled using the <code>hq3x</code> filter to reduce aliasing.	11
2	Examples of how the <code>Scale2x</code> , <code>Scale3x</code> , and <code>Scale4x</code> filters affect a piece of black and white pixel art. Examples are scaled to uniform dimensions for comparison.	13
3	Comparison of how several $4\times$ scaling filters perform on color pixel art with many straight edges. (a) shows the original image. (b) through (g) show pixel domain scaling filters. (h) through (j) show vector filters. .	15
4	Comparison of several of several types of pixel art scalers on an image with smooth contours. The original 16×16 image is scaled $4\times$ using nearest-neighbor interpolation, <code>hq4x</code> , and Kopf-Lischinski vectorization.	16
5	(a) The Haar wavelet at various transformations. (b) Haar-like features. White areas represent positive, black areas represent negative.	19
6	Example of how two Haar-like features are used to measure the difference between characteristically light and dark regions in a facial image of the training set using the Viola-Jones object detection framework. Each feature is computed over multiple scales and translations along the image to identify matching regions. Those regions are then subjected to the next classifier in the cascade.	21
7	(a) A model of an object consisting of nine points. (b) An arrangement of points in a scene to be used for querying. (c) A match of six points has been found, with the query being rotated seventy-two degrees clockwise.	25

Figure		Page
8	A set of color histograms computed from a digital photograph using Adobe Photoshop CS4. These histograms use a very large number of buckets, but still represent an approximation of the color distribution in the original image as the colors were sampled to a discrete color space when the photograph was taken. Histograms have been colorized to visualize how the color intensities are distributed. (a) Original image. (b) Red histogram. (c) Green histogram. (d) Blue histogram. (e) Luminance histogram.	27
9	Computing a labeling of the connected components for a color coherence vector. The grayscale intensity values are placed into three buckets that each contain a range of ten values. The 1's bucket is made up of pixels from two disconnected regions, as is the 2's bucket. These are given distinct labelings, and so from these three buckets, labels <i>A–E</i> are obtained.	29
10	RGB approximation of the NTSC palette used by the NES. The eight right-most color values (appears black in the image above) are reserved and not used.	33
11	An example of a composite sprite for use with the NES hardware sprite engine. Due to limitations of the engine, this sprite was made of several sprites that have been overlaid on top of each other. The body and head were each made of several 8×16 sprites that shared one palette. To draw the face, two additional colors were needed (tan and white), and so the face was a separate set of two 8×16 sprites, using a different palette.	34
12	The layout of the name tables in the NES. The portion that the player sees can be scrolled horizontally or vertically. (a) The player view is not scrolled. (b) The player view has been horizontally scrolled and is shown parts of two name tables.	35
13	(a) Picture Processing Unit (PPU) Viewer and (b) Name Table Viewer debugging screens. The PPU Viewer shows the two “banks” of image data, prior to palette index lookups. The Name Table Viewer shows the layout of the background layer in memory.	38
14	Workflow diagram showing the components and roles in the system.	45

Figure		Page
15	Source code snippet of <code>Fixit1</code> , a function inside <code>ppu.cpp</code> . The developer's original formatting has been preserved. The function does not describe what it fixes, nor is it obvious from the code itself. The FCEUX source code has a large amount of code written in a similar manner.	47
16	A sample of configuration information stored in the Microsoft Windows INI format.	49
17	The game screen is common divided into two regions: a game play region and a player feedback region. In (a), the player feedback region appears below the play region. In (b), it appears above the play region.	54
18	A panel of fully saturated colors, including individual RGB channels, combinations of two channels, and a gradient is processed using several different grayscaling formulas. (a) The original RGB panel (b) Gleam (c) Intensity (d) Luma (e) Luminance (f) Lightness (g) Value (h) Luster (i) Decolorize.	56
19	Two 16×16 tiles are shown with different types of label assignments. (a) and (f) are the original tiles. (b) and (g) use per-quadrant color averaging. (c) and (h) use per-quadrant grayscale averaging. (d) and (i) use full-tile color averaging. (e) and (j) use full-time grayscale averaging. Note that the quadrant based averaging produces tiles that, while similar, have clear differences due to the use of spatial information. Full-tile averaging produces indistinguishable results for both tiles.	57
20	The grayscale average of each quadrant is a value between 0 and 255. It can therefore be stored in eight bits. This means that the grayscale averages for all four quadrants can be packed into one 32-bit word.	58
21	Two tiles that have been artificially constructed to produce identical tile labelings. The tile in (a) produces the same average pixel color for each quadrant as tile (b).	60
22	The post-detection filter grid. The current tile is located at position (i, j) . This filter is swept across every position in T checking for matches at (i, j) and one or more of the neighbor elements. If a match is found, the value at (i, j) will be replaced.	61

Figure		Page
23	Example of how post-detection filtering can be used to differentiate two tiles which, while identical in the source image, should have different representations in the final output. In (a), there are identical 16×16 tiles stacked on top of each other. These tiles represent a wall. In (b), the two tiles have been provided with different labels as a result of post-detection filtering.	61
24	In this set of screenshots, (a) the upgrader has identified a part of the game that represents a scene. (b) A key is pressed, and the upgrader is presented with a file dialog to save the scene tag information to disk. In this case, the the vector $\vec{t}$, representing the scene displayed in (a), is associated with the tag “InTheCave.” Each $\vec{t}_i$ is a packed 32-bit word built from four 8-bit numbers, each of which represents the intensity of the average color in that quadrant.	66
25	Diagram showing several different orientations for background layers. In (a), a sprites are layered on top of a single background layer. In (b), multiple background layers, each with independent alpha transparency and scrolling are present. In (c), some of the background layers have transformations applied (rotated and scaled).	77
26	Two SNES games using Mode 7 in an unexpected way. In (a), the large enemy in the air is actually a Mode 7 background, with support for scaling, translation, and rotation. Everything else on the screen is rendered as a hardware sprite, including the castle platform. In (b), Mode 7 is used for both halves of the screen, with different transformation matrices applied per scanline.	78
27	The SNES game <i>Star Fox</i> used the SuperFX chip to provide primitive 3D polygon rendering in addition to hardware sprite rendering on Mode 7 background graphics. Games that use co-processors to perform graphics work would require additional modifications to the emulator to extract the relevant data.	79
28	(a) Original title screen. (b) Scene recognition is used to perform a complete replacement of the entire viewable display area. This example also plays a modern version of the game’s theme song. . . .	82
29	(a) Original character selection screen. (b) Scene recognition performs fullscreen replacement of the background layer graphics. Sprites are rendered over the overlay. The original game did not feature music on this screen, but this example uses scene recognition to add a musical background.	83

Figure		Page
30	(a) Original first playable game screen. This is the first time in which the player has control of the protagonist character. (b) Background layer graphics have been rendered using the image processing system for tile detection. The scene recognition engine has been used to start the game music and provide subtle graphical overlays above the sprite layer.	83
31	(a) Original cave scene. (b) This screen makes extensive use of the scene recognition system. Upon scene detection, the system adds music and a looping fire sound effect (the original game had neither of these), replaces all background graphics with a pre-rendered background, and plays a one-time sound effect that is a man speaking the words that appear in the original game as text. In addition, after the system has rendered the sprite layer, scene recognition is used to provide a fog overlay and a world overlay, allowing a new sense of depth to the cave.	84
33	(a) Original world map screen. (b) The scene recognition system detects the presence of water, rocks, and fireballs and generates appropriate sound effects. It is also used to detect when the aquatic enemy pops up and plays an appropriate splash sound. The original game did not feature any of these sounds.	85
34	(a) Original “outside dungeon” map screen. (b) The scene recognition engine is used to provide graphical replacement of much the background layer and a graphical world overlay. Appropriate sound effects are provided in response to on-screen action. This scene has been configured to change the music to the overworld theme music if it is not already playing.	85
35	(a) Original dungeon map screen. (b) The scene recognition system engine is performing full replacement of the background graphics layer as well as providing a graphical overlay and fog. It has change the music and looping sound effects in response to the change in environment.	86

CHAPTER 1

Introduction

As faster, network-enabled gaming systems and portable devices have become available and widely adopted by the public, there has been a resurgence of interest in the republishing of older video game titles. This has been accomplished by porting the existing game code (or rewriting the game entirely) to run on newer hardware platforms, or by allowing the original code to run in an emulated environment.

Creating a port of older games to run on modern game consoles is an expensive task. The current average annual salaries for employees involved in professional game development range from fifty thousand dollars for quality assurance testers to eighty-six thousand dollars for programmers [1], and writing a port would require a team of developers. The ideal porting project would start from a well-written source code base that has been thoroughly commented and documented. For many older games, this is not available. Games were written in hardware-specific assembler code, often using undocumented hacks to push the hardware beyond its intended limits. In many cases, the source code or documentation for games have disappeared over the years [2].

Despite the obvious costs and difficulties involved with writing a port of a game, there is a very strong advantage in that the developers have total control of the game behavior. The developers may add new features, modify existing features, correct bugs, and change large portions of the game mechanics. This flexibility comes with a cost, as these changes may significantly alter the feel of a game, whether by design or as a result of not accurately recreating the behaviors of the original code.

As an alternative to porting game code, several game console manufacturers have begun to provide software-based emulators of earlier game consoles on their newer consoles. These emulators load a binary file of the Read-Only Memory (ROM) data of the

game, and simulate how it would behave on the original hardware. This has the advantage of exactly preserving the look and feel of the original game [3]. It is very cost effective, as the only costs involved are marketing and distributing the binary over some medium. As the emulator has been written prior to republishing the game, there are no new development costs. Because it is emulating the original hardware to play the ROM, the primary disadvantage of emulation is clear: the developers are limited to the capabilities of the hardware for which the game was originally designed to be executed on. For many older game consoles, these limitations are substantial, with low-resolution, limited color graphics and primitive sound generators.

1.1 Motivation

1.1.1 Commercial Applications

There are numerous opportunities for commercial application of the system being proposed. Several major manufactures of game consoles provide fee-based services that allow consumers to purchase and play older games through software emulation. Examples of these services are Nintendo's Virtual Console on WiiWare [4], and Sony's PSOne Classics on the PlayStation Network [5]. These emulators provide a strict emulation service. They do not make attempts at altering the visuals of a game, with the sole exception being sprite alterations to deal with expired trademark licensing. Prices are generally set based upon which console the games were originally released for, and follow the rule of "older is cheaper." An obvious reason for this is that the earlier game consoles had substantially more limited graphical and audio capabilities, which potentially limits enjoyment of the game [6], and so it is difficult to ask the consumer to pay a premium price.

By significantly reducing development and testing expenses, this system will allow these publishers to cost-effectively upgrade the way their games look and sound while maintaining existing electronic publishing channels. Publishers will be able to ask for prices that are comparable to the those of games that were written for more recent generation

game consoles, increasing revenue.

As consumers are now playing games on more varied devices, such as traditional consoles, phones and tablets, there may be a need to customize the experience for each device. Emulators for many game systems are available for these devices, and the configurable mapping of the proposed system will allow publishers to customize the output for a number of devices and needs. Potential uses include providing higher-contrast graphics, graphics designed to optimize battery life, graphics designed for better visibility on smaller screens, and customizations for users with visual disabilities.

While publisher-based emulation services provide a low-cost way of republishing older games, it is not always possible due to licensing agreements that may have expired. At present, the only options in this situation are to relicense the characters, trademarks, or other elements in question, attempt to patch the sprite data in the binaries to avoid the licensing issues, or to decide to not republish the game. The proposed system allows the publisher to easily perform a replacement of problematic visuals in the original game, making the licensing a non-issue.

1.1.2 Image Processing and Computer Vision Algorithms

After establishing the feasibility of developing such a system, one must begin the task of developing a functioning implementation. Conventional wisdom suggests that algorithms from the fields of image processing and computer vision may be used to implement portions of the system.

Optimal Detection

Games generally must provide real-time feedback to the player. Any system that processes each image frame must be able to complete its work before the next frame is requested. Further, emulators are increasingly popular on battery-powered devices, which are limited in computational power. High CPU utilization can quickly drain the battery on

such devices. Therefore, it is important that any algorithm implemented in the system be highly efficient. There are very efficient algorithms that have been developed, often with a primary focus on face detection, such as the Viola-Jones object detection system [7]. While these algorithms are faster than what was previously available, they can still be computationally intensive. Further, many game consoles were supplied with hardware that could apply several forms of geometric transformations to images within the game. Algorithms that are invariant to these transformations exist, but these are also computationally intensive. Therefore, it is important look for alternative means for obtaining this information to reduce or eliminate the additional overhead.

Vision Analysis vs. State Analysis

The problem being studied involves the enhancement of a video game that is being run within emulation software. A unique property of this problem is that we have access to a perfect model of the behavior we are trying to analyze, in the form of virtual hardware state information. Without implementing any image processing or vision algorithms whatsoever, all of the goals of the system are achievable by accessing this state information. It is possible that there may be a high labor cost in modifying an emulator to support this functionality, due to each game's unique ways of interacting with the hardware. Retrieving the required information in this manner is possible, but may not be practical. A computer vision system can also obtain all of the required information through analysis of the display, but at a much higher runtime computation cost. One of the goals of this research is to either find a balance between the two methods that allows for a high accuracy detection system with low computational cost, or to develop a recommendation of one method over the other.

Scene Awareness and Event Recognition

In the process of analyzing each frame of the game’s visual output, there exists more than just a simple opportunity to replace individual graphics with a higher quality substitute. It should be possible for an algorithm to look at a set of frames and determine the game’s current environment. The environment will consist of state information regarding different ‘scenes’ of the game, along with a mechanism for recognizing what is happening on-screen. This information can be used to further inform the renderer’s algorithm.

Scene recognition algorithms have been developed for analyzing image content for the purposes of labeling or annotation in static scenes [8][9]. Examples of such algorithms also exist for scene recognition in video [10][11]. Further, event recognition systems have been developed that utilize video of the player as a method of controlling the game, such as in Microsoft’s Kinect.

Games often have a number of events, in which some action occurs due to the event or timer being triggered. Often, the action is limited to a sprite appearing or disappearing, or a single sound effect being played. The proposed system can utilize an event tracking and recognition system that will allow for additional events to be tracked, along with more visual and audio feedback to the player.

The goal of this portion of the system is not to simply label the scenes, but to use that labeling to provide partial or complete enhancement of the original scene.

1.1.3 Cultural Preservation

There is a trend towards video games being considered not just as temporary entertainment, but as cultural artifacts worthy of preservation [3], much as is the case for music and film. Unfortunately, similar to the music and film industries, much of the early history of video games has been poorly preserved and often lost or destroyed [2].

Emulation has proven successful as an avenue for preserving older games and associated hardware [3]. Much of the demand has been driven by nostalgia. With each

passing generation, the graphical standards by which games are judged has steadily risen. As the consumers who obtain the games for nostalgic reasons gradually leave the marketplace, it is likely that this newer generation of gamers may reject the older games, partially based upon the poor game visuals. Greater visual quality in games has been shown to increase sensations of telepresence, leading to higher levels of enjoyment [6]. The proposed system addresses this by providing updated visuals for these games, which may be more palatable to modern gamers, thus increasing their distribution.

1.2 Problem Definition

The problem shall be defined as that of developing a system for the purpose of enhancing the audio-visual output of a video game that is running in an emulator, utilizing state information in the emulator to increase efficiency and accuracy. The problem can be divided into the following sub-problems:

1. *Viability*: Demonstrating that such a system can be developed, and that it can provide better results than methods that have been previously developed.
2. *Object detection*: A method for successfully detecting on-screen game elements reliably and efficiently.
3. *Event detection and scene recognition*: A method for determining when specific events occur, and using this knowledge to begin an action in response to the event.

1.3 Structure of the Thesis

Chapter 2 will provide an overview of the work that has been previously performed in this field. Topics include upscaling filters, object detection algorithms, and algorithms for comparing images for similarity. For each area of related work, a discussion of the disadvantages of that work will be provided.

Chapter 3 will briefly discuss the design and limitations of the original console

hardware of the Nintendo Entertainment System, as well as provide an introduction to the software emulator FCEUX. This will be followed by a detailed discussion of the types of information that could be extracted from the emulator's state information, its suitability for use in enhancement, and the advantages and disadvantages of this approach.

Chapter 4 gives an overview of the system developed as part of this research. It includes discussion on the individual software components required the system, as well as a breakdown of the different human roles involved in producing an enhanced game.

In Chapter 5, object detection techniques will be presented for efficiently detecting on-screen game elements. This includes discussion on image processing techniques as well as using virtual hardware state information as a means of object detection.

The focus of Chapter 6 will be on using the development of an event recognition system that can be used to recognize game scenes and events as they occur. This will include discussion about the types of events that can be detected, as well as what kinds of interference-free actions that can be taken in response to those events.

The discussion of the results obtained in testing an implementation of this system on a commercial game are discussed in Chapter 7. Finally, a summary for the work and suggestions for future work are presented in Chapter 8.

CHAPTER 2

Related Work

This chapter surveys the work that has been done previously in the area of enhancing emulated video games that were originally designed to run on hardware sprite engines. This is followed by a discussion on algorithms in the fields of image processing and computer vision that can be used to extract feature information from the game’s visual output.

2.1 Filter Algorithms

Emulators are frequently used on home computers to emulate commercial game console hardware. The screen resolutions of the display devices used on these computers is typically much higher than the resolution available in the original console hardware. For example, it is very common now for computer monitors to support 1920×1080 display resolutions, but early game consoles often had resolutions closer to 320×240 . While early emulators were typically full-screen applications that were launched from a command line, and used a custom screen resolution at runtime, emulators now commonly run in a windowed environment. To provide a larger view of the game screen, emulators implement algorithms that upscale the game’s visual output to a larger size.

Definition 2.1.1 *Digital Filter*

A *digital filter* is a process or algorithm that removes an undesired component from a digital signal. In this thesis, the signal component of interest is aliasing, which can produce jagged edges in the signal.

A number of filter algorithms have been developed for enhancing the visual output in game console emulators. Here, they are divided into three categories. The first set of filters attempt to emulate characteristics of older display hardware, such as Cathode Ray Tube (CRT) televisions. The second set of filters perform upscaling in the pixel domain.

Finally, a filter that transforms images from the pixel domain to the vector domain is discussed.

2.1.1 Analog Simulation Filters

Most older game console hardware provided analog video output compatible with the standards set by the National Television System Committee (NTSC). Some game consoles were modified to use Phase Alternating Line (PAL) instead of NTSC, primarily in Europe. The video signal would be sent to a standard CRT television. Unlike today's digital display and delivery mechanisms, these analog systems produced a number of visual artifacts (errors in the video signal). While these artifacts are generally considered undesirable, many games took advantage of some of these distortions to produce special effects, such as pseudo-transparency. Some emulators provide filters that attempt to simulate some of these artifacts.

The most well-known NTSC-CRT emulation filter is the algorithm provided in the Blargg NTSC Library [12]. This filter simulates many of the visual artifacts of the analog chain, including CRT barrel distortion, scanlines, phosphor masks, blurring, bloom, and color fringing. Recent revisions of the algorithm have brought improvements in performance, but the algorithm remains computationally expensive. It is most often implemented as a pixel shader program, and requires a Graphical Processing Unit (GPU) to perform the processing.

NTSC filters do not generally perform any form of scaling on their own. They may be used in conjunction with filters specifically designed for upscaling. Additionally, the results produced by these types of filters may not be considered to be an enhancement of the image as it degrades the output, but it degrades in a way that enhances the game play experience.

2.1.2 Traditional Scaling Filters

The filters discussed here are designed to scale an image by some integral amount. Images begin and end as a rectangular region of pixels. The algorithms used in these filters are all forms of interpolation.

Definition 2.1.2 *Interpolation*

Interpolation is a method of reconstructing missing data points by computing an approximate value based on nearby, known data points.

Source-unaware scaling filters are found in many software packages, including image editing software. These filters, which are referred to here as general-purpose scaling filters, do not take any information about the nature of the source image into consideration. Some algorithms, such as those designed to process pixel art, have some awareness of the characteristics of those types of images. The algorithms take advantage of that information to produce better scaling results for those images than could have been obtained otherwise.

General-Purpose Scaling Filters

A number of common scaling filters have been used in emulators for enlarging the display size. Nearest-neighbor interpolation filters are present in most emulators. This algorithm is very fast and can be used to double or quadruple the size of the image without introducing new colors or artifacts into the image. The color value of a pixel in the scaled image is dependent upon a single pixel in the original image. This algorithm is preferred by some game players who want to see the original image preserved as much as possible, but rendered larger than the existing source resolution. As nearest-neighbor interpolation does not make any attempt to smooth the image, the resulting image can have a very blocky or jagged appearance due to aliasing, as can be seen in Figure 1(a).

Bilinear and bicubic filters attempt to resolve this problem by interpolating the color values between pixels in the source image. This produces an enlarged image with smooth

Figure 1: (a) Example of aliasing in pixel art. This image has had a 3×3 nearest-neighbor upscaling filter applied. Notice that diagonal lines have a step-like appearance, which is undesired. (b) The same image scaled using the hq3x filter to reduce aliasing [13].

color gradients approximating the missing pixel colors. These filters will work well on images in which the source data represents continuous color shifts. For pixel art, or any other type of image that has hard lines or contours, the smoothing effect of these filters may produce an image that appears blurry [14].

Pixel Art Scaling Filters

Game systems that used hardware-based sprite engines typically featured low-resolution graphics of limited color depth, and so it was important to develop scaling algorithms that worked well with these types of images. The individual graphical components used in these games are known as *pixel art*, a term coined by Adele Goldberg and

Robert Flegal at the Xerox Palo Alto Research Center in 1982 [15], although the use of this type of digital art dates back nearly a decade earlier (1973) to the Robert Shoup’s SuperPaint software package [16].

Definition 2.1.3 *Pixel Art*

Pixel art is a raster-based digital image created by editing each individual pixel without the use of automatic tools.

The efficiency of algorithms for scaling pixel art was important, as the scaling would need to be performed approximately sixty times per second [17]. To achieve this goal, the filters did not generally allow for arbitrary scaling. The filters were most commonly written for $2\times$, $3\times$, and $4\times$ scaling.

Eric’s Pixel Expansion (EPX) was developed by Eric Johnston at LucasArt in 1992. He created it to port a game engine, which ran on an IBM PC at 320×200 with up to 256 colors, to the Apple Macintosh Color Classic, which was running at nearly double the resolution [18]. This algorithm worked by looking at the four non-diagonal neighbors of the source pixel. The four destination pixels would be computed based on the similarity of the color values of the neighbor pixel pairs in that direction. If three or more of the neighbors were identical, the original pixel color value would be used for all four destination pixels [19].

The AdvMAME $2\times$ /Scale $2\times$ filter was developed in 2001 by Andrea Mazzoleni. It was functionally identical to EPX, but had a more efficient implementation. The AdvMAME $3\times$ /Scale $3\times$ filter works by the same principles as EPX, but achieves $3\times$ scaling by using the color values of all eight neighbors of the source pixel to compute a 9×9 result [20]. The AdvMAME $4\times$ /Scale $4\times$ filter is simply the EPX filter applied twice to achieve a $4\times$ upscaled result [20]. Using this family of scaling filters results in an excessive amount of rounding of sharp edges, as can be seen in Figure 2.

The Eagle filter was created by Dirk Stevens, and it provided $2\times$ scaling. The first step

Figure 2: Examples of how the Scale2 $\times$, Scale3 $\times$, and Scale4 $\times$ filters affect a piece of black and white pixel art [20]. Examples are scaled to uniform dimensions for comparison.

of the algorithm was to enlarge the image 2 $\times$ using a nearest-neighbor interpolation filter. Each pixel is then processed by looking at the neighbors in each diagonal direction, along with the two pixels that are adjacent to both the neighbor and the original pixel. If the three pixel colors match, the new pixel is assigned that color. A disadvantage of this filter is that isolated pixels on a solid background color will disappear. This issue can be seen in Figure 3(e) on Page 15.

The 2 $\times$ SaI (2 $\times$ Scale and Interpolation) filter is a upscaling filter developed by Derek Liauw Kie Fa in 1998. It was first used in Snes9x, which was a popular emulator for the Super Nintendo Entertainment System (SNES) at the time. The goal of this algorithm was to provide provide 2 $\times$ scaling while preserving the smooth areas and the brightness of the original source. The filter matrix was designed to make use of lines and edges of the same color. Of the four pixels produced for each input pixel, one pixel will be identical to the source, and the other three are approximated from the patterns that match the filter matrix [21].

Further work on this algorithm has led to a modified version of the filter. The modified filter is often referred to by the names Super $2\times$ SaI and Super Eagle. These algorithms work in a manner similar to $2\times$ SaI, but perform more blending of the pixel colors to produce a smoother appearance in the output.

The hqx family of filters was developed by Maxim Stepin in 2001. Here, “hq” stands for “high quality,” and “x” stands for magnification. There are three sets of filters: hq2x, hq3x, and hq4x, which provide $2\times$, $3\times$, and $4\times$ scaling, respectively [13, 22, 23]. These filters remain very popular, and have been implemented in several emulators. The hqx filters look at the eight neighbors of the source pixel and attempt to detect shapes by looking for pixels of similar color that match predefined patterns. The patterns are then used to perform a lookup in a table which provides the upscaled, interpolated patterns. The hqx family of filters work very well on images that have geometric patterns, and can produce smooth, high-quality results.

2.1.3 Vectorization Filters

The current state-of-the-art in scaling pixel art is the Kopf-Lischinski algorithm, which converts pixel art into vector graphics.

Definition 2.1.4 *Vector Graphic*

A *vector graphic* is a type of digital graphic that is based on primitive mathematical objects, such as lines, polygons, circles, and Bézier curves. Unlike pixel-based raster images, vector graphics can be moved, scaled, or rotated without any loss of quality, as no interpolation is needed.

Earlier algorithms for producing vector graphics from raster images were not well-suited for pixel art, as small details in the original image were often lost, as can be seen in Figure 3(h–i). In pixel art, these small details (often a single pixel of a color distinct from its surroundings) are very important to the meaning of the image [25].

Figure 3: Comparison of how several $4\times$ scaling filters perform on color pixel art with many straight edges. (a) shows the original image [24]. (b) through (g) show pixel domain scaling filters. (h) through (j) show vector filters.

The Kopf-Lischinski algorithm was designed specifically for pixel art. It uses a small set of weighted heuristics to eliminate edge crossings in a similarity graph, where each vertex represents a pixel and contains edges to the vertices that represent its eight neighboring pixels. Edges between vertices with dissimilar colors are removed. A heuristic algorithm to remove edge crossings is applied, resulting in a planar similarity graph. The edge information is used to create a Voronoi diagram which loosely represents the reshaped pixels. Connected sequences of visible edges are converted into quadratic B-spline curves, which are subjected to an optimization process to remove block-shaped regions [25]. This can be used to produce vector graphic output.

This filter produces very good results for certain types of inputs. In particular, input images that approximate curves appear to vectorize nicely with this algorithm

(see Figure 4), but sharp can edges suffer from excessive rounding. An example of how this algorithm performs on an image with sharp edges can be seen in Figure 3(j). The algorithm also does not work well for anti-aliased images, which are commonly found in pixel art from 16-bit generation consoles [25].

Figure 4: Comparison of several of several types of pixel art scalers on an image with smooth contours. The original 16×16 image [26] is scaled $4\times$ using nearest-neighbor interpolation, hq4x, and Kopf-Lischinski vectorization.

A disadvantage of the Kopf-Lischinski algorithm is that it is computationally expensive and currently unsuitable for real-time use during game play. There exists a video of a game developed for the Super Nintendo Entertainment System being played while this filter is converting the output to vector graphics. This video is not real-time, and was prerecorded and time-corrected for playback [25]. One issue demonstrated by the video is that it processes the entire game screen to generate the vectors, as opposed to individual pieces of pixel art as discussed in the paper. Better results could be obtained by processing the sprite layer and the several layers of parallax scrolling independently. To do this would require the modification of an emulator to extract the different layers prior to screen composition. If the emulator could be modified in this way, it would open up a number of opportunities to make other changes that can lead greater game enhancement capabilities.

2.1.4 The Limits of Interpolation

There is a fundamental problem with all of these filters in that they provide a form of interpolation. That is, they are attempting to reconstruct information that is not present in the original data. While algorithms such as Kopf-Lischinski provide very good results on many data sets, it is not possible for these algorithms to reliably introduce meaningful new detail into the upscaled images.

It is for this reason that the system presented in this thesis will not attempt to interpolate a game's visual output, but will instead replace it entirely. The goal is to detect what type of object is being observed and provide a suitable higher-resolution image for that object. The new replacement image can be constructed to have much more true detail than what could have been obtained from an interpolated resampling of the original data. To enable this functionality, the system will need algorithms or methods for detecting and recognizing different types of objects within a game.

2.2 Object Detection

To enable the system to segment the screen output into distinct units for processing, the system must have some way of analyzing the state of the game's visuals. One way to do this is to perform object detection on the video output of the emulator. A number of approaches have been developed in the fields of image processing and computer vision for tackling this problem.

2.2.1 Detection

Definition 2.2.1 *Object Detection*

Object detection refers to the detection of one or more instances of an object within an image. It is a form of binary classification in that the object at the region of interest either *is* an instance of the object, or it *is not* an instance of the object. Detection also provides localization of the object.

Definition 2.2.2 *Multi-Class Object Detection*

If more than one class of objects need to be detected and distinguished from one another, then it becomes a problem of *multi-class object detection*.

Object detection, as a single-class problem, is commonly encountered in the form of face detection. In this problem, an area of the image being processed is labeled as either containing a face, or not containing a face. It is not concerned with knowing whose face it is (which is facial *recognition*, as opposed to *detection*), just that it appears to be face-like in appearance. If there were two classes of objects being detected (e.g., faces and stop signs), then this would be an example of a multi-class object detection problem.

In general, the system requires a solution for the multi-class object detection problem. In limited circumstances, it can be a single-class detection problem, but this will only occur in situations where only a specific object needs to be detected and modified. An example of a situation meeting this condition is when a single graphical object in a game needs to be replaced for licensing or other reasons.

Early attempts at object detection systems used features that were computationally expensive, such as raw pixel data or edge detection output. For example, edge detection using a Canny filter first requires applying a Gaussian filter to minimize noise in the image, and then applies up to four directional gradient filters, depending on the types of edges that must be detected [27].

A pedestrian detection system (1997) and a general purpose object detection system (1998) based on Haar wavelets were developed by Papageorgiou, Oren, and Poggio [28, 29]. An example of a Haar wavelet is shown in Figure 5(a). A Haar wavelet transform works by producing a matrix such that the first $n/2$ rows produce a weighted average of the input, and the last $n/2$ rows produce a weighted difference of the input. These are low-pass and high-pass filters, respectively. Haar wavelet transformations can be used for compressing information about the image into less data. Papageorgiou et al.

found that pixel and edge-based representations of images were generally inadequate for tracking the objects of interest.

Figure 5: (a) The Haar wavelet at various transformations. (b) Haar-like features. White areas represent positive, black areas represent negative.

Definition 2.2.3 *Haar-like Feature*

A *Haar-like feature* is computationally inexpensive image feature that is based on looking at adjacent rectangular regions and comparing the sums of the pixel intensities of each region [7].

Most state-of-the-art frontal face detection systems are based on the Viola-Jones object detection framework [7]. The Viola-Jones algorithm achieves comparable accuracy with previous algorithms, but is much more computationally efficient. The detection is based around simple rectangular features (called Haar-like features) that capture areas of horizontal or vertical contrast in grayscale images. Examples of Haar-like features can be seen in Figure 5b. These features are computed as the difference of pixel intensity sums over rectangular regions. An example of how these Haar-like features are used by the Viola-

Jones algorithm is shown in Figure 6. By utilizing a matrix of summed areas (which Viola and Jones call the “integral image”), evaluation of these features occurs in constant time per-feature, regardless of scale or translation. The integral image I_+ at a point (x, y) can be computed from an image I by:

$$I_+(x, y) = \sum_{i \leq x, j \leq y} I(i, j). \quad (1)$$

The integral image can be efficiently computed in a single pass by making use of the following pair of recurrences:

$$s(x, y) = s(x, y - 1) + I(x, y) \quad (2)$$

$$I_+(x, y) = I_+(x - 1, y) + s(x, y). \quad (3)$$

Definition 2.2.4 *Classifier*

A *classifier* is a form of supervised machine learning that attempts to label (classify) an object as belonging to one of a set of classes.

The Viola-Jones algorithm uses a cascaded set of classifiers to perform detection. These classifiers rely on supervised learning, which involves training the classifier on a training data set. The expectation is that good performance on the training data set will lead to a model that generalizes well on instances outside of the training set. AdaBoost is used during training for feature selection. Boosting systems, such as AdaBoost, were developed for building accurate classifiers by combining the results of several low-accuracy classifiers [30]. The idea is that the cascade allows extremely fast rejection of regions which do not contain face candidates by starting off with the weak classifiers. Areas of interest (those rectangular regions that were labeled as potentially face-like by the weak classifiers) are subjected to progressively more expensive to compute strong classifiers.

Figure 6: Example of how two Haar-like features are used to measure the difference between characteristically light and dark regions in a facial image of the training set using the Viola-Jones object detection framework [7]. Each feature is computed over multiple scales and translations along the image to identify matching regions. Those regions are then subjected to the next classifier in the cascade.

There are several issues with using the existing object detection frameworks in this system. The first issue is the need to train the classifiers. To make the system as simple as possible for the developers that are using it, it should not require any more technical tasks than are absolutely necessary. Correctly training the classifiers requires some amount of experience in the field. Poor choices in the training sets, or overtraining the classifier (resulting in a model that fits the noise of the training set) can lead to unintentionally poor detection results. AdaBoost is known to be susceptible to overfitting in the presence of noise, but there has been some research into minimizing its effects on classification accuracy [31].

Another issue with object detection algorithms is the amount of computation required. While the Viola-Jones algorithm represents a substantial improvement in high-speed object detection, it is still a form of single-class object detection that uses multiple classifiers over several different scales. The system will be required to detect many objects, typically one object for each background tile and sprite that might appear on-screen, in

each frame of video output. Further, many of the objects to that require identification have a known position, and merely require labeling. Therefore, the localization of these objects is not a required step.

A final issue with these algorithms is that of accuracy with respect to locality of the detected object. The algorithms can detect occurrences of the features within some radius around the actual object, which can result in several false positive detections in the neighborhood nearby the true location. False positives are a problem for the system, as object detection leads directly to visual output to the player, and the player relies on this information to make decisions. There are various solutions for reducing the number of false positives, including algorithms that look for overlapping bounding regions or assigning confidence weights to each labeling. All of these solutions add additional processing time and still provide no guarantee of the location being correct.

Object detection concerns itself with the question of “Is this the object?” Because of the nature of visual output in video games, and because the system is interested in everything on the screen, this question does not need to be asked. It *is* the object. Perhaps the better question is “What is this specific object?” Several types of algorithms have been developed to answer that specific question.

2.3 Image Hashing

Images, even small ones, require a large amount of data. It is not practical to compare images by using the pixel data directly. For example, comparing two $n \times m$ images would require as many as nm pixel comparisons. Computing an identifying label for each image, and then performing comparisons on the labels, can make data lookup considerably faster. That is the basic idea behind a hash table. For an image, a hash value can be computed and used to lookup information about that image using only the hash value.

Definition 2.3.1 Hash Function

A *hash function* is a function $h(x)$ that takes a (possibly) large set of data and maps it to a smaller set of data, commonly a set of integers.

Definition 2.3.2 Collision

Some hash functions make no guarantee that two dissimilar inputs will not produce the same hash value. When this occurs, it is known as a *collision*.

2.3.1 General Hash Functions

A number of well known hash functions have been created and used for a variety of purposes, such as cryptography and error correction. While these hash functions are frequently and effectively used in their respective domains, their use in image processing is limited. This is due to the fact that small changes in the image (even a single bit) can produce large changes in the computed hash. If the purpose of the hashing is to detect equality at the bit level, this may be desired behavior. Good image hashing should have some resilience against small changes, as it is common to require detection of variations of the same image.

2.3.2 Transformation-Invariant Image Hashing Algorithms

It is often the case that several different images represent the “same” image. This can occur when the image data undergoes some form of transformation, whether intentional or not. For example, JPEG compression uses a lossy algorithm which can produce an image that looks nearly identical to the original image at normal viewing distance, but will have a large number of different bits. Images may also be rotated, cropped, or have subtle color alterations. A hashing algorithm that is resistant to these types of changes is called a *robust* hashing algorithm. An example of such an algorithm, developed by Venkatesan, Jabubowski, Koon and Moulin, borrows a number of ideas from the field of cryptography to develop a robust image hashing algorithm that can be used as a substitute for image

watermarking. The output of the hashing algorithm is an n -bit string that can be directly compared for equality between the images [32].

The Venkatesan, Jabubowski et al. algorithm is robust against a number of deformations. Specifically, their algorithm produced identical hashes for images that had been altered by rotation of up to 2 degrees, scaling by up to 10%, cropping of 10%, random line deletion, JPEG compression, and 4×4 median filtering [32]. However, a pair of “identical” images, with one scaled to twice the original size would produce different hash strings, and therefore not be detected as the same image. A solution to this problem is to use *geometric hashing*. The goal of geometric hashing is to be able to locate an object based on common substructures in such a way that the object would be detected regardless of scale, rotation, translation, or occlusion [33].

The earliest work on geometric hashing was done by Schwartz and Sharir in 1986, using boundary-curve matching techniques [34]. Later Schwartz, Wolfson, and Lamdan developed a new technique using point sets that was invariant under affine transformations [35].

Definition 2.3.3 *Affine Transformation*

An *affine transformation* is a transformation that preserves straight lines such that relationships between points on the line remain unaltered. Examples of image operations that are affine transformations include translation, scaling, shearing, and rotation.

The set of features under consideration are points. At a minimum, two points from the set of features form a reference frame. The coordinates of the remaining points are then computed in that reference frame. This is done for both the model of the object, and the query. The chosen points form a *basis pair*. Ideally, all possible combinations of points should be used to form a set of basis pairs. This results in comparing $(m(m-1)/2) \times (n(n-1)/2)$ reference frame systems. This data is placed into a 2D hash table.

Figure 7: (a) A model of an object consisting of nine points. (b) An arrangement of points in a scene to be used for querying. (c) A match of six points has been found, with the query being rotated seventy-two degrees clockwise.

To actually perform a lookup, the features of the object are detected (Figure 7b), and are transformed with respect to a basis. Each feature is looked up in the hash table, which essentially “votes” for which model matches the object. A histogram of the voting will show a list of potential candidates that have passed a specified similarity threshold.

Let m be the number of models (of objects) in the system, and let n be the number of features per model. Let c be the number of points that form a basis. Then, if s is the number of points in an image query, the cost of computing the hash table is $O(mn^{c+1})$, and the cost of looking up a particular model in the table takes $O(Hs^{c+1})$, where H is the time it takes to access the hash table, with $O(1) \leq H \leq O(mn^{c+1})$. For a 2D object under rotation, translation, and scaling, a basis of two points is sufficient. To support affine transformations, a minimum of three points is required for the basis [33].

Geometric hashing provides a relatively efficient method for labeling an object. However, the precomputation stage is very expensive, and at runtime requires very large hash tables to be stored in memory.

2.4 Histograms

The previously discussed algorithms can be used to identify objects within an image when the object has undergone translation, rotation, or other deformations. It is also

possible to use color intensity as a feature, allowing two objects can be compared for similarity on that feature alone. A histogram is a table of frequencies, and can be used to visualize how often something occurs. In the context of image processing, histograms can be used to visualize how features of an image (typically, colors) are distributed. Histograms can be used on both color and grayscale images.

Definition 2.4.1 *Intensity Histogram*

An *intensity histogram* represents the distribution of grayscale pixel values in an image.

Definition 2.4.2 *Color Histogram*

A *color histogram* represents the distribution of colors in an image.

Histograms approximate the distribution of colors or intensities in an image by mapping ranges of values into a single label. For example, grayscale images generally have pixel intensities ranging from 0 to 255. An example histogram will split the pixel intensity into “buckets,” each of which covers a range of thirty-two values. This will provide an intensity histogram with eight buckets. Transformed representations of an object are likely to have a similar distribution of intensity, and so the histogram is a useful feature for performing a comparison of two objects.

There are several ways to construct color histograms, as there are a number of different color spaces. Histograms for red, green, and blue (the components of an image in the RGB color space) can be computed independently. By converting the image data from the RGB color space to Hue, Saturation, and Luminance (HSL), histograms can be obtained about the brightness or overall color intensity of colors in the image. An example of a color histogram for an image created with a digital camera is shown in Figure 8.

There are a number of methods that have been suggested for comparing two histograms for similarity. A commonly used test is the Kolmogorov-Smirnov test. It finds the cumulative distribution function of each of the intensity distributions, and then

(a)

(b)

(c)

(d)

(e)

Figure 8: A set of color histograms computed from a digital photograph using Adobe Photoshop CS4. These histograms use a very large number of buckets, but still represent an approximation of the color distribution in the original image as the colors were sampled to a discrete color space when the photograph was taken. Histograms have been colorized to visualize how the color intensities are distributed. (a) Original image. (b) Red histogram. (c) Green histogram. (d) Blue histogram. (e) Luminance histogram.

computes a distance metric by finding the maximum of the absolute differences between the two cumulative distribution functions [36].

Histograms are a computationally efficient way of obtaining identifying information about an image. Histograms are generally insensitive to scaling, translation, and rotation. However, they provide no spatial information about the distribution of colors or intensities in an image when applied globally. There are several solutions to this problem, which include using sets of histograms over smaller subregions of an image to identify object-specific features. Selection of these subregions typically involve training classifiers.

2.5 Color Coherence Vectors

Computation of histogram features is very fast. Comparing histograms for similarity is also fast. Unfortunately, spatial information about the distribution of pixel data is lost. There is a histogram-based method for comparing images that maintains some spatial information, known as a color coherence vector (CCV).

In a CCV, pixels in the image are divided into coherent and incoherent pixels. Coherent pixels are those pixels that can be recognized as part of a larger, connected region. Incoherent pixels do not belong to such a region. In a traditional color histogram, instances of a single color that occur in several, distinct parts of an image contribute to the same bucket. In a CCV, these different parts of an image will not be recognized as a coherent region (there are incoherent pixels between them), and so the two instances of that region of color are labeled differently. This preserves spatial information about the image [37].

Computing the connected components can be done in linear time. The size of the component determines if the pixels are coherent or not. The size is given as τ , where $\tau \approx 1\%$ of the image area. In Figure 9, a small example of the connected component labeling is shown using a low-intensity, 6×6 grayscale image with $\tau = 4$. Pixel intensities were split into buckets that held ten values. For example, the first bucket holds values from 10 to 19, inclusive. For each bucket, some pixels will be part of a connected region

Figure 9: Computing a labeling of the connected components for a color coherence vector. The grayscale intensity values are placed into three buckets that each contain a range of ten values. The 1’s bucket is made up of pixels from two disconnected regions, as is the 2’s bucket. These are given distinct labelings, and so from these three buckets, labels A – E are obtained [37].

Table 1: The number of pixels belonging to each component is counted. Components C and D each contain less than $\tau = 4$ pixels, and so those pixels are considered incoherent [37].

Label	A	B	C	D	E
Color	1	2	1	3	1
Size	12	15	3	1	5

(more than τ coherent pixels), and some will not (incoherent pixels). Let α_i and β_i represent the number of coherent and incoherent pixels for the i -th bucket, respectively [37]. Then (α_i, β_i) is a coherence pair, and the color coherence vector is:

$$\langle (\alpha_1, \beta_1), \dots, (\alpha_n, \beta_n) \rangle. \quad (4)$$

For the image in Figure 9, the component sums are shown above in Table 1. The computed CCV for this image is shown in Table 2.

Table 2: The color coherence vector for the image in Figure 9, presented in table form. For each color, α represents the number of coherent pixels, and β represents the number of incoherent pixels [37].

Color	1	2	3
α	17	15	0
β	3	0	1

$$d(I, I') = \sum_{i=1}^n |(\alpha_i - \alpha'_i)| + |(\beta_i - \beta'_i)|. \quad (5)$$

Equation 5 shows how the difference between two CCV's I and I' can be computed. It is efficient to compute and provides a value that is typically larger than that produced by taking the difference of two standard color histograms. Therefore, it serves as a better differentiator between two source images [37].

2.6 Related Work Summary

There are several algorithms that have been developed for the purpose of enhancing the visual qualities of scaled pixel art, both in the raster image and vector graphics domains. There has also been a significant amount of research performed in the area of object detection. These object detection algorithms have several drawbacks, including computational expense and inaccuracy in object locality.

The next chapter will investigate the hardware for one of the best-known and most frequently emulated video game consoles. It will examine the architecture and explore features of the sprite engine that can be used to perform object detection and enhancement.

CHAPTER 3

Hardware and Emulation

This chapter will provide an introduction to the Nintendo Entertainment System, along with an overview of its hardware implementation. It will also introduce the FCEUX software emulator.

3.1 History

The Nintendo Entertainment System (commonly referred to as the NES) was an 8-bit video game console developed by Nintendo Co., Ltd. It was initially launched in Japan in 1983, and was released in the US by Nintendo of America in 1985. The NES was officially discontinued in 1995. The NES was very popular, selling approximately sixty-two million units, and is often credited for revitalizing the video game industry in the USA following the “video game crash” of 1983. Games that were developed for this system continue to be very popular, and have resulted in the development of many sequels for later generation game consoles [38].

3.2 Description of the Hardware

The NES contained an 8-bit central processing unit (CPU), the Ricoh 2A03, which was based on the MOS Technology 6502. This CPU ran at 1.79 MHz. The console itself contained two kilobytes of general-purpose random access memory (RAM), along with an additional two kilobytes of video RAM that was used by the Picture Processing Unit (PPU) for background graphics (known as the “name tables”). A small amount of RAM was available for other graphical elements, including 256 bytes for the Object Attribute Memory (OAM) and twenty-eight bytes to store the color palettes. Like many consoles of that era, the NES used a hardware sprite engine. The OAM provided attribute information for sixty-four hardware sprites [39].

Definition 3.2.1 *Sprite*

A *sprite* is a 2D digital image that can be rendered onto a larger image. For the purposes of this thesis, sprites are small pieces of 2D pixel art that are stored and rendered separately from the background bitmap.

Game contents were stored in Read Only Memory (ROM) contained in the game cartridges. The NES was capable of addressing up to thirty-two kilobytes of program ROM. This could be expanded through the use of bank switching to allow the CPU to access additional chunks of ROM memory [39].

3.2.1 Color Palettes

The NES allowed for the use of any color selected from a palette of forty-eight colors, and six shades of gray.

Definition 3.2.2 *Palette*

A *palette* is a finite set of set of colors used in the creation a digital image.

A sample palette is shown in Figure 10. The NES palettes were based on NTSC, and so emulators use an RGB approximation of the original colors. Colors in the palette are referenced by index, and are arranged such that the first four bits control the hue (horizontal axis in Figure 10) and the second four bits control the intensity (vertical axis in Figure 10). A background color for the entire screen could be specified. The tiles and sprites each were able to select from eight palettes (four for tiles, and four for sprites), each of which contained three colors, plus one value reserved for transparency. This allowed the NES to display a maximum of twenty-five colors on the screen at any one time [39].

A rarely used featured of the original hardware was the ability to tint the palette through the use of the palette's color emphasis bits. Three bits were used for tinting to palette towards red, green, or blue hues. The bits could be used individually, or combined. For example, setting the red bit would de-emphasize the blues and greens. This feature is

Figure 10: RGB approximation of the NTSC palette used by the NES. The eight right-most color values (appears black in the image above) are reserved and not used.

not well supported in many NES emulators.

3.2.2 Display

The display resolution of the NES was officially 256×240 pixels. Due to the limitations of NTSC-based television sets of the time, the eight pixels along the top and bottom margins were often not viewable, making the effective resolution 256×224 pixels. Game screens were stored in a 2×2 arrangement, with a particular screen being mirrored depending on whether the game was configured for horizontal or vertical scrolling. Parallax scrolling behavior was not provided by the hardware, but it was possible to change the scrolling bits between scanlines to simulate two independently scrolling layers. The screens were arranged in tiles of 8×8 pixels [39].

3.2.3 Sprites

The OAM provided storage of attribute information for up to sixty-four sprites. Each sprite was allocated four bytes of memory in this table, and the information stored included the index of the sprite, x and y coordinates, bits for controlling horizontal and vertical flipping, a priority bit, and a palette index. Several of the bits were not used. A sprite could be 8×8 or 8×16 , but all sprites were required to use the same dimensions. Each sprite was limited to using only three colors, in addition to a transparent color index that allowed the background to show through. To work around this limitation, some games used a composite sprite, in which two or more sprites are rendered in the same location to

Figure 11: An example of a composite sprite for use with the NES hardware sprite engine. Due to limitations of the engine, this sprite was made of several sprites that have been overlaid on top of each other. The body and head were each made of several 8×16 sprites that shared one palette. To draw the face, two additional colors were needed (tan and white), and so the face was a separate set of two 8×16 sprites, using a different palette [40].

give the illusion of a single sprite with more than just three colors. An example of such a sprite, extracted from Capcom’s game “Mega Man,” is shown in Figure 11 [40]. The use of composite sprites was common in NES games, but at a great cost, as there could never be more than sixty-four sprites on-screen at the same time and composite sprites could quickly use up the available OAM slots. A further limitation of the hardware sprite engine was the requirement that there be no more than eight sprites rendered as part of any single scanline. The default behavior of the hardware was to simply not render any sprite pixels that occur after the eight sprite for that scanline. Some game developers attempted to detect when this occurs, and would cycle the order of the sprites in the OAM to trigger a flickering sprite, ensuring that all of the sprites are seen at least some of the time. This limitation could also be used to the advantage of game developers, as hidden sprites also contribute to this behavior. This allowed for partial masking of sprites, which was a feature not provided by the hardware itself [39].

3.2.4 Background Graphics

The NES stored background graphics in an area of memory known as the “name tables.” Each name table was a 16×15 grid of tiles (each of which was 16×16 pixels) and

Figure 12: The layout of the name tables in the NES. The portion that the player sees can be scrolled horizontally or vertically. (a) The player view is not scrolled. (b) The player view has been horizontally scrolled and is shown parts of two name tables.

used approximately one kilobyte of memory. The NES could have up to four name tables active. The NES only had two kilobytes of graphical memory to store the data, and so there were only ever two tables actually being stored, with the remaining two being copies. It was possible to adjust the player’s view of the name tables by adjusting horizontal or vertical scroll registers (see Figure 12), but not both at the same time.

The name table functions as a tile map, much as the OAM tables provide a listing of information about on-screen sprites. It provides an access point by which the system can determine which background graphics are located on the screen and at what location.

3.2.5 Audio Channels

The NES produced music and sound effects using five audio channels. There were two pulse wave channels with variable volume, a triangle wave channel with fixed volume, and a two-mode white noise channel with variable volume. The final channel was a differential pulse-code modulation channel. The sounds were generated by the CPU, and were output on two pins: one for the two pulse channels, and the other for everything else [38]. Emulators can access this information directly. There exists a number of hardware hacks that tap these two pins to allow for discrete two-channel output, providing a form of pseudo-stereo sound. In an unmodified NES, these channels were mixed together prior to the audio

output jacks, making the output effectively monaural.

3.3 Description of the Emulator

The NES emulator being used in this system is FCEUX, version 2.14a [41]. This emulator has been in development since 2008, and is based on the earlier work of Family Computer Emulator (FCE). The source code consists of approximately 168,000 lines of C++, and is supported on many compilers and platforms.

FCEUX successfully emulates the NES hardware, as well as much of the expansion hardware provided on various game cartridges. It also provides a number of features to enhance the game experience, some of which remove limitations in the original hardware.

3.3.1 Visual Enhancements

FCEUX does not allow a game to use more than the number of colors allowed by the original hardware, but does allow the palettes to be configured to custom colors. The system does not require a specific palette, but once a palette has been chosen it should not be altered. Custom video modes can be configured to exceed the original resolution of 320×240 . A number of options are provided for controlling the method of upscaling, and the emulator supports $hq2x$, $hq3x$, $Scale2\times$, and $Scale3\times$ upscaling algorithms [42].

The allowable drawing area can be configured to emulate traditional CRT televisions, which could not render the top and bottom eight pixels. Further, it allows clipping of the left and right eight pixels, which can reduce color artifacts on screen edges when palette swaps occur during scrolling [42].

The final graphical enhancement provided by FCEUX is the option of removing the limitation of having only eight sprites per scanline [42]. For some games, this can greatly improve the visual quality the game by reducing flickering, but can break the functionality of games that depend on this behavior.

3.3.2 Audio Enhancements

FCEUX allows for additional control over the sound channels. It features a built-in resampler for upsampling, individual volume controls for each channel, as well as an independent master volume for the mixed sound output [42]. While not explicitly provided as an option, little effort would be required to map each channel to a distinct location in the stereo image.

3.3.3 Debugging Facilities

FCEUX provides a substantial amount of debugging functionality built-in to the product. The background and foreground graphics layers can be independently enabled or disabled [42]. This is critically important for any object detection algorithms used in the system, as it allows for the use of more efficient algorithms which do not take occlusion into account.

The PPU Viewer tool displays the contents of the image data banks. The data on the left is generally used for sprites. In the screenshot shown in Figure 13(a), there are 8×16 sprites are loaded, and these are stored in such way that the upper 8×8 half is immediately followed on the right by the lower 8×8 half. The data on the right generally represents the images used in the background layer. In this example, it includes elements for text display. All images are rendered in three colors on a black background. This is because the PPU viewer does not have any information about which palettes are being used for the sprites or tiles at any given time.

The Name Table Viewer tool shown in Figure 13(b) displays the name table memory, which represents the current view of the background layer tile grid. This screen shows the tile map as rendered in the color palettes currently in use during game play.

(a)

(b)

Figure 13: (a) Picture Processing Unit (PPU) Viewer and (b) Name Table Viewer debugging screens. The PPU Viewer shows the two “banks” of image data, prior to palette index lookups. The Name Table Viewer shows the layout of the background layer in memory [42].

3.3.4 Random Access Memory

Games maintain all of their state information by storing the state data in Random Access Memory (RAM) during gameplay. Each game is responsible for all of the management of this data, and so there are no standard memory addresses for specific types of game state information. For example, the memory address at which a score is stored for one game might not store similar information in a different game.

FCEUX allows for debugging hooks that occur whenever a particular set of memory addresses are read or written to. If the system were to be expanded to include game-specific memory checks, FCEUX provides the necessary tools for getting access to the required information.

3.3.5 Summary

The NES hardware maintains a large amount of useful information about the different types of game assets that are currently loaded. The FCEUX emulator provides a virtual emulation of this hardware, allowing software to obtain access to this information. Additionally, the FCEUX emulator provides a number of debugging aids that will make the development and testing easier.

CHAPTER 4

Description of the System

Definition 4.1 *The System*

For the purposes of this thesis, *the system* is composed of several pieces of computer software and digital data, along with a set of prescribed algorithms for use with that data, that have been developed as part of this research to implement the game enhancement process described in Chapters 4–7.

The goal of this system is to successfully provide an enhanced user experience to the player of a video game by providing a semi-automated replacement of the sounds and graphics in the game. Elements may be enhanced by increasing the number of colors or the resolution of the game graphics, although this is not strictly necessary. The system targets emulators for hardware sprite engine-based game consoles. These types of game consoles maintained a video frame buffer for the background, and sprites were combined with the background during scanline rendering (one row of pixels at a time).

The visual output that is displayed to the game player represents the state of the game at a given moment in time. When enhancing the game’s visual output, it is important that what the game player sees remains an accurate representation of the game’s internal state. Therefore, a restriction is placed on the system such that any enhancement made to a game does not alter the game player’s ability to perceive what is happening in the game.

4.1 Software Components

Several software components are required for the the system. Executable software components include an emulator for a video game console, and a ROM image of a game to run in that emulator. Non-executable software components include various configuration files, such as mapping data, digital images, and digital audio files.

4.1.1 Emulator

The term *emulator* was coined by developers at IBM in 1963, and at that time referred to the emulation of one piece of hardware on another by implementing microcode on the new machine that performed the tasks of the old machine. This is in contrast to *simulation*, which referred to software-based emulation [43].

Definition 4.1.1 *Emulator*

An *emulator* is a piece of computer software that simulates most or all of the functionality of another computer system. This enables the computer running the emulator to execute programs that could only be run on the computer system being emulated.

An emulator for the game console is a required part of the system. The emulator software can be written specifically for this system, or it may be part of an existing emulation package. During the initial development phase, the source code of the emulator must be available for modification. The specific modifications that must be made will vary depending on the emulation target. After the modifications have been completed, only the binaries of the emulator (along with any supporting files) are required for the system.

Excluding ongoing maintenance, the development of the emulator is a fixed-cost in terms of the system. Development of the features required by the system (or the emulator itself, if not modifying an existing codebase) is a one-time event, and does not need to be repeated for each game to be enhanced.

4.1.2 Game Binary Image

Definition 4.1.2 *ROM Image*

The executable code and data for a console game were typically stored in one or more Read Only Memory (ROM) chips inside the game cartridge. A digital file that contains a copy of the data in the ROM chips is known as a *ROM image*.

The system requires a ROM image of a game. This will be loaded and executed by the

emulator. The purpose of the ROM image is to provide all of the game logic, and to be used as a tool for performing analysis on the game's visual output.

4.1.3 Configuration Data

The system must have a way to configure how the new assets are loaded and recognized. This information can be stored in one or more data files. The files can then be loaded by the modified emulator at runtime.

Several pieces of information should be user-configurable. At a minimum, a list of all replacement graphics, along with a unique label that maps some piece of state information from the original game to the replacement graphic, is required.

There are several common file formats used for specifying configuration data for an application. Examples include INI (common in older applications running on Microsoft Windows) and Extensible Markup Language (XML). A custom file format may be used, and is not necessarily restricted to being a text-only file. The only restriction is that the modified emulator must support reading and writing data in the specified format.

4.1.4 Art Assets

One of the stated goals of the system is to visually alter visual output of the game. Alteration, for the purposes of this system can be defined as providing a substitute image for one or more images in the original game's display. The substitutes can be higher-quality upscalings, modified images, or complete redrawings of the game's content.

Artwork must be provided in the form of a digital file that can be loaded and rendered by the modified emulator. The image format chosen should support 32-bit RGBA images, including support for alpha channel transparency. There are several image formats that meet this requirement. Among the most commonly used are Portable Network Graphic (PNG) and row bitmap images. Currently, the system only considers raster-based images, but there are no technological reasons why a vector graphics format could not be used if

the appropriate rendering code is implemented in the modified emulator.

The number of image files required depends on how much of the original game imagery is marked for replacement in the configuration data. While the system can be used for total replacement of the game's visual output, it is also conceivable that only single tiles or sprites may be replaced. The digital image files are, in general, unique to each game ROM being enhanced.

4.1.5 Audio Assets

The system does not require specific audio assets to be provided. If an emulator supports sound output, it may play the original sounds specified in the ROM image.

The system should be able to use an alternative set of sounds for the game, including sound effects and background music. Any suitable digital audio format may be used. If implemented, the software libraries used for sound playback should support a minimum of two simultaneous streams of audio to allow for the concurrent playback of background music and least one sound effect.

4.1.6 Redistributable Package

Once the system has been deployed, there must be a mechanism for loading the configuration data and modified art and sound assets into the emulator. The system does not propose any specific requirements for how this may be performed.

4.2 Development Roles

Each component of the above system requires some amount of human interaction. A number of distinct roles have been identified in this system, and they can be categorized by the amount of ongoing effort required by that role in support of the system. The number of people required by the system is, at an absolute minimum, one person. This occurs in the event that one person is performing all of these tasks by themselves. A more likely scenario is that there is at least one person or more per role.

4.2.1 Zero-Effort Roles

The role of *original developer* is defined as the set of people who were responsible for developing the game in its originally published state, including all design, writing, programming, and creation of game assets. The contribution of the original developer is the game ROM image. It is assumed that the game ROM image, being a binary copy of an already existing game, requires no further development effort in the context of this system.

4.2.2 Fixed-Effort Roles

The system requires the use of an emulator software package that has been modified to support the functionality required by the system. The role of *emulator developer* is defined as the set of software developers who are responsible for writing the programming code for the emulator, or making the required changes in the source code of an existing emulator. Upon successful completion of the modified emulator, the emulator developer's contribution to the system ends.

4.2.3 Variable-Effort Roles

The role of *upgrader* is defined as a set of one or more people who will use the software provided by the emulator developer in conjunction with a game created by the original developer to produce an enhanced game. It is the responsibility of the upgrader to analyze the game, determine what is required for enhancement, and produce artifacts to satisfy those requirements. It is possible for the original developer and the upgrader to be the same entity, but they need not be.

The role of *Asset developer* is responsible for taking the art and sound requirements from the upgrader and producing the digital image and sound files for use in the enhanced game. The upgraders may be the same as the asset developers, but need not be. The asset development role is one that requires a variable amount of effort, as each game may require a different amount of asset creation and the assets are (generally) unique for each game.

The final role is that of the player. The player is defined as the end user of the enhanced game. The player is not required to perform any function in the development process, but may be used to obtain feedback about the enhanced game. This feedback can be used to make further refinements to the game prior to final distribution.

4.2.4 Workflow

When a decision to enhance a game is made, several steps must take place. First the upgrader must acquire a copy of the game ROM from the current copyright holder. This may or may not be the original developer. The game ROM is loaded into the modified emulator, and the upgrader uses the the emulator to create the configuration data and to document what assets need to be created. This information is passed to the asset developers, who produce the digital art and sound files for use in the enhanced game. These files are provided to the upgrader for testing in the system. If changes are required, the asset developers are informed, and this process repeats until asset development is complete. The upgrader can then package the game for redistribution. A diagram of the expected workflow is shown in Figure 14.

Figure 14: Workflow diagram showing the components and roles in the system.

CHAPTER 5

Object Detection Implementation

This chapter will discuss the implementation of the object detection portion of the system. Object detection in the system consists of detecting sprites and background tile graphics.

5.1 Modifying the Emulator for Game Enhancement

The emulator must be modified to provide support for object detection, event detection, and asset replacement. In the process of doing so, there will be opportunities to access some of the emulator's knowledge about the state of the virtual hardware. This section of the thesis will talk about the process of determining what pieces of information might be useful, the implementation of new emulator features to extract and use this information, and a discussion of the difficulties encountered.

5.1.1 Initial Analysis of the Emulator

The source code base of FCEUX is very large, consisting of approximately 168,000 lines of C++ code. Building the source code was not an easy task, as the the source code distribution shipped with an incorrect version of the luaperks.lib file. Fixing the linking errors that resulted from this issue required finding a correct version of the library file. This was eventually found as part of another open-source project, and after replacing the originally provide file, the project successfully compiled and linked.

The largest obstacle that was encountered was the lack of documentation for the source code. There is no formal development reference, and documentation inside the source code itself was sparse. There are approximately 23,000 lines of commenting in the source code. Unfortunately, a large amount of this consists of licensing information at the top of each file. Meaningful comments are missing in sections of code where their presence

would be critical to the understanding of what it does. This is especially important in the code for functions with vague names, such as `Fixit1` and `Fixit2`. As an example, the source code for `Fixit1` is shown in Figure 15. Constants are defined, but are generally ignored, making it difficult to understand what is happening when checking or setting the many hardware flags. It is unclear what this function is actually trying to fix. There are many similar examples scattered throughout the source code for this emulator.

```
static void Fixit1(void)
{
    if(ScreenON || SpriteON)
    {
        uint32 rad=RefreshAddr;

        if((rad&0x7000)==0x7000)
        {
            rad^=0x7000;
            if((rad&0x3E0)==0x3A0)
            {
                rad^=0x3A0;
                rad^=0x800;
            }
            else
            {
                if((rad&0x3E0)==0x3e0)
                    rad^=0x3e0;
                else rad+=0x20;
            }
        }
        else
            rad+=0x1000;
        RefreshAddr=rad;
        //PPU_hook(RefreshAddr); //,-1);
    }
}
```

Figure 15: Source code snippet of `Fixit1`, a function inside `ppu.cpp`. The developer’s original formatting has been preserved. The function does not describe what it fixes, nor is it obvious from the code itself. The FCEUX source code has a large amount of code written in a similar manner.

5.1.2 Information Available to the System

The emulator has access to all of the information about the state of the virtual hardware, including the contents of CPU registers, RAM, video RAM, video ROM, and game ROM. The fact that it is accessible does not necessarily mean that it is easily obtainable. Further, there must be some discussion about how much of this information is actually beneficial to the system.

The image processing component will require access to the video output buffer or a similar representation of the visual data that will be sent to the screen. The contents of the Object Attribute Memory (OAM) contain valuable information about the location and identity of the sprites that are displayed on-screen. The name tables contain similar information about the background layer. The state of the sound output can be obtained by intercepting the input to the Audio Processing Unit (APU) for each of the five sound channels.

The previously mentioned pieces of data can be processed in a blind manner, in that the same techniques can apply for any game ROM being used in the emulator. System RAM can be tapped to determine the value stored at a specific location in memory, but the contents are entirely dependent upon the specific coding of the game. For example, many games keep track of a numeric score for the player. The NES hardware does not provide a register for this information, and there is no standardized memory location or format. One game may store the score data at a specific memory address, but another game may use that same address to store completely unrelated information.

5.1.3 User Configurable Behavior

The system must expose an interface for configuring how game assets in a game ROM are mapped to new replacement assets. The modified emulator must support reading and writing from a data source that contains this information, such as a file or network connection.

At a minimum, the system requires the ability to map a specific identifier to a replacement game asset. The format of the identifier may be unique to each type of asset. For example, the current implementation of the system uses an unsigned integer to represent the identifier for both sprites and tiles. Sprite identifiers are chosen from a set of numbers from 0 to 255, which identify their index values in the OAM table. Tile identifiers may be any number between 0 and $2^{32} - 1$, but may be split into smaller, easier to manage subcomponents as the 32-bit data represents four 8-bit numbers packed into a single word.

```

; Sample System Configuration File
[SPRITES]
0 = 'C:\GameData\char.png'
1 = 'C:\GameData\enemy.png'
2 = 'C:\GameData\obstacle.png'

[TILES]
(0,0,128,128) = 'C:\GameData\tile1.png'
(32,255,255,64) = 'C:\GameData\tile2.png'

[SCENES]
Cave = 'C:\GameData\InTheCave.scenedata'

```

Figure 16: A sample of configuration information stored in the Microsoft Windows INI format.

Configuration data may be stored in any format that is convenient for the emulator developer and upgrader. INI files are common on the Microsoft Windows platform, but developers may also choose to use Extensible Markup Language (XML) or a binary storage format. An example of a partial configuration file in Windows INI format is shown in Figure 16.

5.2 Sprite Detection

Detection of multiple moving objects is a difficult problem due to the need to separate the foreground from the background, and because multiple objects may be occluding each other. A number of temporal algorithms have been developed to eliminate the foreground objects from a set of image frames [44]. This assumes that there are enough frames to achieve satisfactory results, and that the foreground objects are actually moving. Occlusion can make object labeling and localization difficult by obscuring object features that are important for detection. By utilizing information contained within the OAM, we can obtain a complete listing of all in-use sprites and information about their location and orientation without having to perform any background subtraction pre-processing and without having to use an occlusion-invariant detection algorithm.

Table 3: Memory layout of an entry in the Object Attribute Memory [39]. One entry exists for each sprite that is on-screen during a frame.

OAM Entry		
OAM Byte	Bit Index	Value
0	N/A	y -coordinate of the top side of the sprite.
1	0	Tile bank
	1–7	Tile index
2	0–1	Palette index
	2–4	—Unimplemented—
	5	Priority: 0 $\leftarrow$ in front. 1 $\leftarrow$ behind.
	6	Horizontal flip
	7	Vertical flip
3	N/A	x -coordinate of the left side of the sprite.

The layout of an OAM entry is described in Table 3. Object localization can be obtained by examining the zeroth and third bytes to obtain the x and y coordinates of the sprite.

Definition 5.2.1 *Unordered Map*

An *unordered map* is a template-based associative container in C++ (`unordered_map`) that provides an iterable hash table implementation [45].

Object classification can be obtained by examining the first byte. Specifically, bits 1–7 can be viewed as a labeling of the sprite and used for lookup in an unordered map. A second unordered map can be maintained if the background name table is being used for foreground rendering. In this case, the zeroth bit can be used as an index to select the correct unordered map.

The system maintains storage for up to sixty-four identified sprites per frame, mirroring what is happening in the hardware. This list is ordered identically to the OAM

list, as the ordering is important when rendering. For a given scanline, the OAM entries are rendered starting from the end of the list. The list can then be traversed in-order, retrieving the required data from the unordered map at rendering time.

The second byte of each OAM entry requires some additional explanation. The bits for the palette are an index into the palette table. Sprites are only allowed to use palettes 4–7. The priority bit is used to control if the sprite is rendered behind the background layer, or in front of the background layer. The three unimplemented bits were actually non-existent in the hardware, and so OAM entries were only twenty-nine bits wide. Of the data stored in the second OAM byte, the current implementation of the system makes use of the bits for horizontal and vertical flipping to handle on-screen objects that use a single half-sprite that is horizontally or vertically flipped to produce the full image. A more complete implementation could use the palette information to select a from a different unordered map, allowing games that use the same sprites with different palettes to preserve this difference in upgraded versions.

5.3 Background Detection

For sprites, the system directly accesses information in the OAM to identify and localize individual sprite objects. Similar information about the background graphics layer is available, but this implementation will use an image processing approach to detect background tile graphics in order to allow for the comparison of the two implementations.

5.3.1 Video Output Buffer

All image processing and computer vision systems require one or more frames of graphical data to analyze. FCEUX provides several opportunities to access information that could be used as an input for the image processing component. To get the most accurate representation of the data that is being sent to the screen, the system accesses the image data after all scanline rendering to the memory bitmap has been completed, but

Table 4: Required configuration values for FCEUX 2.14a. The choices for these values are restricted to ensure that the current implementation of the system works as expected.

Option	Restricted Value	Reason
Full Screen	Unchecked	Screen resolutions may vary
Force Integral Scaling Factors	Checked	Prevent aspect ration changes
Video Mode	Custom	Force specific values
Resolution	320×240	Fixed size
Bits Per Pixel (BBP)	8	Force color depth
NTSC First Line	8	NTSC restriction
NTSC Last Line	231	NTSC restriction
Graphics:OBJ	Disabled	Disable sprite rendering
PPU	“Old PPU”	Uses the older PPU code
Sound Enabled	Unchecked	Disable APU output

prior to being sent to the graphics card. This image data is accessible in `vidblit.cpp`, in the `Blit8ToHigh` function.

The emulator has several code branches responsible for performing this task, as the specific function calls and data structures vary depending on user-configurable options, such as applying video filters for upscaling. The system is not interested in processing the output of the scaling filters, and so those branches can be skipped. The emulator can also be configured to generate output at a certain color depth [42]. We restrict the color depth to 8-bit color output to ensure that the expected code path is always followed. A complete listing of the configuration restrictions for this implementation are shown in Table 4.

The system now has a single point of access to the image data for each video frame. The data is in the form of a device independent bitmap (DIB), with the image stored upside-down. It is not uncommon for DIB images to be stored in this manner on platforms

that maintain backwards compatibility with IBM OS/2's Presentation Manager [46]. This image data will be passed to the image processing component for storage once per frame, and is vertically flipped at that time. Additionally, a flipped and $4\times$ nearest-neighbor upscaled version of the image is rendered to the video output surface using the Win32 `StretchDIBits` function. This will serve as a background layer upon which all other rendering is performed. It also serves as a visual guide for the developer during the upgrading process.

5.3.2 Preparation

The original data is a stream of bytes that represent a palette-indexed image. The first step is to perform the palette index lookups and use this information to convert the data to a 32-bit RGBA raster bitmap image. The dimensions of this image are $256 \text{ pixels} \times 224 \text{ pixels}$. The bytes of the image are obtained inside the output stage of the emulator after all scan lines have been rasterized, but prior to actually displaying the image on-screen.

The contents of the image represent the background layer of the game's visual output as seen by the Picture Processing Unit (PPU). As the contents of the sprite layer can be efficiently detected through the emulator's hardware state information, eliminating the sprite layer from the rendering leaves a clean tile map image. This removes the constraint that the tile detection algorithm be invariant to occlusion.

5.3.3 Tile Recognition

Many games divide the screen content into two distinct regions. One region contains the actual game play field. This region is composed of a series of square graphics known as tiles. The second region is reserved for scores and player feedback information. This region will be referred to as the banner region. The banner region is primarily static, with only a small amount of content changing in response to some event in the game. It most commonly appears along the top or bottom of the game play region (see Figure 17). While

Figure 17: The game screen is common divided into two regions: a game play region and a player feedback region. In (a), the player feedback region appears below the play region [47]. In (b), it appears above the play region [48].

the game play region requires tile-by-tile detection, the banner region may be specified manually through the configuration file if needed.

In the tile-by-tile detection phase, each tile is assigned a labeling. Tile localization is not required, as the location is known from its position on the tile grid. The labeling is based on a feature composed of spatial color information. Color can be an important cue in object detection, especially when other features are unreliable or non-existent in the image [49]. For the NES, up to four colors may be used per-tile, and information about the spatial distribution of these colors can be used to differentiate between two or more tiles. Color coherence vectors (CCV) were considered as a way of labeling each tile. A CCV is a way of computing a color histogram of an image that treats color regions that are spatially separate (incoherent) as distinct “colors.” CCV’s generally consider a region of pixels to be coherent if the size of the region exceeds approximately 1% of the size of the image [37]. The system is working with tiles of size 16×16 , and so a coherent region could consist of a portion of the image that is as small as three pixels. Dithered images, which may alternate two colors in a 1×1 checker-board pattern, are not uncommon in pixel art. It appears that the use of CCV’s may not be the best option for use with these types of images as the

dithering patterns will result in large quantities of incoherent regions. As an alternative, this system will use a labeling based on the average colors of the four quadrants of each tile.

Each 16×16 tile is divided into four 8×8 quadrants, Q_1, Q_2, Q_3 , and Q_4 , each of which contain sixty-four pixels. Each quadrant is processed to find the average pixel color A_{RGB} for that quadrant, with $0 \leq R, G, B \leq 255$:

$$A_{\text{RGB}} = \frac{1}{64} \left(\sum_{x=1}^8 \sum_{y=1}^8 Q(x, y) \right). \quad (6)$$

Each color channel has an intensity range of 0–255, requiring eight bits (one byte) for storage. A_{RGB} requires a minimum of three bytes to store the red, green, and blue components, and thus requires twelve bytes to store the information for all four quadrants. This is not a large amount of memory, but it can be potentially inconvenient to the upgrader to require them to input 3-tuples for every tile. Further, it can be difficult to visualize this information in a usable form on the tile map display.

To resolve this problem, the data that represents the tile image will be converted to grayscale. The algorithm for this must not lose any of the spatial information about the color distribution, and it should avoid producing duplicate grayscale values for different colors.

Let A_R, A_G , and A_B be the red, green and blue components of A_{RGB} respectively. Then the grayscale value A_G is:

$$A_G = \mathcal{G}(A_{\text{RGB}}) = 0.299A_R + 0.587A_G + 0.114A_B \quad (7)$$

where $\mathcal{G}$ is a grayscaleing function. The function $\mathcal{G}$ shown here is a perceptual grayscaleing formula known as the *Luminance* formula [50].

It is common for color images to be converted to grayscale for object detection tasks. A number of grayscaleing formulas were considered. Kanan and Cottrell found that perceptual

Figure 18: A panel of fully saturated colors, including individual RGB channels, combinations of two channels, and a gradient is processed using several different grayscaling formulas [50]. (a) The original RGB panel (b) Gleam (c) Intensity (d) Luma (e) Luminance (f) Lightness (g) Value (h) Luster (i) Decolorize.

grayscaling formulas were not as effective as as linear formulas for general object recognition. However, one of the exceptions that they found was that perceptual formulas had an advantage in recognition of textures [50]. The data contained in each tile image represents a low-resolution texture. Further, the palettes in use are of very limited color depth, and so it would not be uncommon to have equal values in the color subcomponents. Perceptual grayscale models use a weighted combination of the color subcomponents, minimizing the likelihood of two colors generating the same grayscale value. Figure 18 shows the results of several grayscaling formulas applied to a set of fully saturated RGB colors. The Luminance formula (Equation 7) provides a good balance of color perception (see Figure 18e) and low computational cost, and so the Luminance formula was chosen for function G .

A_G can be stored using only eight bits, as $0 \leq A_G \leq 255$. This means that the average

Figure 19: Two 16×16 tiles [48] are shown with different types of label assignments. (a) and (f) are the original tiles. (b) and (g) use per-quadrant color averaging. (c) and (h) use per-quadrant grayscale averaging. (d) and (i) use full-tile color averaging. (e) and (j) use full-tile grayscale averaging. Note that the quadrant based averaging produces tiles that, while similar, have clear differences due to the use of spatial information. Full-tile averaging produces indistinguishable results for both tiles.

grayscale value for each of the four quadrants can be packed into a single 32-bit unsigned integer (see Figure 20). This integer contains information about the color distribution of the tile and preserves some information about the localization of those pixels within the tile. In Figure 19, two 16×16 tiles with a similar color distribution are shown. The tile in Figure 19f has most of its red component in the upper right corner, whereas the tile in Figure 19a has the red component more or less equally distributed across all four corners. Performing color averaging per-quadrant and converting to grayscale preserves this spatial color information and using full-tile averaging does not.

By reducing the amount of information used by the feature to four small numbers, the upgrader can more easily work with the information. Within the emulator's display, all four numbers can easily fit within a tile square on-screen, allowing for easy visualization of the labeling data. The numbers also allow for easier data entry into the configuration files. This is implementation specific, and a more thoroughly-developed, user-friendly development tool could hide much of the complexity from the upgrader, allowing for more data per tile if needed.

Figure 20: The grayscale average of each quadrant is a value between 0 and 255. It can therefore be stored in eight bits. This means that the grayscale averages for all four quadrants can be packed into one 32-bit word.

The dimensions of the color image I that the system is processing are 256×224 pixels. The NES uses a tile size of 16×16 pixels, and thus the background layer is a 14×16 grid of tiles. Let $T \in \mathbb{N}_{\leq 2^{32}-1}^{14 \times 16}$ be the matrix used to store the computed tile identifiers, with $T_{i,j}$ being the tile detection result of the tile with its upper-left corner at position (i, j) in I . Then $T_{i,j}$ can be computed as :

$$T_{i,j} = \begin{bmatrix} A_G(A_{\text{RGB}}(Q_1)) \\ A_G(A_{\text{RGB}}(Q_2)) \\ A_G(A_{\text{RGB}}(Q_3)) \\ A_G(A_{\text{RGB}}(Q_4)) \end{bmatrix} \quad (8)$$

where Q_1, Q_2, Q_3 and Q_4 are the four quadrants of the tile with its upper-left corner at position (i, j) in I .

Computation of $T_{i,j}$ relies only on arithmetic operations and uses no branching other than the highly-optimized branching in the generated loop code. This allows for very fast computation of the features, and thus efficiently labels the tile.

The computed 32-bit word (Figure 20) is used as a hash key in an `unordered_map`. The key represents the image content of the tile, and can be used to obtain a handle to a replacement image. During each frame of game play, all tiles on the screen are processed to compute their hash key. The system then queries the table to obtain an image handle. If

a bitmap handle is found, the system can render the replacement image onto the display. If the key cannot be found in the table, nothing will be rendered onto the screen, resulting in the original image showing through. This serves as a visual guide during development, but can also be useful if the developer only wants to replace a small number of the graphical assets. If no changes in the tile matrix have been detected from the previous frame, the most-recently rendered bitmap is reused.

Prior to performing the hash key lookup, the upgrader may want to perform additional processing on the tile detection matrix. For this, the system provides post-detection filtering.

5.4 Post-detection Filtering

The tile detection algorithm attempts to determine what is on the screen. By itself, the algorithm only provides enough information for a single candidate replacement tile. This is because the tile labeling is used as a key for looking up an image in a hash table. A second algorithm is used for post-filtering the tile detection matrix.

Definition 5.4.1 *Post-Detection Filter*

The *post-detection filter* is an algorithm that is applied after tile labeling has occurred. For each tile, it examines the labelings of adjacent tiles to determine if the current tile's label should be altered.

Several reasons have been identified for requiring post-filtering of the tile detection matrix. The detection algorithm produces an integer labeling, and the labeling may not be unique. An example of two tiles that produce the same labeling is given in Figure 21. A combination of the tile labeling, along with contextual information from neighboring tiles can be used to correctly locate an acceptable candidate label for replacing the tile.

The detection algorithm will generate an identical tile labeling for identical tiles. While this is the desired behavior for determining what is currently on-screen, the upgrader may

Figure 21: Two tiles that have been artificially constructed to produce identical tile labelings. The tile in (a) produces the same average pixel color for each quadrant as tile (b).

want to provide alternative graphics depending on contextual clues from neighboring tiles. As an example, large areas filled with a single tile may utilize several different replacement graphics to create a less repetitive or more context-sensitive appearance. This feature has no dependence on the the specific method by which T was obtained, and can also be applied if T was populated from virtual hardware state information. An example of how post-detection filtering may be applied is shown in Figure 23. Correction of a tile labeling resulting from non-unique labels is a subset of this problem, and both can be treated identically.

The tile detection matrix T stores the initial labelings for each tile. The post-detection matrix T_P maintains a second set of labelings that are initially undefined. Let i and j be the row and column indices of the tile currently being processed in the post-detection filter. Then the filter can access $T(i', j')$ and $T_P(i', j')$, with $i' \in \{i - 1, i, i + 1\}$ and $j' \in \{j - 1, j, j + 1\}$. Figure 22 illustrates the set of neighbor tiles that can be evaluated as part of the filter.

If post detection filtering is requested, a filter is initialized with data read from the user configuration data. The filter is applied to all elements of T . For each element, the

Figure 22: The post-detection filter grid. The current tile is located at position (i, j) . This filter is swept across every position in T checking for matches at (i, j) and one or more of the neighbor elements. If a match is found, the value at (i, j) will be replaced.

filter determines if the current element should be modified. If it should be modified, the replacement label is stored at the same position in T_P . At rendering time, a modified version of T , referred to here as T' , is used to provide the renderer with a set of labels to use for image replacement. T' is T , with each element of T that has a non-empty equivalent in T_P replaced with the labeling in T_P .

Figure 23: Example of how post-detection filtering can be used to differentiate two tiles which, while identical in the source image, should have different representations in the final output. In (a), there are identical 16×16 tiles [48] stacked on top of each other. These tiles represent a wall. In (b), the two tiles have been provided with different labels as a result of post-detection filtering.

5.5 Scene Detection

The 14×16 tile detection matrix T can be viewed as a 28×32 grayscale image, as each element in T can be broken into four independent components (two rows and two columns). Low-frequency energy is preserved during the production of T through the averaging process, as the labelings of each element in T is effectively a form of low-pass filtering.

It has been shown color cues are a primary scene recognition cue and that colored scenes could be identified by humans even at very low resolutions [51]. Further, it has been suggested that 32×32 is the minimum resolution for a color image on which scene recognition can be reliably performed [52]. For grayscale images, the required image dimensions grow to approximately 64×64 [52]. These dimensions were recommended in the context of querying a large database of *natural* images (i.e., naturally occurring images, captured by photography or other means). Color information has been discarded in T . The image represented by T cannot be considered a natural image as it is algorithmically derived from an artificial image which may or may not represent a naturally occurring scene.

The image represented by T is used as a query to determine if T matches a scene of interest. Unlike many content-based image retrieval systems, the goal is not to obtain a list of all images known to be similar to the query. The goal is to find a single image that is similar (beyond a specified scoring threshold) to the image represented by T . If one does not exist, then T does not represent a scene.

The image represented by T lacks noise, due to both the nature of the source of the data as well as the results of the low pass filtering. There is also no need to perform any form of object detection or segmentation on the image. Therefore, the 28×32 grayscale image should be sufficient for this purpose.

CHAPTER 6

Scene Recognition and Event Detection

Thus far, the components of the system that have been described in earlier chapters are capable of detecting the locations and orientation of sprites, and of detecting the presence of specific tiles on the game screen. This provides enough information to the system to enable one-to-one replacement of game images. Through the use of post-detection filtering, individual tiles can be mapped to one of several context-sensitive replacement images. While this is sufficient to provide an enhanced game experience, it is possible to further enhance the output by providing the system with a higher-level awareness of what is happening within the game. In this chapter, the methods used by the system for detecting larger-scale events are discussed.

6.1 Events

To perform event processing, the system must have a list of situations to detect, along with a set of responses to those situations. These two elements shall be called triggers and events, respectively.

Definition 6.1.1 *Trigger*

A trigger is a user-defined detectable situation that the system may take notice of in order to produce some change in the system state.

Definition 6.1.2 *Event*

An event is an action performed in response to a trigger.

To provide an example of how triggers and events work, consider a scenario where the system has detected that the player has entered a specific room in a game. This serves as a trigger. In response, the system may request that the background music change. The request to change the background music is an example of an event.

6.1.1 Triggers

Triggers can be broken into several sub-categories. An appearance trigger recognizes when an object with a specific labeling has been newly detected over a series of video frames. The objects of interest may include background objects located on the tile map, or foreground sprite objects. Objects that are on-screen may also disappear, and so the the absence of an object that was previously known to be on-screen can be considered as a disappearance trigger.

A movement trigger produces an event when objects are found to be moving according to some user-defined pattern, or when they are found to be nearby a specific location. Examples of movement triggers include the detection of specific objects changing direction or speed, as well as the detection of a collision or adjacency of two or more objects.

Whereas the previously described triggers are relatively simple to detect, the system should also be able to process events of greater complexity. This may include boolean combinations of simple triggers, or it may extend to full-scale scene processing. The current version of the system used in the modified emulator provides implementations for appearance and disappearance triggers.

6.2 Scenes

A game can often be divided into several distinct parts. Common examples of this include title screens, configuration screens, and changes in the game play environment. Specific map locations or combinations of on-screen elements may also be of interest. In the context of object detection, a *scene* often refers to a complex image that contains more than one object [53]. In this system, the definition is narrowed such that a scene represents a particular set of objects in a specific, fixed spatial relationship.

Definition 6.2.1 *Scene*

A scene is a complex trigger, defined by a set of state information that, when taken together, uniquely identify a specific moment in the game.

In a typical game, there exists many scenes that can be detected. Of these, relatively few have much meaning to the player. A procedure for marking interesting scenes is provided in the modified emulator.

6.2.1 Marking Procedure

The upgrader plays the game in the modified emulator and presses a specific key sequence (in this implementation, it is the TAB key) when presented with a moment in the game that should be marked as a scene. The upgrader must then enter an identifying tag for the scene, which will be stored on-disk along with a vector representation of the tile detection matrix for the current screen. An example of the user interface for this process is shown in Figure 24.

In the mapping data, the identifying tag entered during marking is entered as an event trigger. This process is repeated for each scene that the upgrader wants to identify with an event. For each scene, multiple events can be generated.

6.2.2 Representation and Recognition

The dimensions of the game screen are 256×224 pixels, divided into a grid of tiles that are each 16×16 pixels. This results in a grid of tiles that is 14×16 tiles in size. Let Σ be the set of all scenes that have been marked for recognition. To perform scene recognition, we again let $T \in \mathbb{N}_{\leq 2^{32}-1}^{14 \times 16}$ be the tile detection matrix for the current frame, and we let $S \in \mathbb{N}_{\leq 2^{32}-1}^{14 \times 16}$ with $S \in \Sigma$ be the data matrix associated with a scene to be recognized.

S and T can be viewed as small grayscale images. One method that is commonly used to test for image similarity is to compute the Euclidean distance between the two images:

$$d(T, S) = \sqrt{\sum_{i=1}^m \sum_{j=1}^n (T(i, j) - S(i, j))^2}. \quad (9)$$

(a)

(b)

Figure 24: In this set of screenshots, (a) the upgrader has identified a part of the game that represents a scene. (b) A key is pressed, and the upgrader is presented with a file dialog to save the scene tag information to disk. In this case, the the vector $\vec{t}$, representing the scene displayed in (a), is associated with the tag “InTheCave.” Each $\vec{t}_i$ is a packed 32-bit word built from four 8-bit numbers, each of which represents the intensity of the average color in that quadrant.

An alternative distance measure is to compute the Manhattan distance between the two images:

$$d(T, S) = \sum_{i=1}^m \sum_{j=1}^n |T(i, j) - S(i, j)|. \quad (10)$$

There is a problem with both of these distance metrics when used with the data in T and S . A large distance between any two elements of T and S does not indicate that it is “more different” than would a smaller distance. Any difference at all indicates that the tiles were labeled differently. Thus, a distance of nine, computed using either distance metric, can be misleading. Was it one tile that had a value that differed by nine? Or was it nine tiles that each differed by one? There is no way to obtain this information directly through these measurements.

Color coherence vectors (CCV) were again considered for this task. Color information was stripped during the tile labeling phase, leaving a set of grayscale values in T . Treating the grayscale values as “colors,” the image represented by T can be broken up into sets of coherent and incoherent regions. This preserves some spatial information, but not enough to uniquely identify a particular scene.

A simple method for identifying if the current scene T is known in Σ is through testing for matrix element equality. For every $S \in \Sigma$, compare S with T . If S is strictly equal to T , then T has been recognized as a scene.

This method does not allow for any differences between T and S (i.e., the query scene and the scene must be identical). If T is supposed to represent a scene, but has had a minor alteration (for example, a moved block or a text overlay), then T will no longer be considered a recognizable match for a scene. Another simple solution is to use matrix subtraction. Let $Z \in \mathbb{N}_{\leq 2^{32}-1}^{14 \times 16}$ be a matrix and $Z = |S - T|$. The number of non-zero elements in Z represent a crude form of similarity scoring, being equal to the number of

tiles that were not exact matches. This method has the advantage of being computationally fast and is also easy to implement. This distance measure is defined as:

$$d(T, S) = \sum_{i=1}^m \sum_{j=1}^n \phi [T(i, j) - S(i, j)], \quad (11)$$

where $\phi(x)$ is the function:

$$\phi(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise.} \end{cases} \quad (12)$$

When a scene is saved to disk, the contents of the file are the matrix S , written as a set of vectors $\vec{s}_i$. When the system loads the list of scenes at startup, this data is read back to form the set of scenes Σ .

During gameplay, the tile detection matrix T is created during every frame. T is used as a query to determine if there is any $S \in \Sigma$ that matches T within a certain dissimilarity tolerance. There are 224 elements in every S , as the matrices are 14×16 . Therefore, a mismatch in one element (i.e., a single tile was labeled differently in T and S) means that there is a $\approx 0.446\%$ difference between the two scenes.

Tolerances may be specified per-scene in the configuration data. This allows the upgrader to specify that a scene S can only be matched by a query that results in a similarity score that exceeds some threshold. For example, if the scene and the query differ by three tiles (at a penalty of 0.446 per tile), then the threshold must be set less than 98.662% to register as match.

6.3 Non-interfering Reaction

Through event detection and scene recognition, the system can have awareness to complex situations within a game. However, the system must be restricted in the ways that it can respond to these events. The reason for this is that the game's logic, as stored in the

ROM, cannot be altered. For example, an event in response to a trigger could not open up a hidden passage through a wall where one would not have already existed.

Responses to an event can be either visual or audible. Visual responses can include replacement or alteration of individual graphics, or could extend to replacing the entire background layer with one or more pre-rendered bitmap images.

Audible event responses may include the playing of digital audio files when a specific enemy or item appears on-screen. Scene detection may be used to generate an audible response in the form of changing the background music when a specific scene has been recognized.

6.3.1 Sound Events

The sound output of the NES was rather primitive. There were no “instruments” to be heard, but they were instead simulated by a pair of square wave oscillators, a triangle wave oscillator, one static noise generator, and one pulse code modulation (PCM) channel. Pitch and duration were specified by the game coding for each channel. This information can be accessed from within the emulator [39].

The Musical Instrument Digital Interface (MIDI) was developed in the early 1980s as a way for electronic musical instruments to communicate with each other. It supports a number of different types of event data, including pitch and duration [54].

It is possible to silence the output of the emulator’s APU code (see Table 4 on page 52), and instead translate the input data for each channel into a set of MIDI events. The MIDI event data can then be sent to any MIDI-compatible device, including external hardware or software-based synthesizers. This allows for enhancement of the sound data by directly utilizing the original audio coding information as a data source for the MIDI synthesizers.

For this implementation of the system, a decision has been made to bypass this system entirely, and utilize the scene recognition engine for loading and playing external digital music files. This enables the system to use a greater number of polyphonic lines, and greater

flexibility in sound choices.

To enable the system to load and play several simultaneous digital audio files, the BASS Audio Library has been added as an externally loaded library in the modified emulator. One audio stream handle (an HSTREAM) is provided for each simultaneous channel of audio. The sound being played on a channel can be initialized or altered by calling the `BASS_StreamCreateFile` function and passing in the file path to the digital audio file. The stream handle returned can then be passed to the `BASS_ChannelPlay` function to begin playback. Playback can be controlled in several ways, including specifying that a sound file should loop when it has reached the end of the file. Channel playback is stopped by using the `BASS_ChannelStop` function and passing in the corresponding stream handle.

The current implementation of the system maintains audio stream handles for background music, one non-looping sound effect, and one looping sound effect. The handles are updated based upon the triggered events, including scene recognition.

CHAPTER 7

Discussion of the Results

I have developed an implementation of the system by modifying the source code of the FCEUX emulator to support the required features. To test the system, a commercial game, *The Legend of Zelda*, developed by Nintendo, was chosen for the game ROM image [48].

7.1 System Feasibility

The system was developed and tested using a pre-existing emulation software package (FCEUX) for the emulator portion of the system. This emulator is capable of successfully emulating the original game, with all in-game behaviors unchanged. A set of upgraded graphics and sound assets were used for replacing the original in-game data.

7.1.1 Modifying an Emulator

The FCEUX emulator software used in this system has been in development since 1994. The developers of FCEUX had no awareness of this system, and so the requirements of the system had no impact on the technical design of emulator. Despite this, I was able to modify the source code for FCEUX to support the requirements of the system without needing to make any substantial architectural changes. Therefore, it is clear that an emulator does not need to be specifically designed in a particular way to support this system. An off-the-shelf emulator package can be successfully modified to use this system, as I have done in this implementation.

The difficulty of making those modifications depends greatly on the understandability of the source code for the emulator, as well as on the quantity and accuracy of the documentation. In this particular implementation, the source code for FCEUX was difficult to understand and lacking sufficient documentation. This is not an uncommon scenario in many of the open-source emulators that I have examined. This did not prevent me from under-

standing enough of the code to make the changes that were required. If an emulator were to be newly developed, support for the functionality required by this system would require very little additional effort to build into the design.

This research has shown that using a modified game console emulator to provide enhanced versions of sprite-based games is an approach that shows promise, and that enhanced games can be developed at a substantially lower cost than developing a traditional port of the game.

7.1.2 Specific Features Implemented

In the current build of this system, I have implemented a subset the possible functionality. Sprite detection is implemented by reading the virtual hardware state information from the emulator's OAM table. Background graphics are detected through the tile detection algorithm. Several examples of post-detection filtering are implemented. A scaling factor of $4\times$ has been chosen for this implementation.

The event detection implementation currently processes appearance and disappearance triggers, in addition to scene recognition triggers. There are nine defined scenes, four of which produce a complete replacement of the background layer with a pre-rendered bitmap image.

The system provides a digital audio soundtrack that uses several scene-dependent MP3 audio files. Sound effects are also provided in the form of MP3 audio files.

7.2 Object Detection

The system provides two approaches for processing on-screen game elements. This implementation of the system implements sprite detection by extracting data from the OAM tables within the emulator. Tile detection is obtained by obtaining the video buffer and applying the tile detection algorithm. Both approaches provide successful detection.

7.2.1 Sprite Detection Results

The NES game console supported up to sixty-four sprites on-screen per frame. The sixty-four sprites were stored in the OAM, and each had an index into the video memory of the PPU. Image data for up to 256 sprites could be loaded at any one time. Each sprite required sixteen bytes of storage. Some game cartridges had additional hardware storage in the form of expansion chips. This storage could be used to store additional sprite image data that could be loaded into the PPU video memory on-demand. The largest known commercial NES game had 512 kilobytes of ROM reserved for sprite data. The game used in testing the system uses two banks of 256 sprites drawn from the game ROM. The two sprite banks have several sprites in common.

From the viewpoint of the game player, the 8×16 sprites generally represent “half-objects,” being the left and right halves of a 16×16 object composed of two sprites (see Figure 11 on page 34). In my original attempt at replacing the original sprite images with 64×64 images, I relied on detecting specific halves of a complete sprite being on-screen, and inferring the contents of the other half. For example, by identifying the left half of a composite sprite, the a replacement for the complete composite sprite could be rendered by a single precomposed image. This approach was successful, but required specifying additional data for some sprites. This was due to specific sprites that were used as left or right halves with horizontal flipping, but matched with a different sprite depending on orientation. The additional data was in the form of appearance triggers, which were used to set flags within the system. I resolved this problem by using 32×64 replacement images, with one replacement image for each piece of 8×16 image data present in the video ROM. I then modified the source code of the sprite replacement rendering to respect the horizontal and vertical flip bits for each OAM entry. Therefore, sprite-based objects that were composed of multiple sprites in the original game remain that way in the new system.

Table 5: Comparison of hardware sprite engines available in popular video game consoles. This details the total number of hardware sprites available, how many sprites can be rendered on a single scanline, the number of colors per-sprite, and possible sprite sizes.

Device	# Sprites	(per scanline)	Colors	Size(s)
NES	64	8	3	8×8 , 8×16
SNES	128	34	15	8×8 , 16×16 , 32×32 , 64×64
Game Boy	40	10	3*	8×8 , 8×16
Game Boy Advance	128	128	256	8×8 , 16×16 , 32×32 , 64×64
Sega Master System	64	8	15	8×8 , 8×16
Mega Drive	80	20	15	8×8 , 16×16 , 32×32 , 64×64
Neo Geo	381	96	15	$1 \times 2 - 16 \times 512$

* Grayscale.

The system detects the 8×16 sprites used in the selected game with 100% accuracy. Here, accurate detection means that the system has correctly identified what sprite is on-screen, and has correct localization information for that sprite. Localization includes identifying the x and y screen coordinates, horizontal or vertical orientation, and z -ordering.

The sprite detection system has a very efficient implementation. By utilizing the OAM information from the emulator, detection per-sprite occurs in constant time, as it only requires looking up a bitmap handle in a hash table. Detection of all on-screen sprites is approximately linear with the number of sprites displayed.

7.2.2 Applicability to Other Emulators

All emulators for the NES will have the same virtual hardware state information available. Sprite detection, as described above, can be implemented in those emulators without changing the general logic. Table 5 provides information about the hardware sprite

engines in a number of popular video game systems. The Game Boy Advance (GBA) and the Neo Geo hardware sprite systems supported additional features not present in older game consoles. For example, the Neo Geo could shrink sprites by scaling them, and the GBA supported arbitrary scaling and rotation of sprites.

Detection of these sprites by traditional object detection algorithms would require the use of an algorithm that is invariant to scale and rotation, such as geometric hashing. Geometric hashing is computationally expensive, but easily parallelizable. The devices that may implement this system might not provide multiple cores or GPU processing, and may be limited in the amount of system memory.

For emulators of game consoles that have scaling and rotation capabilities, using the state information inside of an emulator to locate and label sprites will provide a substantial improvement in performance and accuracy. Specifically, the system will maintain $O(1)$ running time for per-sprite detection, and thus $O(n)$ for all n sprites, even if there are 128 sprites with arbitrary sizes, rotation, and occlusion.

7.2.3 Tile Detection Results

Tiles are pieces of image data that are 16×16 pixels in size. A set of 224 tiles in a 14×16 arrangement make up the background layer of the game's visual output. The tile grid can be scrolled in either the x or y dimension, but not both simultaneously. The arrangement of tiles on-screen is stored in an area of the PPU memory known as the name tables. A total of 256 distinct 8×8 subtiles can be used at any one time. Limitations of the NES PPU mean that these subtiles only occur in groups of four, and so there are a total of sixty-four possible tiles that can be used at any one time. The size of the image dataset used for these tiles could be arbitrarily extended through expansion hardware in the game cartridge. For the game used in testing the system, there are two sets of tile images, with forty-eight shared tiles between the two sets.

In the current implementation of this system using FCEUX, I have chosen a relatively

simple algorithm that assigns an integer labeling for each tile. The integer is a composite value composed of four 8-bit numbers, each ranging in value from 0 to 255. These numbers represent a grayscale intensity level for a quadrant of the tile. A color average is computed for each quadrant, and this average color is converted to grayscale using the Luminosity grayscale formula. Dividing the tile into quadrants maintains some spatial information about the color distribution in the tile, and using a perceptual grayscale formula preserves some information about the difference in individual color channels.

Accurate tile detection is considered as finding a unique labeling for each graphical tile seen by the system such that the labeling can be used as a hash key. As shown in Figure 21 on Page 60, the labeling algorithm used in the current implementation of the system does not produce a unique value for all inputs. These examples were artificially constructed to exploit this weakness, and I do not anticipate such collisions happening in real-world inputs. For the specific game being tested, tile detection was 100% accurate when not scrolling. Scrolling across name tables (in particular, vertical scrolling) produced graphical glitches as the algorithm was not able to detect the partial tiles reliably. This is due to the tile no longer aligning to the tile detection grid, thus providing different labelings than what is expected. At some computational expense, a sliding grid can be used to align the grid with the scrolled image data for better results. Localization was not a consideration in tile detection as it was inferred from the tile's position within the tile grid.

7.2.4 Weaknesses of the Image Processing Approach

The NES hardware supported only one tile-based background layer, with limited scrolling functionality (illustrated in Figure 25a). A listing of hardware-supported background modes for commonly emulated game systems is given in Table 6. Later game consoles had hardware support for multiple background layers, which were not necessarily tile-based. Additionally, these layers could be scrolled (Figure 25b) at independent rates. On some systems, layers could be rotated, or scaled independently (Figure 25c). The intro-

duction of these features poses a problem for the system’s current tile detection algorithm, as the algorithm expects an image of one background layer as an input and uses a computed hash for each tile.

Figure 25: Diagram showing several different orientations for background layers. In (a), a sprites are layered on top of a single background layer. In (b), multiple background layers, each with independent alpha transparency and scrolling are present. In (c), some of the background layers have transformations applied (rotated and scaled).

Of the game consoles that I examined, the SNES presents the most challenges to the current implementation of the system. The SNES had several background modes. Each mode had distinct features. One background mode of the SNES, Mode 7, is especially problematic, as it allows the use of an affine transformation matrix to provide a sense of perspective. Mode 7 allowed for scaling and rotation to be changed for each scanline by changing the values in the transformation matrix during H-blank. Developers were particularly creative with this mode, often using it in unexpected ways. Figure 26 shows two examples where Mode 7 was used in ways that would present problems for the present image-processing based approach to tile detection. Figure 26a presents a screenshot from a game in which the background (in Mode 7) was used to represent an enemy. This enemy was much larger than typical sprite-based enemies, and could be freely transformed through rotation and scaling. Screen-elements that appear to be the “background,” such as the gray bricks, were actually sprites rendered over the background. Figure 26b shows an

example of Mode 7 being used to present a pseudo-3D perspective. In this example, Two different views of the same background layer are presented in a split screen by changing transformations per-scanline.

Figure 26: Two SNES games using Mode 7 in an unexpected way. In (a), the large enemy in the air is actually a Mode 7 background, with support for scaling, translation, and rotation. Everything else on the screen is rendered as a hardware sprite, including the castle platform [26]. In (b), Mode 7 is used for both halves of the screen, with different transformation matrices applied per scanline [55].

Some games for the SNES made use of additional on-cartridge hardware. Examples of this hardware include the SuperFX and SuperFX2 chips. This hardware enabled the SNES to play games that used a limited number of 3D polygons, and provided additional 2D scaling and stretching effects. Figure 27 shows an example of a game that combines Mode 7 with hardware sprite rendering, along with 3D polygons overlaid onto the screen bitmap.

Detecting the background layers of these later game consoles can be computationally expensive, due to the requirement for transformation invariance in the detection algorithms. For this implementation of the system, image processing was shown to be a successful technique for identifying background elements. It is clear that this approach could work if implemented in an emulator for any of the other game consoles in Table 6, but at a significant increase in both development costs and computation per-frame. The

Figure 27: The SNES game *Star Fox* used the SuperFX chip to provide primitive 3D polygon rendering in addition to hardware sprite rendering on Mode 7 background graphics. Games that use co-processors to perform graphics work would require additional modifications to the emulator to extract the relevant data [56].

modified emulator for any of these systems can be further altered to allow each layer to be independently accessed as either a indexed tile map, or as a bitmap image. Layer-specific information, such as scroll values, ordering, scaling, and rotation can be read directly from the emulated hardware registers. For SNES games that use an on-cartridge co-processor, such as the SuperFX, there may need to be additional changes made to the emulator source code. Emulators generally provide an implementation for these co-processors, and so it should be possible to extract information from them, much as the current implementation of the system does for sprites in the OAM. For relatively simple hardware, such as the NES, image processing is slower than extracting the state information, but still achieves acceptable performance and accuracy in the most common scenarios. For more complex hardware, this research clearly demonstrates that an image processing approach has significant disadvantages when compared to using virtual hardware state information.

Table 6: Comparison background layer bitmap support in popular video game consoles. This details the number of background layers, limitations per layer, and comments about additional features or limitations, such as whether the hardware supports scaling and rotation of those layers.

Device		Layers	Colors	Size(s)	Comments
NES		1	13	256×240	$(256 \times 224 \text{ usable})$
SNES	Mode 0	4	4	256×224	Each layer uses its own palette of 4 colors.
	Mode 1	3	16, 4	256×224	Two layers with 16 colors. One layer with 4 colors.
	Mode 2	2	16	256×224	Each tile can be individually scrolled.
	Mode 3	2	256, 16	256×224	One layer has 256 colors. The other has 16 colors.
	Mode 4	2	256, 4	256×224	Each tile can be individually scrolled. One layer has 256 colors. The other has 4 colors.
	Mode 5	2	16, 4	512×224	One layer has 16 colors, and the other layer has 4 colors. 512-width resolution, interlaced.
	Mode 6	1	16	512×224	As Mode 5, with support for scrolling individual tiles.
	Mode 7	1	256	1024×1024	Rotation and scaling through matrix transformations.
Game Boy		1	4	160×144	Grayscale.
GBA		4	256	240×160	2 layers support scaling and rotation.
Sega Master System		1	16	256×192	—
Neo Geo		0	4096	320×224	The background graphics were made up of sprites in a tile arrangement.

7.3 Event Detection Results

The current implementation of the system detects appearance and disappearance triggers. The system maintains a boolean flag for each object that the upgrader configures as a trigger. The object can be a sprite or a tile. In each frame, the detection of any object that is in the trigger list sets the corresponding flag to *true*. The absence of an object that was previously known to be on-screen causes its flag to be reset.

Detection of these event triggers occurs after all recognition has been performed, regardless of whether the object detection results were created by using the emulated state information or through image processing. Therefore, the specific method used does not impact the accuracy of the event detection, assuming that objects were detected with 100% accuracy.

There were no errors in the detection of either appearance or disappearance triggers. The difficulty comes in the form of determining what labeling values to use for triggers. Some graphical elements that may be suitable for use as event triggers sometimes occur in unexpected ways, as developers would often reuse parts of these graphics for other visual elements. This will vary from one game to the next, and so it requires some trial and error from the upgrader to ensure that good triggers are chosen.

An image-processing implementation of this feature in the emulators for later-generation game consoles may be problematic due to the reduced guarantees of accuracy when labeling objects. This form of error would result in the unexpected presence or absence of an event. Using the emulator state information eliminates this possibility and ensures consistent event detection results.

7.4 Scene Recognition Results

Scene recognition has been implemented by way of querying the current scene against a list of known scenes. As with the event detection system, scene recognition occurs after all object detection has been performed, regardless of the method used. Likewise, the choice

of method will have no effect on the performance of the scene recognition system assuming that the query scene is an accurate representation of what is actually displayed on-screen. This guarantee cannot be reliably made if a image processing approach to object detection is used, particularly on game consoles that can use a variety of transformations on the background graphics layer. Using the emulator’s hardware state information provides the guarantee that is required to have this feature work reliably each and every time.

Fortunately, the hardware of the NES is simple enough to allow the system to successfully use an image processing approach. Several scenes were marked using the user interface. Similarity thresholds were specified in the configuration data for each scene, and the system correctly detected the scene events when presented with matching scenes during gameplay.

Figures 28–35 show several examples of how the scene recognition has been used to perform enhancement during different parts of the game *The Legend of Zelda* [48]. This implementation uses scene recognition to replace background graphics, add graphical overlays, replace background music, and add new music and sound effects where none were previously used.

Figure 28: (a) Original title screen. (b) Scene recognition is used to perform a complete replacement of the entire viewable display area. This example also plays a modern version of the game’s theme song.

(a)

(b)

Figure 29: (a) Original character selection screen. (b) Scene recognition performs fullscreen replacement of the background layer graphics. Sprites are rendered over the overlay. The original game did not feature music on this screen, but this example uses scene recognition to add a musical background.

(a)

(b)

Figure 30: (a) Original first playable game screen. This is the first time in which the player has control of the protagonist character. (b) Background layer graphics have been rendered using the image processing system for tile detection. The scene recognition engine has been used to start the game music and provide subtle graphical overlays above the sprite layer.

(a)

(b)

Figure 31: (a) Original cave scene. (b) This screen makes extensive use of the scene recognition system. Upon scene detection, the system adds music and a looping fire sound effect (the original game had neither of these), replaces all background graphics with a pre-rendered background, and plays a one-time sound effect that is a man speaking the words that appear in the original game as text. In addition, after the system has rendered the sprite layer, scene recognition is used to provide a fog overlay and a world overlay, allowing a new sense of depth to the cave.

(a)

(b)

Figure 32: In this screen, scene recognition has been used to provide a world overlay (the tops of the trees) allowing the player character to walk behind the tree tops. The scene recognition system also detects the presence of the rock throwing enemies and plays appropriate sound effects.

Figure 33: (a) Original world map screen. (b) The scene recognition system detects the presence of water, rocks, and fireballs and generates appropriate sound effects. It is also used to detect when the aquatic enemy pops up and plays an appropriate splash sound. The original game did not feature any of these sounds.

Figure 34: (a) Original “outside dungeon” map screen. (b) The scene recognition engine is used to provide graphical replacement of much the background layer and a graphical world overlay. Appropriate sound effects are provided in response to on-screen action. This scene has been configured to change the music to the overworld theme music if it is not already playing.

Figure 35: (a) Original dungeon map screen. (b) The scene recognition is performing full replacement of the background graphics layer as well as providing a graphical overlay and fog. It has change the music and looping sound effects in response to the change in environment.

7.5 Discussion Summary

This implementation of the system has proven to be an effective method of enhancing the audio-visual content of games that are running in an emulator for a game console that utilizes a hardware sprite engine. Further, it has shown that using the virtual hardware state information inside the emulator is an efficient and reliable means of performing object detection within the system and that this method has significant advantages over image processing algorithms that perform the same task.

CHAPTER 8

Conclusions and Future Work

8.1 Summary

For over twenty years, emulators have been used to play video games on hardware other than that for which they were originally designed to run on. Today, even mobile devices such as cell phones are equipped with far greater capabilities for audio and video playback than was what was available on the original game console hardware. The development of network-based services have provided game publishers with an avenue for republishing older game titles as digital downloads for use on an emulator. These emulators are limited in the amount of enhancement that can be applied to the game, and are generally limited to upscaling filters. Previously, game developers would have to rewrite a game for a new game console to take advantage of increases in graphics technology. This thesis has suggested an alternative approach to increase the graphics and sound quality of these games without requiring an expensive rewrite of the game code.

Prior work on emulator-based game enhancement has relied on interpolation of pixel data to produce upscaled raster images or vector graphics. The algorithms used to obtain these earlier results did not use any form of object detection and processed the original video image as an atomic unit. Decomposing the image into several distinctly labeled parts can aid enhancement by allowing specific portions of the image to be replaced with substitute images. Finding a meaningful subdivision of the original image requires object detection. Multi-class object detection becomes computationally inefficient when the needs for invariance against several transformations are introduced.

In this research, the system takes a novel approach to the problem by relying on an emulator that has been specifically modified to allow access to portions of the virtual hardware's state information. Access to the state information provides all of the benefits of object detection (decomposing the original image into several independent units) without

any of the the disadvantages (high computational cost and no guarantee of 100% accuracy in either the labeling or localization). The results of this research show that this system is superior to previous methods that depended on processing full video frames.

8.2 Suggestions for Future Work

Now that a working implementation of this system has been developed, there are a number of avenues for further development. This includes additional work on the modified FCEUX emulator implementation, as well as implementing the system in emulators for other game consoles.

8.2.1 Further Development on FCEUX

This implementation was tested using a single commercial game. There were approximately eight hundred games released for the NES. While many of these games do not do anything out of the ordinary, several used various hacks and undocumented features of the NES hardware. Many games also used additional hardware inside the game cartridge itself.

NES emulation is rather mature at this point, and most of the tricks used by these games are well known. Nearly two decades after the release of the last official title for the NES, some emulators still have difficulty emulating every corner case. Therefore, it would not be surprising if some of these games did not work well with the system as currently designed. It may be worthwhile to test the system with these games to determine what, if any, changes need to be made to accommodate these behaviors.

The current implementation uses the virtual state information for the sprite layer and an image processing approach for the background layer. This decision was only made for the purposes of comparing the two methods, and this research has shown that replacing the background layer code with new code that uses the virtual state information will provide better performance and accuracy.

8.2.2 Other Hardware Sprite Engine Emulators

The NES is the earliest of the hardware sprite-based game consoles that still enjoys widespread popularity. Later game consoles, such as the SNES, Game Boy, and Game Boy Advanced all use hardware sprite engines, and so much of the design of the system can be applied to modify emulators for those platforms. Like the NES implementation, image processing should be abandoned in favor of using the virtual hardware state information. This change is especially important for these game consoles as they are the first generation of game consoles that support multiple, simultaneous background layers and increased numbers of sprites, all with varying amounts of transformation applied. This is a case where the existing object detection algorithms have been specifically identified as being potential performance bottlenecks.

8.2.3 Semi-Automated Image Retrieval

The system currently relies on the upgrader and the asset developer to work together to produce suitable digital image files for use by each game. This requires an effort proportional to the number of unique files that must be created for each game. However, many games feature graphics that look different on-screen, but represent similar real-world objects.

It may be possible to produce a digital art library of generic tiles and sprite replacement graphics for use with an image retrieval system. In such a system, graphics extracted from the game are used as a query into the image retrieval system to find a list of candidate replacement images. As an example, many games have levels that feature various square blocks for the player to walk on. These blocks may look like low resolution versions of bricks, stones, or other materials. A ‘red brick block’ search would retrieve a list of potential matches in the image library for images that may be suitable to replace those tiles.

The large square image set assembled by Torralba, Fergus, and Freeman [52] would make a good starting point. There are approximately eighty million square images in the

database. Each image is labeled with exactly one English noun. During enhancement, the database can be queried for by a noun for each tile. Results can then be filtered by image similarity. The final choice can then be made by the upgrader or asset developer.

8.2.4 Non-Raster Image Replacement

Prior to this research, the last major development in enhancing emulated game graphics was the Kopf-Lischinski algorithm for converting pixel art to vector graphics. One of the goals of this research was to overcome some of the limitations encountered by Kopf-Lischinski, such as high computational complexity and an inability to separate foreground and background objects.

This system is free from those limitations. At the present time, the system only considers raster bitmap images for replacing on-screen game elements. This decision was made primarily for convenience, as it was considerably easier to work with graphics in this format. With the system in place, it is now possible to take a further step to perform Kopf-Lischinski vectorization at design-time using isolated graphical elements, allowing for the use of vector graphics at runtime without any of the penalties that would be imposed by previous implementations.

LIST OF REFERENCES

- [1] R. Newman and B. Sheffield, “10th Annual Salary Survey,” *Game Developer*, vol. 18, no. 4, pp. 8–13, 2011.
- [2] J. Andersen. “Where Games Go To Sleep: The Game Preservation Crisis, Part 1.” Jan. 2011. [Online]. Available: http://www.gamasutra.com/view/feature/6271/where/_games/_go/_to/_sleep/_the/_game/_php/ [Last Accessed: March 18, 2013]
- [3] M. Guttenbrunner, C. Becker, and A. Rauber, “Keeping the Game Alive: Evaluating Strategies for the Preservation of Console Video Games,” *International Journal on Digital Curation*, vol. 5, no. 1, 2010. [Online]. Available: http://publik.tuwien.ac.at/files/PubDat_190893.pdf
- [4] Nintendo of America, Inc. “Virtual Console.” Oct. 2011. [Online]. Available: <http://www.nintendo.com/wii/online/virtualconsole/> [Last Accessed: March 18, 2013]
- [5] Sony Computer Entertainment America, LLC. “PlayStation Network.” Oct. 2011. [Online]. Available: <http://us.playstation.com/psn/> [Last Accessed: March 18, 2013]
- [6] C. C. Bracken and P. Skalski, “Telepresence and Video Games: The Impact of Image Quality,” *PsychNology Journal*, vol. 7, no. 1, pp. 101–112, 2009. [Online]. Available: [http://www.psychnology.org/File/PNJ7\(1\)/PSYCHNOLOGY_JOURNAL_7_1_BRACKEN.pdf](http://www.psychnology.org/File/PNJ7(1)/PSYCHNOLOGY_JOURNAL_7_1_BRACKEN.pdf)
- [7] P. Viola and M. Jones, “Robust Real-Time Face Detection,” *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2002.
- [8] L.-J. Li and L. Fei-Fei, “What, Where and Who? Telling the Story of an Image by Activity Classification, Scene Recognition and Object Categorization.” in *Computer Vision: Detection, Recognition and Reconstruction*, ser. Studies in Computational Intelligence. Springer, 2010, vol. 285, pp. 157–171. [Online]. Available: <http://dblp.uni-trier.de/db/series/sci/sci285.html#Fei-FeiL10>
- [9] L.-J. Li, S. Hao, L. Yongwhan, and L. Fei-Fei, “Objects as Attributes for Scene Classification,” in *Proceedings of the 2010 European Conference on Computer Vision, International Workshop on Parts and Attributes*, Crete, Greece, Sept. 2010.
- [10] D. Zhang and S.-F. Chang, “Event Detection in Baseball Video Using Superimposed Caption Recognition,” in *Proceedings of the tenth ACM International Conference on Multimedia*, ser. MULTIMEDIA ’02. New York, NY, USA: ACM, 2002, pp. 315–318. [Online]. Available: <http://doi.acm.org/10.1145/641007.641073>

- [11] W. Zhou, A. Vellaikal, and C. C. J. Kuo, "Rule-based Video Classification System for Basketball Video Indexing," in *Proceedings of the 2000 ACM Workshops on Multimedia*, ser. MULTIMEDIA '00. New York, NY, USA: ACM, 2000, pp. 213–216. [Online]. Available: <http://doi.acm.org/10.1145/357744.357941>
- [12] S. Green. "Blargg's NTSC Libraries." 2011. [Online]. Available: <http://slack.net/~ant/libs/ntsc.html> [Last Accessed: March 18, 2013]
- [13] M. Stepin. HiEnd3D. "hq3x Magnification Filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq3x.html> [Last Accessed: March 18, 2013]
- [14] N. A. Dodgson, "Image Resampling," Wolfson College and Computer Laboratory, University of Cambridge, Tech. Rep. TR261, 1992. [Online]. Available: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-261.pdf>
- [15] A. Goldberg and R. Flegal, "ACM President's Letter: Pixel Art," *Communications of the ACM*, vol. 25, no. 12, pp. 861–862, 1982. [Online]. Available: <http://dl.acm.org/citation.cfm?id=358728.358731>
- [16] R. Shoup, "SuperPaint: An Early Frame Buffer Graphics System," *Annals of the History of Computing, IEEE*, vol. 23, no. 2, pp. 32–37, 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=929909
- [17] K. Jack, *Video Demystified*, 5th ed. Elsevier Science, 2007, pp. 247–249.
- [18] A. Mazzoleni. "Scale2x and EPX." 2011. [Online]. Available: <http://scale2x.sourceforge.net/scale2xandepx.html> [Last Accessed: March 18, 2013]
- [19] K. Thomas. MacTech. "Fast Blit Strategies: A Mac Programmer's Guide." June 1999. [Online]. Available: <http://www.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html> [Last Accessed: March 18, 2013]
- [20] A. Mazzoleni. "Scale2x Algorithm." 2001. [Online]. Available: <http://scale2x.sourceforge.net/algorithm.html> [Last Accessed: March 18, 2013]
- [21] D. Liauw Kie Fa. "2xSaI: The Advanced 2x Scale and Interpolation Engine." 2001. [Online]. Available: <http://vdnoort.home.xs4all.nl/emulation/2xsai/> [Last Accessed: March 18, 2013]
- [22] M. Stepin. HiEnd3D. "hq2x Magnification Filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq2x.html> [Last Accessed: March 18, 2013]
- [23] M. Stepin. HiEnd3D. "hq4x Magnification Filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq4x.html> [Last Accessed: March 18, 2013]
- [24] Microsoft Corporation. "Computer Icon, Windows 3.1." 1992.

- [25] J. Kopf and D. Lischinski, "Depixelizing Pixel Art," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 99:1 – 99:8, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1964921.1964994>
- [26] Nintendo. Nintendo of America. "Super Mario World." 1991.
- [27] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–98, June 1986. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/21869365>
- [28] M. Oren, C. Papageorgiou, P. Sinha, E. Osuna, and T. Poggio, "Pedestrian Detection Using Wavelet templates," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997, pp. 193–199. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=609319
- [29] C. P. Papageorgiou, M. Oren, and T. Poggio, "A General Framework for Object Detection," in *Proceedings of the Sixth International Conference on Computer Vision*. IEEE Computer Society, 1998, p. 555. [Online]. Available: <http://dl.acm.org/citation.cfm?id=938978.939174>
- [30] Y. Freund, "Boosting a Weak Learning Algorithm by Majority," *Information and Computation*, vol. 121, no. 2, pp. 256–285, 1995. [Online]. Available: <http://www.recognition.mccme.ru/pub/papers/boosting/freund90boosting.pdf>
- [31] T. Bylander and L. Tate, "Using Validation Sets to Avoid Overfitting in AdaBoost," in *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, 2006, pp. 544–549. [Online]. Available: <http://www.aaai.org/Papers/FLAIRS/2006/Flairs06-108.pdf>
- [32] R. Venkatesan, S.-M. Koon, M. Jabubowski, and P. Moulin, "Robust Image Hashing," *Proceedings of the 2000 International Conference on Image Processing*, vol. 3, pp. 664–666, 2000. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=899541
- [33] H. Wolfson and I. Rigoutsos, "Geometric Hashing: An Overview," *Computational Science and Engineering, IEEE*, vol. 4, no. 4, pp. 10–21, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=641604
- [34] J. T. Schwartz and M. Sharir, "Identification of Partially Obscured Objects in Two and Three Dimensions by Matching Noisy Characteristic Curves," *International Journal of Robotics Research*, vol. 6, no. 2, pp. 29–44, 1986. [Online]. Available: <http://dl.acm.org/citation.cfm?id=33847.33850>
- [35] Y. Lamdan, J. Wolfson, and J. T. Schwartz, "Affine Invariant Model-Based Object Recognition," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 5, pp. 578–589, 1990.

- [36] R. Collins and W. Ge, “CSDD Features: Center-Surround Distribution Distance for Feature Extraction and Matching,” in *Proceedings of the 2008 European Conference on Computer Vision*, Marseille, 2008, p. 146. [Online]. Available: <http://www.cse.psu.edu/~rcollins/Papers/eccv08CollinsGe.pdf>
- [37] G. Pass, R. Zabih, and J. Miller, “Comparing Images Using Color Coherence Vectors,” in *Proceedings of the Fourth ACM International Conference on Multimedia*. Boston, Massachusetts: ACM, 1996, pp. 65–73. [Online]. Available: <http://doi.acm.org/10.1145/244130.244148>
- [38] M. S. Goldberg and J. Hunt. “Nintendo Entertainment System (NES) – 1985–1995 - Classic Gaming.” 2011. [Online]. Available: <http://classicgaming.gamespy.com/View.php?view=ConsoleMuseum.Detail&id=26&game=5> [Last Accessed: March 18, 2013]
- [39] M. Korth. nocash.emubase.de. “NES Specifications.” 2004. [Online]. Available: <http://nocash.emubase.de/everynes.htm> [Last Accessed: March 18, 2013]
- [40] Capcom Co., Ltd. Capcom. “Mega Man.” 1987.
- [41] FCEUX. “FCEUX.” 2011. [Online]. Available: <http://www.fceux.com/web/home.html> [Last Accessed: March 18, 2013]
- [42] FCEUX. “FCEUX Configuration Help.” 2011. [Online]. Available: <http://www.fceux.com/web/help/fceux.html> [Last Accessed: March 18, 2013]
- [43] E. W. Pugh, *Building IBM: Shaping an Industry and Its Technology*. MIT, 1995, p. 274.
- [44] M. Piccardi. University of Technology, Sydney. “Background Subtraction Techniques: A Review.” 2004. [Online]. Available: <http://www-staff.it.uts.edu.au/~massimo/BackgroundSubtractionReview-Piccardi.pdf> [Last Accessed: March 18, 2013]
- [45] Microsoft Corporation. “unordered_map.” 2012. [Online]. Available: <http://msdn.microsoft.com/en-us/library/bb983026.aspx> [Last Accessed: March 18, 2013]
- [46] Microsoft Corporation. “Microsoft Support - DIBs and Their uses.” 1992. [Online]. Available: <http://support.microsoft.com/kb/q81498/> [Last Accessed: March 18, 2013]
- [47] Nintendo. Nintendo of America. “Super Mario Bros. 3.” 1988.
- [48] Nintendo. Nintendo of America. “The Legend of Zelda.” 1987.
- [49] M. Swain, “Color Indexing,” *International Journal of Computer Vision*, vol. 7, no. 1, pp. 11–32, Nov. 1991. [Online]. Available: <http://www.springerlink.com/index/10.1007/BF00130487http://www.springerlink.com/index/N231L41541P12L1G.pdf>

- [50] C. Kanan and G. W. Cottrell, “Color-to-Grayscale: Does the Method Matter in Image Recognition?” *PLOS ONE*, vol. 7, no. 1, Jan. 2012. [Online]. Available: <http://dx.doi.org/10.1371/journal.pone.0029740>
- [51] A. Oliva and P. G. Schyns, “Diagnostic Colors Mediate Scene Recognition.” *Cognitive Psychology*, vol. 41, no. 2, pp. 176–210, Sept. 2000. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/10968925>
- [52] A. Torralba, R. Fergus, and W. T. Freeman, “80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1958–70, Nov. 2008. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/18787244>
- [53] Y. Amit, *2D Object Detection and Recognition: Models, Algorithms, and Networks*. The MIT Press, 2002, p. 13.
- [54] MIDI Manufactures Association, Incorporated. “MIDI 1.0 Specification.” 2001. [Online]. Available: <http://www.midi.org/techspecs/midispec.php> [Last Accessed: March 18, 2013]
- [55] Nintendo. Nintendo of America. “Super Mario Kart.” 1992.
- [56] Nintendo and Argonaut Software. Nintendo of America. “Star Fox.” 1993.

BIBLIOGRAPHY

- Amit, Y., *2D Object Detection and Recognition: Models, Algorithms, and Networks*. The MIT Press, 2002, p. 13.
- Andersen, J. "Where Games Go To Sleep: The Game Preservation Crisis, Part 1." Jan. 2011. [Online]. Available: http://www.gamasutra.com/view/feature/6271/where/_games/_go/_to/_sleep/_the/_game/_php/
- Bracken, C. C. and Skalski, P., "Telepresence and Video Games: The Impact of Image Quality," *PsychNology Journal*, vol. 7, no. 1, pp. 101–112, 2009. [Online]. Available: [http://www.psychology.org/File/PNJ7\(1\)/PSYCHNOLOGY_JOURNAL_7_1_BRACKEN.pdf](http://www.psychology.org/File/PNJ7(1)/PSYCHNOLOGY_JOURNAL_7_1_BRACKEN.pdf)
- Bylander, T. and Tate, L., "Using validation sets to avoid overfitting in adaboost," in *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, 2006, pp. 544–549. [Online]. Available: <http://www.aaai.org/Papers/FLAIRS/2006/Flairs06-108.pdf>
- Canny, J., "A computational approach to edge detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–98, June 1986. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/21869365>
- Capcom Co., Ltd. Capcom. "Mega Man." 1987.
- Collins, R. and Ge, W., "CSDD Features: Center-Surround Distribution Distance for Feature Extraction and Matching," in *Proceedings of the 2008 European Conference on Computer Vision*, Marseille, 2008, p. 146. [Online]. Available: <http://www.cse.psu.edu/~rcollins/Papers/eccv08CollinsGe.pdf>
- Dodgson, N. A., "Image resampling," Wolfson College and Computer Laboratory, University of Cambridge, Tech. Rep. TR261, 1992. [Online]. Available: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-261.pdf>
- FCEUX. "FCEUX." 2011. [Online]. Available: <http://www.fceux.com/web/home.html>
- FCEUX. "FCEUX Configuration Help." 2011. [Online]. Available: <http://www.fceux.com/web/help/fceux.html>
- Freund, Y., "Boosting a weak learning algorithm by majority," *Information and Computation*, vol. 121, no. 2, pp. 256–285, 1995. [Online]. Available: <http://www.recognition.mccme.ru/pub/papers/boosting/freund90boosting.pdf>
- Goldberg, A. and Flegal, R., "Acm president's letter: Pixel art," *Communications of the ACM*, vol. 25, no. 12, pp. 861–862, 1982. [Online]. Available: <http://dl.acm.org/citation.cfm?id=358728.358731>

- Goldberg, M. S. and Hunt, J. "Nintendo Entertainment System (NES) – 1985–1995 - Classic Gaming." 2011. [Online]. Available: <http://classicgaming.gamespy.com/View.php?view=ConsoleMuseum.Detail&id=26&game=5>
- Green, S. "Blargg's NTSC Libraries." 2011. [Online]. Available: <http://slack.net/~ant/libs/ntsc.html>
- Guttenbrunner, M., Becker, C., and Rauber, A., "Keeping the Game Alive: Evaluating Strategies for the Preservation of Console Video Games," *International Journal on Digital Curation*, vol. 5, no. 1, 2010. [Online]. Available: http://publik.tuwien.ac.at/files/PubDat_190893.pdf
- Jack, K., *Video Demystified*, 5th ed. Elsevier Science, 2007, pp. 247–249.
- Kanan, C. and Cottrell, G. W., "Color-to-grayscale: Does the method matter in image recognition?" *PLOS ONE*, vol. 7, no. 1, Jan. 2012. [Online]. Available: <http://dx.doi.org/10.1371/journal.pone.0029740>
- Kopf, J. and Lischinski, D., "Depixelizing pixel art," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 99:1 – 99:8, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1964921.1964994>
- Korth, M. nocash.emubase.de. "NES Specifications." 2004. [Online]. Available: <http://nocash.emubase.de/everynes.htm>
- Lamdan, Y., Wolfson, J., and Schwartz, J. T., "Affine invariant model-based object recognition," *IEEE Transactions on Robotics and Automation*, no. 5, pp. 578–589.
- Li, L.-J. and Fei-Fei, L., "What, Where and Who? Telling the Story of an Image by Activity Classification, Scene Recognition and Object Categorization." in *Computer Vision: Detection, Recognition and Reconstruction*, ser. Studies in Computational Intelligence. Springer, 2010, vol. 285, pp. 157–171. [Online]. Available: <http://dblp.uni-trier.de/db/series/sci/sci285.html#Fei-FeiL10>
- Li, L.-J., Hao, S., Yongwhan, L., and Fei-Fei, L., "Objects as Attributes for Scene Classification," in *Proceedings of the 2010 European Conference on Computer Vision, International Workshop on Parts and Attributes*, Crete, Greece, Sept. 2010.
- Liau Kie Fa, D. "2xSaI: the advanced 2x scale and interpolation engine." 2001. [Online]. Available: <http://vdnoort.home.xs4all.nl/emulation/2xsai/>
- Mazzoleni, A. "Scale2x Algorithm." 2001. [Online]. Available: <http://scale2x.sourceforge.net/algorithm.html>
- Mazzoleni, A. "Scale2x and EPX." 2011. [Online]. Available: <http://scale2x.sourceforge.net/scale2xandepx.html>
- Microsoft Corporation. "Computer Icon, Windows 3.1." 1992.

- Microsoft Corporation. "Microsoft Support - DIBs and Their uses." 1992. [Online]. Available: <http://support.microsoft.com/kb/q81498/>
- Microsoft Corporation. "unordered_map." 2012. [Online]. Available: <http://msdn.microsoft.com/en-us/library/bb983026.aspx>
- MIDI Manufactures Association, Incorporated. "MIDI 1.0 Specification." 2001. [Online]. Available: <http://www.midi.org/techspecs/midispec.php>
- Newman, R. and Sheffield, B., "10th Annual Salary Survey," *Game Developer*, vol. 18, no. 4, pp. 8–13, 2011.
- Nintendo. Nintendo of America. "The Legend of Zelda." 1987.
- Nintendo. Nintendo of America. "Super Mario Bros. 3." 1988.
- Nintendo. Nintendo of America. "Super Mario World." 1991.
- Nintendo. Nintendo of America. "Super Mario Kart." 1992.
- Nintendo and Argonaut Software. Nintendo of America. "Star Fox." 1993.
- Nintendo of America, Inc. "Virtual Console." Oct. 2011. [Online]. Available: <http://www.nintendo.com/wii/online/virtualconsole/>
- Oliva, A. and Schyns, P. G., "Diagnostic Colors Mediate Scene Recognition." *Cognitive Psychology*, vol. 41, no. 2, pp. 176–210, Sept. 2000. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/10968925>
- Oren, M., Papageorgiou, C., Sinha, P., Osuna, E., and Poggio, T., "Pedestrian detection using wavelet templates," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997, pp. 193–199. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=609319
- Papageorgiou, C. P., Oren, M., and Poggio, T., "A general framework for object detection," in *Proceedings of the Sixth International Conference on Computer Vision*. IEEE Computer Society, 1998, p. 555. [Online]. Available: <http://dl.acm.org/citation.cfm?id=938978.939174>
- Pass, G., Zabih, R., and Miller, J., "Comparing images using color coherence vectors," in *Proceedings of the Fourth ACM International Conference on Multimedia*. Boston, Massachusetts: ACM, 1996, pp. 65–73. [Online]. Available: <http://doi.acm.org/10.1145/244130.244148>
- Piccardi, M. University of Technology, Sydney. "Background subtraction techniques: A review." 2004. [Online]. Available: <http://www-staff.it.uts.edu.au/~massimo/BackgroundSubtractionReview-Piccardi.pdf>
- Pugh, E. W., *Building IBM: Shaping an Industry and Its Technology*. MIT, 1995, p. 274.

- Schwartz, J. T. and Sharir, M., "Identification of partially obscured objects in two and three dimensions by matching noisy characteristic curves," *International Journal of Robotics Research*, vol. 6, no. 2, pp. 29–44, 1986. [Online]. Available: <http://dl.acm.org/citation.cfm?id=33847.33850>
- Shoup, R., "Superpaint: An early frame buffer graphics system," *Annals of the History of Computing, IEEE*, vol. 23, no. 2, pp. 32–37, 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=929909
- Sony Computer Entertainment America, LLC. "PlayStation Network." Oct. 2011. [Online]. Available: <http://us.playstation.com/psn/>
- Stepin, M. HiEnd3D. "hq2x magnification filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq2x.html>
- Stepin, M. HiEnd3D. "hq3x magnification filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq3x.html>
- Stepin, M. HiEnd3D. "hq4x magnification filter." 2001. [Online]. Available: <http://www.hiend3d.com/hq4x.html>
- Swain, M., "Color Indexing," *International Journal of Computer Vision*, vol. 7, no. 1, pp. 11–32, Nov. 1991. [Online]. Available: <http://www.springerlink.com/index/10.1007/BF00130487><http://www.springerlink.com/index/N231L41541P12L1G.pdf>
- Thomas, K. MacTech. "Fast blit strategies: A Mac programmer's guide." June 1999. [Online]. Available: <http://www.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html>
- Torrallba, A., Fergus, R., and Freeman, W. T., "80 million tiny images: a large data set for nonparametric object and scene recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1958–70, Nov. 2008. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/18787244>
- Venkatesan, R., Koon, S.-M., Jabubowski, M., and Moulin, P., "Robust image hashing," *Proceedings of the 2000 International Conference on Image Processing*, vol. 3, pp. 664–666, 2000. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=899541
- Viola, P. and Jones, M., "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2002.
- Wolfson, H. and Rigoutsos, I., "Geometric hashing: An overview," *Computational Science and Engineering, IEEE*, vol. 4, no. 4, pp. 10–21, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=641604

- Zhang, D. and Chang, S.-F., “Event Detection in Baseball Video Using Superimposed Caption Recognition,” in *Proceedings of the tenth ACM International Conference on Multimedia*, ser. MULTIMEDIA '02. New York, NY, USA: ACM, 2002, pp. 315–318. [Online]. Available: <http://doi.acm.org/10.1145/641007.641073>
- Zhou, W., Vellaikal, A., and Kuo, C. C. J., “Rule-based Video Classification System for Basketball Video Indexing,” in *Proceedings of the 2000 ACM Workshops on Multimedia*, ser. MULTIMEDIA '00. New York, NY, USA: ACM, 2000, pp. 213–216. [Online]. Available: <http://doi.acm.org/10.1145/357744.357941>