GameScripter - A Vision Based Tool for Playing Games

Peter Lipka,
prl07@doc.ic.ac.uk,
Imperial College - DoC,
Supervisor: Dr. Andrew Davison

June 2011

Abstract

Over the past few years, computer vision and modern games have been increasingly crossing paths. Virtual reality has hit consumer markets on modern, camera-wielding smart phones while critically-acclaimed innovations, such as the Microsoft Kinect, have meant that games consoles have started to forgo traditional game-pad input devices, and moved into the realm of using the player themselves as input.

However, no research has been done into applying computer vision on the games themselves. Many bold statements have been said about the progress computers have made in challenging human players. Fundamentally however, they are flawed, as they all assume the AI can interact with the game through pre-built interfaces - an abstraction level not available to people.

In this project I plan to level the playing field. I create a framework that allows the user to easily write software with the ability interface with games in the same way as humans - by observing the screen. It provides a platform-independent way to parse game state through the use of computer vision and input commands using the game's traditional methods - game-pads or keyboards/mice. The project aims to showcase the feasibility of such an approach and I demonstrate this with a number of working examples.

Acknowledgements

I would like to thank my supervisor, Dr. Andrew Davison, for believing in me and accepting this self-proposed project. I am grateful for his guidance and great ideas.

I would also like to thank all my fellow students in DoC for their constant interest in my project and its progress on playing their favourite games. I would especially to thank Chris Emery for his unrelenting daily *"does it play Quake yet?"* enquiry.

Contents

1 Introduction **4**
1.1 Objective . . . 4
1.2 Motivation . . . 4
1.3 Challenges and issues to be solved . . . 6

2 Background **7**
2.1 Existing Solutions . . . 7
2.1.1 AI in games . . . 7
2.1.2 Poker Bots . . . 8
2.1.3 Cheating in Modern Games . . . 8
2.2 Domain Specific Languages . . . 10
2.2.1 Internal vs External DSLs . . . 10
2.2.2 External DSLs in Industry . . . 10
2.2.3 Extension Languages . . . 11
2.2.4 Lua . . . 11
2.3 Platforms . . . 13

3 Design and Implementation **15**
3.1 GameScript as a Domain Specific Language using Lua . . . 15
3.2 Overview of System Architecture . . . 18
3.2.1 Thread of control . . . 19
3.3 X11 Input/Output . . . 21

4 GameProfiles **23**
4.1 Whack-a-mole . . . 24
4.2 Generic Whack-a-mole . . . 32
4.3 Breakout . . . 40
4.4 Generic Breakout . . . 44
4.5 Generic Tetris . . . 57
4.6 First Person Shooter . . . 70
4.7 Real-world Application . . . 95

5 Evaluation **98**
5.1 Case-study . . . 98
5.2 GameScript . . . 101
5.3 Efficacy of vision . . . 102

5.3.1 2D Object recognition 102
5.3.2 3D Object recognition 103
5.3.3 Background/Foreground Segmentation 103
5.3.4 Optical flow 103
5.4 Performance 104
5.5 Game Applicability 104

6 Conclusions **106**
6.1 Things I have learnt 107
6.2 Future work 107
6.2.1 New Features 107
6.2.2 Hindsight 109
6.3 Closing Remarks 109

A Additional Figures **111**
A.1 Hu Invariant Moments 111

B GameScript Variables **112**

C GameScript call-outs **113**

D GameScript Standard Object Table **114**

E GameScript call-ins **115**
E.1 Input call-ins 115
E.2 Vision call-ins 116

Chapter 1

Introduction

1.1 Objective

The high-level objective for GameScripter is to create a framework which can interact with games like a human, for the purpose of playing the game, or augmenting the users actions in the game.

Unlike all current implementations of AI for games, which interact with the game through an application programming interface (API), this system will strive to interact with the game in exactly the same way as a human user. It will see the game in the same way as a human user; it will try to derive meaning from the output on the screen. It will also interact with the game in the same way; not through an interface provided by the game, but through an imitated mouse and keyboard on the PC, or an imitated game pad if on a console.

The idea behind the framework is to provide the user a system, controlled through a simple scripting language, which will allow them to create "GameProfiles" which run alongside the game. These will allow the user to write "rules" on how the system should parse the game state and react to game events.

1.2 Motivation

The greatest motivations of this project is that something like this has never been tried before. There have been only a limited number of successful projects creating software which is able to play games - the only viable projects are the variety of poker bots in existence which are designed to just play poker. However, as will be discussed in chapter 2:Background, these are of very limited scope and very prone to small changes in the game design. By using more advanced computer vision and machine learning techniques I hope to achieve a single piece of software that will be more adaptable and completely platform independent - it will work with PCs, games consoles, and in fact any games hardware.

The main area of research of this project involves computer vision alongside some machine learning. With the rise of powerful hardware and better algorithms, both these areas are rapidly expanding into exciting new fields, moving out of the 2 or 3 industries which they

have been prevalent in the past decade: medical imaging, machine vision in industry and military applications.

- Google revealed it has been testing self-driving cars on public roads for over a year now. These make extensive use of video cameras, which are complemented by radar sensors and lasers, to pilot the cars in busy traffic.
- CCTV is common-place now and there are commercial systems available that analyse the images and inform users of potential threats such as unattended bags.
- Movie studios are using computer vision to capture actor expressions and translate them into animated character expressions.
- Even the rapid rise of smart-phones has led to novel new ways to use computer vision to exploit on-board cameras and processing power. Augmented reality applications such as Layar can overlay 3D images over the terrain in real-time. Other applications are more useful and provide a valuable tool to their users, such as Word Lens which is able to read written text, translate it and then super-impose the translated text in real-time[1].

The motivation behind applying computer vision and machine learning to games instead of the real environment are:

- Computer games are a more constrained environment - you no longer have to worry about the deficiencies and imperfections of cameras and camera lenses.
- You can set up the environment as you want it; consistent and easily reproduce-able.
- Getting data to test on from a computer game is far simpler than having to use cameras and film. This is important when I will be investigating machine-learning which will require large datasets.
- All this makes automated testing and analysing results easier for evaluation

This project could bring a lot interesting possibilities further down the line. As the software will interact with games in an identical way to humans, then in theory it would be impossible to detect. Unlike other tools used by people who wish to cheat and boost their gaming ability, such as aim-bots (see subsection 2.1.3:Cheating in Modern Games), this software will not exploit bugs in the games or modify them in anyway. The credibility of scores in online games would diminish as nobody will be sure they have been achieved without the help of any tools. This would be true even on the traditional safe-havens; consoles. At the moment cheating through the use of 3rd party tools is rare on consoles, as they are vendor locked down and only manufacturer-approved and signed applications can be run on them. The only way to run 3rd party home-brew applications is to modify the consoles, either in software or often physically with a *mod-chip*[2]. Often this results in the banning of consoles from playing online, due to possible piracy and cheating.

[1] `http://questvisual.com/`

[2] http://www.gamerlaw.co.uk/2010/01/are-modchips-illegal.html

1.3 Challenges and issues to be solved

As will be discussed in the background section, there is very little research into the area of software playing games, so I believe this project is breaking new ground. The project is very open-ended, and with no comparisons available, I decided to take an iterative approach. I started from simple 2D games and built towards creating a framework that will be able to handle complicated, photo-realistic 3D games. I have managed to produce a framework, which I believe shows that this goal is certainly possible, for at least a subset of some of the most popular games available.

The project relies much on computer vision and investigates the use of machine learning, so I was able to refer to papers on these subjects. However, the task is not a simple one - there exist entire PhD papers on matters which I need to be able to solve efficiently - e.g. object detection. Throughout the project I tried to gather and effectively use as many of these existing resources, usually choosing to use or implement an existing algorithm rather than designing my own. This has resulted in a wide-variety of tools available to a GameScript, which helps increase capability of the tool to work over a wider variety of games.

I also had to take into consideration performance. The best performing algorithms in terms of accuracy, tend to be the slowest too. For turn-based games this may be less of an issue, but most games are real-time where reaction speed counts! Where applicable I tried to investigate, and test various algorithms, often switching between multiple algorithms when necessary.

Throughout the report I will also discuss what type of games lend themselves to computer vision analysis, and which ones are more difficult.

The software also aims to be completely platform-independent. It works well on it's development platform - a standard PC, but it should also work with consoles such as the Nintendo Wii and Sony Playstation 3. GameScripter runs on the PC, but with the ability to connect the output from the console to the PC (in the form of video, usually connected to a TV) game state can be parsed in the same way as if the game was running on the host platform. I chose to use the Playstation 3 as a proof of concept.

Chapter 2

Background

My project is pretty unique in that there is little research available in the area of computers observing and playing games through the use of computer vision. While there is vast amounts of research on the use of machine-learning and AI in games, it all involves the use of low-level API access to the game, where the AI has easy access game state. Research in computer vision in games is limited to actually observing the player themselves rather than the game, and using that as input to the game. The widely successful Microsoft Kinect[1] uses this concept.

I will investigate the techniques used in the real world and see their applicability to 2D and 3D games. The project has a few very distinct areas which I have split up into their own sections:

- Existing solutions
- Domain Specific Languages
- Platforms
- Computer Vision and Machine Learning

I have put background information on computer vision and machine learning under separate, smaller background sections in chapter 4:GameProfiles. I felt that there, they provided more clarity and context.

2.1 Existing Solutions

2.1.1 AI in games

Many games appear to have very advanced AI, however most of the time, this is just an illusion[1]. AI built into games has the advantage of being able to access all possible game-state, including state unavailable to a human opponent. In many cases this gives them a

[1]http://www.xbox.com/en-GB/kinect

massive advantage over a human player. Some games take it further and actually cheat the game mechanics; e.g. giving themselves more money than human players. Finally, as the AI is constricted to a single game, it tends to be very scripted through the use of finite state machines and decision trees where every the action needs to programmed for all possible scenarios[2]. It will be unrealistic for my system to just "learn" how to play any games with complex rules; like AIs in industry, this will have to be scripted too.

2.1.2 Poker Bots

Poker-bots share a subset of my project's goals. Poker-bots such as OpenHoldemBot[2] are programs which automatically play poker (usually Texas Holdem) on online poker sites. As the use of these bots is virtually always against terms and conditions of the companies that run the websites (their legal status is disputed) they do not use any API to connect to the poker games. Instead they have to interact with the games in the same way as humans to avoid detection. This means they usually use screen-capture to gain information about game state, and mouse/keyboard imitation to interact with the game. In fact, as they are becoming a major problem on online sites, many websites search your computer for the existence of these bots, but this is usually unsuccessful as these bots can run on isolated virtual machines to avoid detection. In this sense they fulfil my requirements of playing the game as a human would, but similarities end there.

Poker-bots are reasonably simple programs. You log into your poker website online, join a table and then run the program, specifying the coordinates of where the web-browser is positioned. The screen is then captured and custom-made table maps are applied to interpret game state (see figure 2.1). These table maps have to be custom built for every online poker website - they define where state can be parsed from - such as where every player is sitting, which bit of text corresponds to which player's pot and where the cards are laid out. Apart from the use of optical character recognition to read size of the pot, there is no real computer vision or machine-learning used. The need for modifications of the table-map are frequent - when the website decides to change the look of the table, often even colour or texture changes will require a modification of the table map.

The second part of poker-bots is the AI itself. Once game state has established using table-maps, it is fed to the AI which can calculate the next move of the player. The poker-bots then imitate mouse/keyboard inputs to fulfil the AI's requests, based on the areas to click defined by the table maps.

2.1.3 Cheating in Modern Games

Undoubtedly, running a GameProfile that can play a game, or give the player an advantage can be classified as "cheating". However, unlike current cheats, my system does not exploit or modify the game in any way.

For example, one of the most popular cheats is an aimbot - used in first-person shooter (FPS) games to assist the player in target acquisition - usually achieved by moving the weapon

[2]http://code.google.com/p/openholdembot/

cross-hair towards the enemy. All current aimbots use one of the following techniques[3][3]:

- **Client Exploitation**. These work by modifying the executable on the client machine, or by directly patching the instruction cache. This means program flow can be manually redirected in order to give the player an advantage.
- **Colour Aimbots**. These were once very popular aimbots in earlier FPS games. They work by skinning the enemy players to give them a distinct colour (e.g. giving them a pink uniform). On each frame, each pixel is scanned until the right colour value is found, identifying an enemy. However this is not used much in modern games, as skinning tends to be banned, and state-of-the-art visual lighting effects severely distort colour recognition accuracy.
- **Graphic driver-based Aimbots**. These hijack the graphical APIs such as DirectX and OpenGL. Simple analysis on the polygon models can be performed which is used to determine enemy positions.
- **Network Exploitation**. In many online games, game state is shared through the internet. By analysing the incoming/outgoing packets, game state can be deduced, giving the player an advantage. Even the presence or absence of encrypted internet traffic can give away some information about game state.

[3] `http://en.wikipedia.org/wiki/Aimbot`

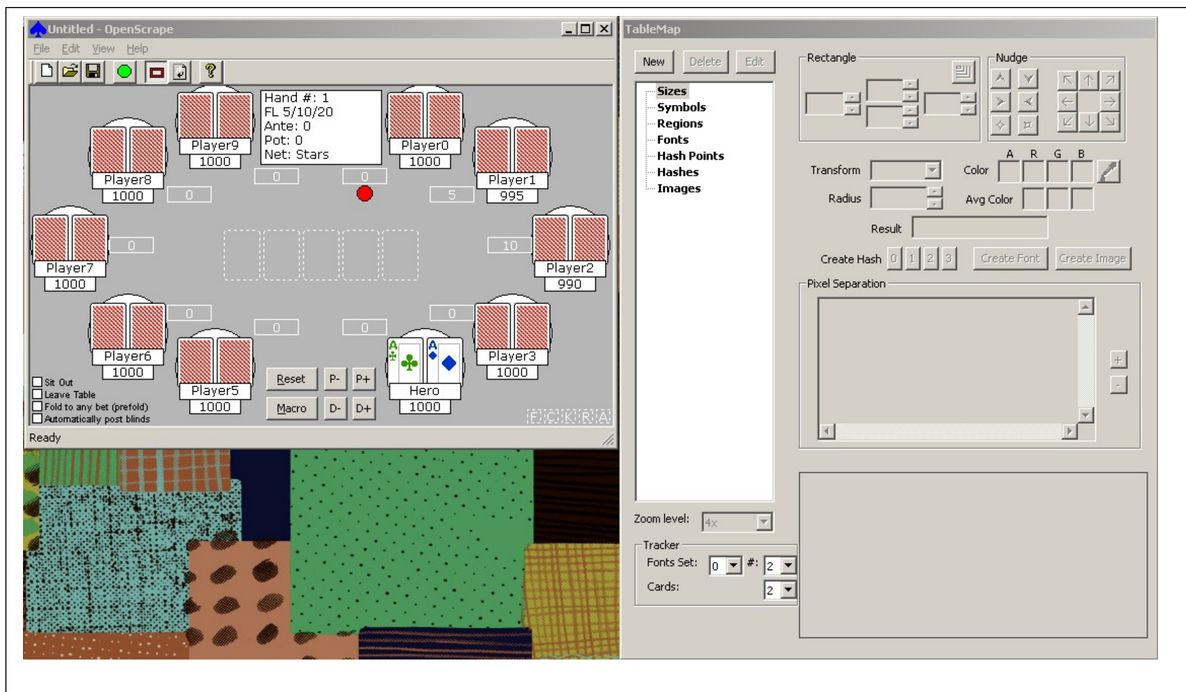

Figure 2.1: Creation of a Table Map using the OpenScrape framework of OpenHoldemBot.

2.2 Domain Specific Languages

Computer Vision and imitated keyboard/game-pad input will provide the means for Game-Scripter to play a game, however the aim of the project is to provide a framework which is easily accessible to users wishing achieve this. They will want to create game-specific GameProfiles, which they can easily tweak and modify until they get the results they wish. Therefore, forcing them to code in a general-purpose-programming language, such as Java, or in this case C++, is not optimal. I would like a system, that would run in the background, scripts can be written quickly & easily, dynamically run without needed compile/restart phases, and be very simple and intuitive. There are many systems out there that require these same requirements, and these are usually solved with the use of some kind of Domain Specific Language (DSL).

Domain Specific Languages are programming languages which are specific to the domain of the problem that a system has been created to solve. By having their functionality tied to a particular system, they allow solutions or problems to be expressed much more clearly than existing languages. Looking at the code should convey information about what the code is doing rather than just providing the functionality of actually doing it. Once a DSL has been defined, developers and end-users can spend more time thinking about the problem itself and less time on code.

2.2.1 Internal vs External DSLs

There are two types of domain-specific languages: *internal* (or *embedded*) languages and *external* languages. An internal DSL is using ones own defined language within another language (the host language). This includes ideas as simple as naming methods and variables sensibly, using certain design patterns and custom types. External DSLs are completely custom languages, where the syntax, grammar, keywords and semantics must all be defined. External DSLs give maximum flexibility but involve a big upfront effort, which includes writing your own tools such as a parser.

2.2.2 External DSLs in Industry

Logo is one of the simplest examples of a domain-specific language, one with which many children are introduced to programming. It provides a way to interact with the program moving a robot without using a GUI. The following program is written in LOGO and is incredibly simple to understand.

```
PENUP
FORWARD 50
LEFT 140
PENDOWN
FORWARD 100
LEFT 90
```

It's very concise, with no "boiler-plate" code, no need to worry about any memory management or any complicated language specific issues. This is what I want to achieve for my DSL - "GameScript".

Another popular tool, which is often seen as a DSL is SQL. It's sole purpose is for managing data in relational database management systems (RDMS). It has been far more successful than the original system it was designed for (System R) and is used in virtually every relational database in existence. It is much more powerful than Logo, supporting many advanced programming constructs such as user defined functions, arithmetic, loops and recursion, but it has retained it expressibility and conciseness.

```
SELECT *
    FROM Book
    WHERE price > 100.00
    ORDER BY title;
```

2.2.3 Extension Languages

There is a lot effort involved to create your own DSLs, and as a result, there is a growing trend to use an alternative instead: extension languages. A number of languages have been designed so that they are easily embeddable in application programs, while retaining DSL-like characteristics. Examples include AngelScript, MEL and Lua. These are languages on their own can be classified as general-purpose-programming languages, but have features that lend themselves to being used as extension languages, not least easy interfacing with other more mainstream languages such as C, C++ and Java.

2.2.4 Lua

Lua is described by its creator Roberto Ierusalimschy, as "the language of choice for anyone who needs a scripting language that is simple, efficient, extensible and portable"[4] It was designed from the beginning to be integrated with software written in C and other conventional languages. It does not try to do what languages like C already do well, such as achieve good pure performance, but instead gives the user a simple high-level language with dynamic typing, automatic memory management and high order functions, so that the user can express their problems in a much more natural way. It is also seen as glue language in that it can easily glue together existing high-level components, usually written in more mainstream compiled languages, such as C or C++[4].

Lua is a "multi-paradigm" language, providing a very small set of features that can be extended to fit different problem types. For example, Lua does not contain explicit support for object orientation or inheritance, but these can be easily implemented using Lua metatables. This allows Lua to be tailored to suit the application, keeping it small and efficient.

Due to these features, Lua seems like an ideal candidate for my project. In fact, Lua has had great success in industry too, with many applications using it as their main application scripting language. Examples include **Adobe Photoshop Lightroom**[4], which uses Lua for

[4] http://www.adobe.com/devnet/photoshoplightroom.html

its user interface, or **Cisco**, which uses Lua to implement Dynamic Access Policies[5] on it's routers. However the real proponent of Lua is the games industry itself where it has become the most dominant scripting language[5]. Hundreds of modern games solely use Lua to script various in-game elements - such as player animations, game AI and GUI scripting.

Lua can be used in two ways - as *extension language* and as a *extensible language*. When used as an extension language, the core of the program has the control and Lua is the library; hence the core is *application code*. When used as an extensible language, Lua has the control and the core is the library. Here the core is called *library code*. As will be seen in section 3.1:GameScript as a Domain Specific Language using Lua, I plan to use a mixture of both these concepts in GameScripter.

SpringRTS

SpringRTS is an open-source game engine for real-time strategy games, and it heavily relies on Lua for its core functionality [6]. It shares a lot of common requirements with my project, so it is a useful resource to investigate. A Lua script in SpringRTS is called a `widget` and can be dynamically loaded & unloaded while spring is running. All performance-heavy algorithms are implemented in C++, and it uses a simple system of *call-ins* and *call-outs* to allow them to affect behaviour of the game.

- *Call-outs* - functions defined in your script that Spring Engine calls when a determined event takes place. For example, the `gadget:Initialize()` call-in is run by Spring when the widget is loaded.
- *Call-ins* - functions defined in the Spring engine (in C++) which you can run at whatever moment you desire. For example, calling `Spring.GetUnitTeam(unitID)`, returns the team of the unit identified by `unitID`.

SpringRTS uses call-outs so that the developer can run their own code when any in-game event happens. For example when a new unit that has been created and leaves a factory, any loaded script that has the appropriate `UnitFromFactory` call-out will be invoked (see figure 2.2). `Spring.GiveOrderToUnit()` is an example of a call-out.

```
function widget:UnitFromFactory(unitID, unitDefID, unitTeam,
                               factID, factDefID, userOrders)
   Echo("A unit has left the factory")
   Spring.GiveOrderToUnit(unitID, CMD.freeRoam)
end
```

Figure 2.2: When a new unit has been created, an order is given telling the unit to freely roam, looking for enemies

[5] http://www.satori.org/2009/03/the-engine-survey-general-results/
[6] `http://springrts.com`

The `Spring` object (in fact it's a Lua table) is initialised with all functionality that Spring provides to script-writers upon loading the script. In this case, `Spring` provides an method to give orders to units.

2.3 Platforms

As my system will interact with games just like a human, in theory it should be possible to make it completely platform-independent. The only pre-requisites for a platform to be supported are as follows:

1. A way to plug the output of the platform device into the computer running the software so that it can be analysed
2. A way for GameScripter to send input and interact with the platform device.

The platforms I intend to target are personal computers and the major consoles. However there is nothing stopping the software being used elsewhere, e.g. on mobile phones, so long as there is a way to achieve the 2 above requirements.

Personal Computers

This is the platform which I concentrated on, and on which the software will actually run on. Most of the testing was be done on this platform but there is no reason why everything tested here should not work with any other platform.

Sony Playstation 3

The second platform I tested on is the Sony Playstation 3 (PS3). The PS3 is a console released by Sony in 2006. Like most other consoles, to play it you need to connect it's video output to a television. This is the output that will need to be analysed by the system. There are 2 possible ways to do this: through the multitude of possible analogue outputs (such as component) or through HDMI. HDMI is the preferred choice as it is a digital connection so there will be no need to deal with noise (however small), as would be the case with analogue outputs.

The controller used by the PS3 is the Sixaxis controller and it communicates with the PS3 over bluetooth. Sending input to the PS3 can be achieved by spoofing a sixaxis controller - i.e. the computer pretends to be the Controller so it is able to communicate directly with the PS3. To make it possible for my software to analyse the user's inputs to the system, a man-in-the-middle attack could be implemented as shown in Figure 2.3. The sixaxis controller is actually connected via bluetooth to the computer. The computer receives input from the users' PS3, analyses it and then passes them on directly to the PS3, augmenting it if necessary.

As mentioned in the introduction, what makes it very interesting to target a console such as the PS3, is that consoles are usually regarded as not susceptible to cheating.

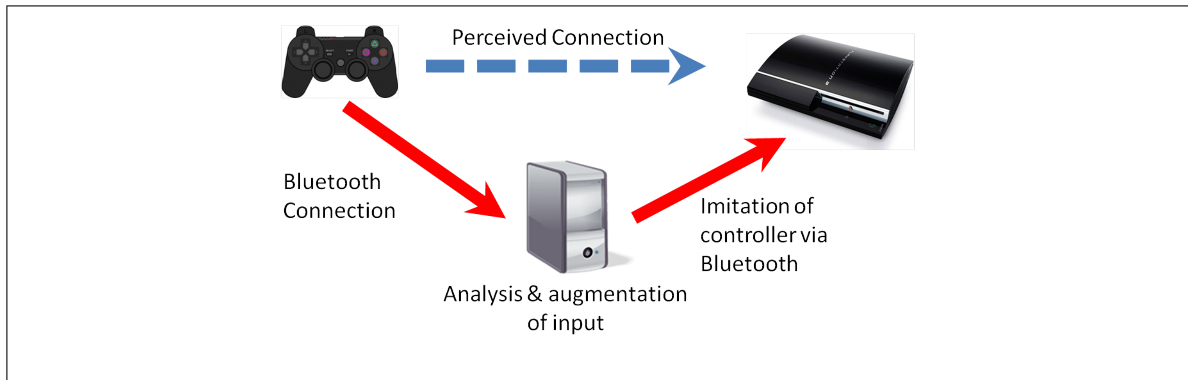

Figure 2.3: Man In Middle Attack on Sixaxis Controller

Nintendo Wii

The Wii was released by Nintendo in November 2006, and is the best selling current generation console.

Unfortunately the Wii does not have a digital output; the best quality is given through an analog, 480p (853x480) output. Although the picture may have some noise to it, the algorithms used in the software should be affected very little; however this assumption would need to be tested. The advantage of having such a low resolution (2.2x lower) is that performance will be less of an issue as most vision algorithms' performance decreases in line with increases in resolution.

Like the PS3, the controllers use a bluetooth protocol to communicate with the console. This should make it simple to spoof a Wii controller on a computer, and implement a similar man-in-the-middle attacks as described above.

Microsoft Xbox 360

The Xbox 360 is a console released by Microsoft in 2005. Newer Xbox 360s also have a HDMI output, so this would be the input of choice (it also runs at the same 720p resolution as the PS3). However the controllers use a proprietary 2.4 GHz protocol, which would need to be reverse-engineered so that it could be intercepted. Another other option would be to use Xbox wired controllers, which run a proprietary protocol over standard USB plugs. The final resort would be to take apart a controller and physically connect the appropriate switches in the controller to a microprocessor (e.g. an arduino). This could then interface with a PC, and trigger button presses.

Chapter 3

Design and Implementation

3.1 GameScript as a Domain Specific Language using Lua

GameScript is the name I have chosen for the language GameProfiles are to be written in. It was an unrealistic goal for me to write my own complete DSL from ground-up, so I decided to use an extension language as the core language of GameScript, specifically Lua (see 2.2.4). Although the GameProfiles are parsed using a Lua interpreter, I was able to tweak the language so that it acted very much like a DSL, while retaining all the benefits of using an existing, heavily tested programming language.

- I can use existing Lua language constructs in GameProfiles - everything from simple variable assignments, for loops and functions to advanced features such coroutines and closures.
- GameProfiles can build upon other GameProfiles without needing them to be rewritten or copied by using the Lua `loadlib()` command, which loads other Lua files.
- GameProfiles can load lua modules (libraries) which provide further functionality. These can be written in any language conforming to the Lua library specification. For example a GameProfiles could load a networking library with a single line `loadlib(httpd)` and now a GameProfile has access to networking utilities. I foresee this feature being used extensively to load external artificial intelligence needed to play games, with GameProfiles being used just to parse the game state.

Communication between Lua and C++ will use the Lua C API - a set of functions that allow C code to interact with Lua:

- Functions to read and write Lua global variables.
- Functions to run pieces of Lua code.
- Functions to register C functions so that they can later be called by Lua code.

This is done through an omnipresent virtual stack. All data exchange between Lua and C, such as arguments to functions and function results, will pass through this stack. This solves the impedance mismatches between Lua and C[4]:

1. Lua is garbage collected while C is not.
2. Lua is dynamically type while C is statically typed.

I took inspiration from the Lua scripting in the Spring Engine (see subsection 2.2.4:Lua) for the way GameProfiles communicate with the rest of the program. Firstly scripts can be dynamically loaded and unloaded during run-time. This is especially useful for testing, as GameProfiles can be modified and then reloaded at the press of a button without any need for down-time or recompilation. Similarly to the Spring Engine, there are 3 classes of functions in GameScript: call-ins, call-outs, local functions.

Call-ins There are two classes of call-ins - `input` call-ins which interact with the game itself such as `input.moveCursor()` and `vision` call-ins which interact with the vision core of GameScripter such as `vision.startBackGroundSubtraction()`.

Call-outs Call-outs are functions which will be invoked upon by the the core of GameScripter when events happen in-game. Examples include `callouts.newFrame()` which is called on every new frame or `callouts.buttonsPress(button)`, which is called every-time a button is pressed on the keyboard or game-pad. The button string (e.g. 'A' or 'Up') is passed in as an argument.

Local functions These are the same as functions in virtually any other language - user-defined functions that can be used anywhere in a GameProfile, and have no connection with the core of GameScripter.

In this kind of setup I'm using Lua as both extension language and as an extensible language (subsection 2.2.4. The core of GameScripter cannot easily be placed in the realm of application code or library code, as although the C++ core has the overall thread of control (this will be explained in section 3.2, all functionality of what GameScripter does is defined by the GameProfiles.

Apart from call-ins and call-outs, there is also a third method of sharing information between the core and GameProfiles, although it's only uni-directional. We can access the global state of a GameProfile from within the core, so we are able to set global variables as configuration parameter, which are loaded on start-up:

```
targetFPS = 30
screenZeroCoordinate = {x=100,y=200}
screenSize = {500, 600}
```

This script will try to run at 30 frames per second, and only analyse a subset of the screen, defined by the screenZeroCoordinate and size. Technically it would be possible to

expose many low-level parameters of the computer vision algorithms, however I have actively tried to avoid this situation. The main aim of GameProfiles should be that they are simple to write. Exposing individual parameters to the end-user would require them to know the inner-workings of the algorithms involved, which is undesired.

The full power of GameProfiles, written in GameScript will be shown in the the next section. There, most call-ins and call-outs will be described, however for a full list of all call-ins, call-outs and configuration parameters please see Appendix C.

One nice feature of using the Lua C API, is that all state for the Lua interpreter is held in a single structure (of type `lua_State`). The Lua library is fully re-entrant - there are no global variables. This means that it is very easy to open up multiple Lua files at the same time as each has their own state encapsulated in a single instance of `lua_State`. I have designed all functions that communicate between the C++ core and the GameProfiles to take a `lua_State` as an argument - hence adding the ability to actually run multiple GameProfiles concurrently.

3.2 Overview of System Architecture

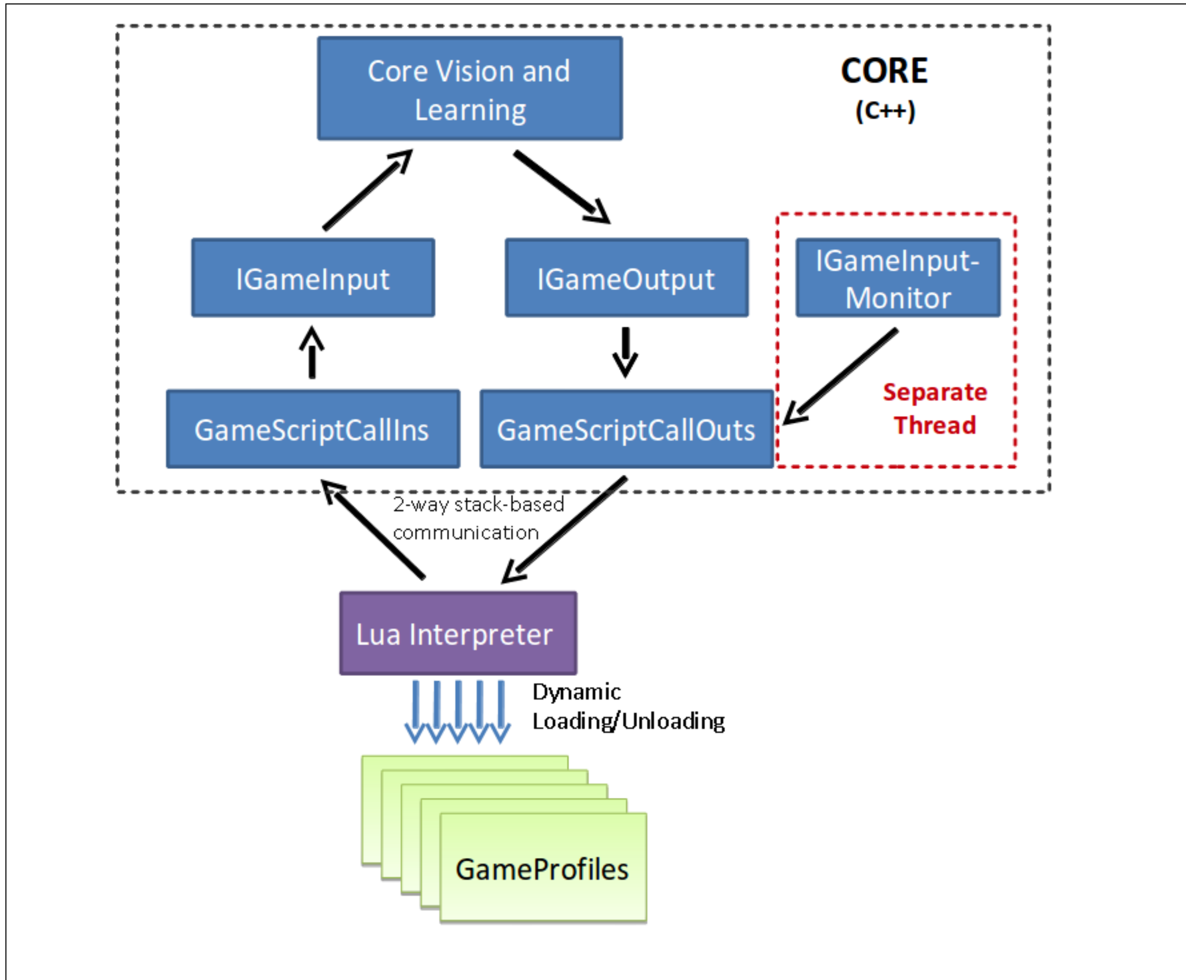

Figure 3.1: High Level representation of System Architecture

From the start I designed the system to be extensible and open to new platforms. All the computer vision and machine learning algorithms have been implemented in C++, and reside in a collection of classes encompassed by the "Core Vision and Learning" node. The `IGameInput` and `IGameOutput` interfaces provide a modular way of supporting new game machines. A concrete implementation of these interfaces is all that is required for each game console, isolating the rest of the system - including all vision algorithms and GameScript call-ins/call-outs (see section 3.1:GameScript as a Domain Specific Language using Lua), from any kind of dependency of the platform type. The default concrete implementation for the `IGameInput` is `X11Input`, for the `IGameOutput` is `X11Output`, and `IGameInputMonitor` is `X11InputMonitor` as described in section 3.3.

IGameOutput A concrete implementation will provide an interface with which one can access the output provided by the game - e.g. the video produced. It provides functions such as: `getCurrentImage()` which returns the current image produced by the game. In the future, this can be extended to expose sounds created from the game too.

IGameInput This provides a platform-independent way to send commands to the games console. It provides functions such as `moveCursorTo(x,y)`, `pressButton(Button)`

IGameInputMonitor The concrete implementation will run in a separate thread to the program logic of the rest of the program. It monitors the users interaction with the game, processes them, and forwards them to the appropriate GameScript call-ins. It needs to run in a separate thread, as actions performed by the user may need to be acted on immediately. If a computationally-expensive computer vision task is running, we cannot wait until it has finished before processing user input.

3.2.1 Thread of control

GameProfiles, written in GameScript, define the functionality of the program through the use of call-ins & call-outs like described in section 3.1, but they do contain the main thread of control. This is done by the Control class, which controls the core cycle that can be seen in figure 3.1. It is not explicitly in the figure.

Below is a summary of what the tasks Control class does:

Initialisation Initialises all the necessary classes and the default concrete classes.

GameScript Choice Parses the working directory for GameProfile and displays a GUI to allow the user to select the profile.

Initialising Chosen GameProfile It performs the following tasks on the chosen GameProfile:

- Checks the GameProfile for syntax errors.
- Parses the GameProfile and runs it using the Lua interpreter, so that all initialisation code is run
- Accesses the GameProfile state to load parameters such as `TargetFPS`, `MaxFPS`, and screen region of interest.

At this point, the IGameInputMonitor thread is started so that user input can be monitored.

Main loop of program Control also handles the main program loop. Each iteration of the loop analyses only one frame from the game.

- Get a new image from `IGameOutput` which is the frame to analyse.
- Informing the Vision & Machine Learning classes that a new frame available so that they are able to per-frame tasks (e.g. if they are in the process of learning the background model, that would be done at this point).
- Calculate the current FPS, and inform the Vision & Machine Learning classes of this, so that they can adapt to try and hit the target FPS.
- Call the `newframe()` call-out of the GameProfile.

- If the $TargetFPS < currentFPS$, slow the program down, by delaying the time before the next iteration.
- Listen to user-input for commands such as pausing the program or shutting it down.

Deallocation of Resources Once the main program loop has been stopped, the `closeGameProfile()` call-out is called and all resources are deallocated.

For an empty GameProfile, the main loop will run as follows. It will loop indefinitely, each frame it will take a new snapshot of the game image, and inform the vision modules that there exist new frames. However these will have no tasks to do (as none have been defined in the GameProfile), so nothing will be done. The program will end when the user presses the exit key (by default this is "Escape").

One interesting feature of GameScripter is that it can run in 2 modes - normal or debug. Debug mode gives the user feedback on what the application is doing at any-time through the use of debug windows. For example if GameScripter is running background subtraction and finding foreground objects, windows will pop-up showing the result of any operations Other information such as current FPS will also be available to the user. The Control class controls the mode by listening to user input - by default one can turn on debug mode by pressing Alt-D.

Of-course, many of these debug windows require extra processing and drawing to the screen, so performance takes a hit (5%-30% depending on GameProfile), however it can be very useful for the user to see the results of the computation to gain an insight into how the algorithms are performing.

3.3 X11 Input/Output

My default development platform was a linux system[1]. To be able to interact with games within linux, I needed GameScripter to be able to interact with the system linux uses to display graphical user interfaces - the X Window System. This was done by implementing the 3 simple interfaces described above. This however it proved to be a harder task than I initially envisaged. X Windows implements a client-server model. The server communicates with clients, handles requests for graphical output (e.g. display this window) and sends back user input (from the mouse or keyboard).

Requirements

GameProfiles have have the requirements to run alongside the games, without interfering with the ability to play the game, unless specifically scripted to do so. In other words, it should be possible to write a profile that records what the user sees and how they react (i.e. mouse movements, key presses) without the user being necessarily aware of this happening. Unfortunately these are also the *exact* same requirements that many security compromising tools also strive for, such as key loggers. The X Window System (or just "X" for short) was designed with security in mind, and as a security feature allows only 1 application at a time to listen to keyboard commands at any one time. This does not impede in daily use with the computer - only one application is "in focus" at any one time, and only that application receives commands from the keyboard.

The reason why some shortcuts seem to work across all applications, such as alt-tab, is because an application actually registers interest in only a subset of the keys. This allows the keys that haven't been registered, to be passed up the window hierarchy safely, without ever being intercepted.

Simulating keyboard presses is also impossible with standard X. For an application to be able to send key events to the server, it usually needs to "grab" the keyboard (or a subset of keys). However only one application can grab a keyboard at any one time, hence this methodology is not suitable for GameScripter.

Solution

There exist 2 extensions to the X windows system that provide a solution: XRecord[2] and XTest[3].

XRecord supports the recording and reporting of all core X protocols within the X server itself. It provides a mechanism for capturing all events, including input device events that do not go to any clients. X11InputMonitor uses this extension extensively to record the users actions. The basic principle is that a client (in my case GameScripter) creates a XRecordContext structure, specifying what type of events it is interested in, and then requests

[1] My linux distribution of choice is Ubuntu 10.10
[2] refspecs.freestandards.org/X11/recordlib.pdf
[3] www.x.org/releases/X11R7.5/doc/Xext/xtestlib.pdf

the XServer to be informed of any of those. Protocol data that interest was expressed in is recorded and returned to the recording client via a callback with a XRecordInterceptData object. This contains all the relevant information of the event - e.g. if it was a buttonRelease event, it would specify the x, y coordinates, the button that has been released, as well as other information such as what window it occurred in.

XTest was designed as a minimal set of client and server extensions required to completely test the X11 server with no user intervention. It provides limited synthesis of input device events, almost as if a cooperative user had moved the pointing device or pressed a key or button. The X11Input class makes full use of this extension to synthesize "FAKE_EVENT_TYPE" input events to the X11 server such as KeyPress, KeyRelease, ButtonPress, ButtonRelease, MotionNotify.

Problems

Unfortunately these 2 extensions have been plagued with problems. At the time of starting my project, both XTest and XRecord had critical bugs with the latest version of X, rendering them unusable. Instead I tried to adopt a different approach: Polling the special device files under /dev/, corresponding to the keyboard and mouse (i.e. /dev/input/mouse0, /dev/input/kb). I had 2 problems with this approach:

1. To be able to access these device files, I required root privileges. Although this would be OK for this project, it would be unsatisfactory in a final polished version
2. I was reading the raw data. This would have to be parsed and translated into something that made sense.

I eventually abandoned this methodology due to unsatisfactory results and the time-consuming task of having to decipher the raw data. Instead I downgraded my to a version of X that would work correctly with the XRecord and XTest extensions. It was also necessary to set up the X server to run in a multi-threaded environment as the X11InputMonitor and X11Input ran in separate threads, otherwise X would often crash.

Chapter 4

GameProfiles

I chose to use an iterative technique for writing GameScripter, basing it on real-world games, rather than speculate in advance what functionality I would need. In this section I document some of the games that have helped mould GameScripter into the state it is today. Each game builds upon the previous ones; extending the functionality of the GameScript language and the core vision techniques themselves. For each game, I will try to document the following:

- The game itself.
- The desired GameProfile I aimed for playing this game.
- Background on existing vision and machine learning techniques for solving similar problems.
- Implementation
- Results and possible improvements.

All benchmarks have been taken on my development machine:

Processor : Intel Core i7, with 4 cores (8 logical) running @ 2.66 GHz.

RAM : 6 GB DDR3 SDRAM running @ 1333 MHz

GPU : Nvidia GTX 570 with 480 CUDA cores running @ 1464 MHz.

GPU Memory 1280MB GDDR5 running @ 1900 over a 320-bit interface.

4.1 Whack-a-mole

Introduction

Whack-a-mole is one of the simplest games in existence. The idea is that there are a number of holes from which ”moles” (or any in general, any creature) can pop out and the players task is to whack these moles, usually by clicking on them. Scoring usually remains constant across implementations - the aim of the game is to hit as many moles as possible. Most of the games have a time limit, during which you try and score as many points as possible, but some have a limited number of lives. These are lost when a mole manages to pop-out then pop back into the hole, without being hit.

It was a good, simple game to start my iterative approach on.

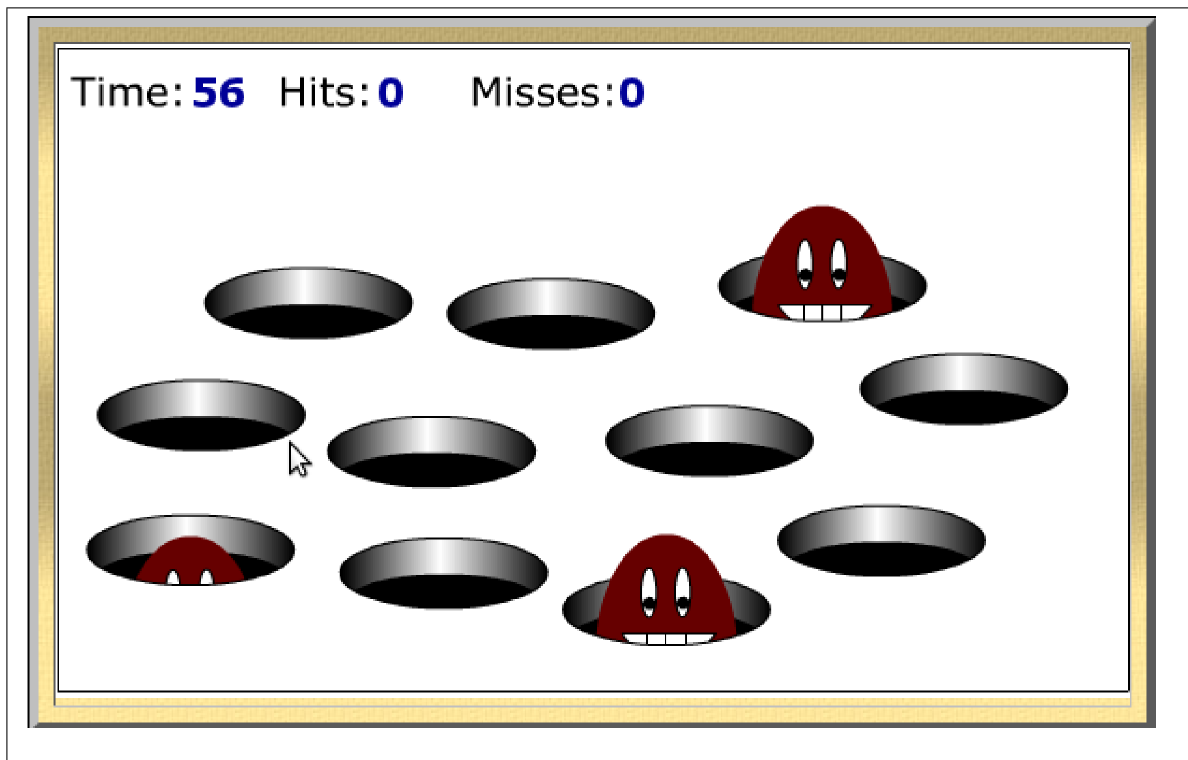

Figure 4.1: Whack - a simple online Whack-A-Mole game

Desired GameProfile

```
function callOuts.newframe()
  x, y = vision.matchImage("mole.png")
  input.clickOn(x,y)
end
```

Here we introduce 1 call-out - `callOuts.newFrame()` and 2 input call-ins - `input.clickMouse()` and `vision.matchImage()`.

`callOuts.newFrame()` is called every new frame. Note that this not every new frame of the game itself, but every time 1 iteration of the control loop has finished, and a new image has been received from the game. (see subsection 3.2.1:Thread of control)

`input.clickOn(x,y)` - this call-in moves the cursor to position x,y and then clicks there. As described in section 3.2:Overview of System Architecture it uses the IGameInput interface, and so is not tied to any particular platform. In this case however, we are using X11Input and so this moves the mouse and clicks there.

`input.matchImage("picture")` - This returns the x,y coordinates of the best match of the file named "picture", or nil if no match is found. In this particular GameProfile, there is pre-made screenshot of how the mole looks called "mole.png".

Background

Here, I am interested in algorithms to implement `input.matchImage("picture")`.

Object Detection and Recognition

One of the most researched subjects in computer vision, and indeed the most fundamental one required for my project, is "object detection"; the process of detecting instances of objects of a certain class in an image (e.g. faces or cars). A solid implementation of object detection is necessary in my project; being able to recognise and distinguish objects is required for virtually all games, from the simplest online web-browser games to advanced 3D first person shooters.

My hypothesis is that different games will suit different object detection techniques. For example techniques that can be used in simple 2D games, with 2D sprites will be unsuitable for 3D games. However, if possible, I wish to create a generic technique which works well across as many games as possible. Techniques can be evaluated under the following criteria[6]:

1. **Recall**
$$\frac{Number\ of\ correct\ positives}{Total\ number\ of\ positives\ in\ dataset}$$
Recall gives us the proportion of objects that are detected.

2. **Precision**
$$\frac{Number\ of\ correct\ positives}{Number\ of\ correct\ positives\ + Number\ of\ false\ positives}$$
Precision gives us the number of false detections relative to the total number of detections made by the system.

3. **Training sample size**
The number of positives (images displaying the object) and the number of negatives (images without an object) will greatly affect the recall and precision rates. Different techniques will require different quantities of samples.

4. **Training Speed**
The speed with which a particular classifier/detector learns from the sample set.

5. **Detection Speed**
This signifies the speed with which a particular technique can find objects in a image. This statistic is vital to the project, the faster the detection speed, the more video frames it can analyse, increasing the efficacy of the system.

Template Matching

This is the simplest technique for object recognition, but one that could be potentially very useful for 2D games. The basic premise is simple, you have an image of the object (a template) you wish to find, and then you ”slide” this template over the input image, at each point calculating the accuracy of the match [7]. The method to calculate the accuracy of the match can vary, but they all involve cross-correlating the corresponding pixels.[8]

This technique can produce very accurate matches if you have a template that is virtually identical to the objects in the input image. This is true for the case of most 2D games, as the objects are always identical (being just an image, that may have been replicated multiple times and/or are moving). Although this technique is translation invariant, it is not invariant to scale and rotation, which is necessary for other games. Performance is also an issue, as you need iterate through the entire image. Lets say the image has a resolution of $X * Y$ and at each point has to evaluate the difference between each pixel of the template $(M * N)$, resulting in a complexity of $O((X - M) * (Y - N) * M * N)$ (taking into account that pixels near borders will not need to be evaluated).

There is research into increasing the performance of template matching and tackling scale/rotation invariance. A well-known approach is to evaluate an image pyramid for the template and the input image and to perform the comparison by a top-down search [9].

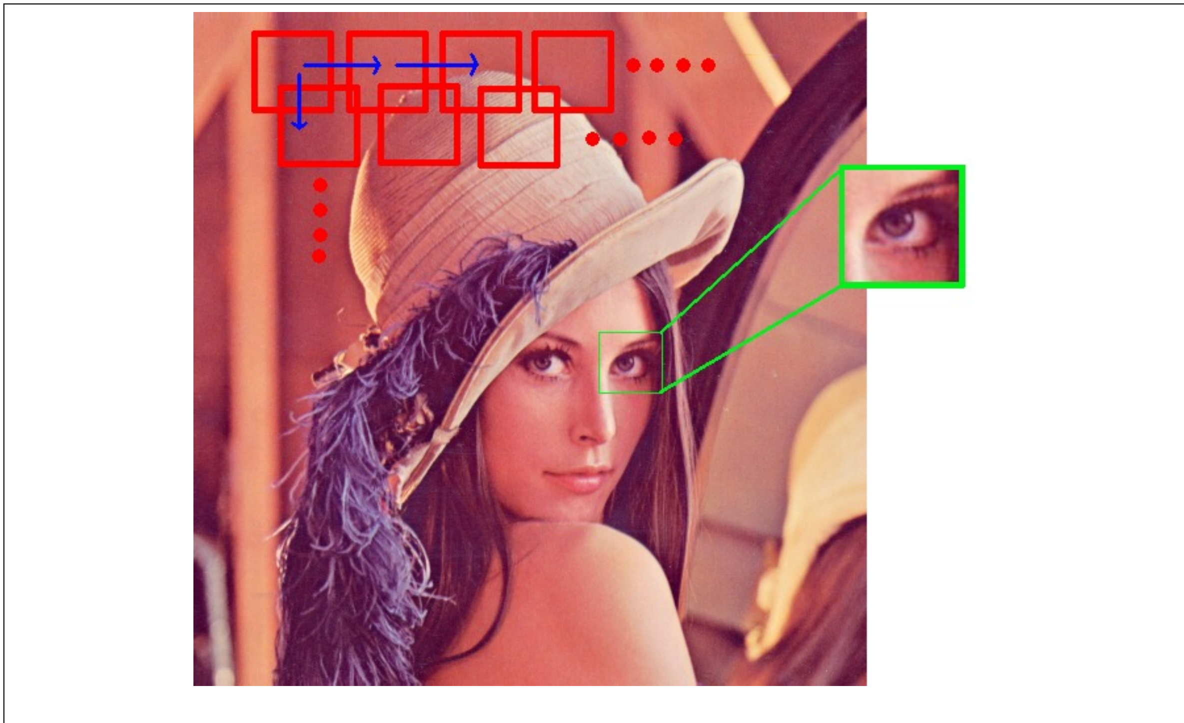

Figure 4.2: Template matching in practice - the template is swept across the image, calculating correlation at each point.

In pyramidal structures several versions of the same image at different resolution levels are available, and the solution is refined successively step by step towards the base. This is discussed in section 4.2:Image Pyramids. Other techniques concentrate on using a subset of the templates or use a coarsely spaced grid first, then iterate until a better match is found[7].

One interesting solution is the use of point correlation, where certain points are *selected* from the template[10]. These points represent the "most important" points in the template, which are found using a set of heuristics. From test results given, the authors found the accuracy of point correlation vs template matching to be virtually identical on images with little noise (in noisy images there are significant differences due to less points being used, however this does not apply to my project as games are noise-free). Varying the amount of points directly correlated with the performance boost, but they found that 50 points was sufficient to provide nearly the same accuracy as the entire template (of over 300 points). This technique can be used side-by-side with pyramidal structures, or course grids.

Rotation invariance is a harder technique to achieve with template matching. There have been various approaches taken to achieving rotation invariance such as graph matching [11], geometric hashing[12], geometric moment-based matching[13], generalised Hough transform[14] and orientation codes [15].

Template matching is a simple object detection mechanism. It only requires a single positive image, no training is needed, can have fast detection speed and in many cases, achieves excellent recall and precision rates. However it usually performs very badly in 3D

environments where an objects template can differ greatly depending on which angle you look at them from. More advanced techniques will be needed for any reasonable accuracy.

Implementation

The implementation I chose to use is the `matchTemplate` implementation from OpenCV[1]. The Open Source Computer Vision is a library of programming functions with a strong focus on real time computer vision.

This implementation does not use any of the techniques described in the background section, and does matching in the frequency domain instead of the spatial domain. As correlation is much faster to do in the frequency domain (as it is simply a multiply), Fourier transforms of both the template and the image are taken (using a discrete fast Fourier transform implementation). In order to multiply the two images in the frequency domain, they must be the same size, therefore the template is padded with black pixels (i.e. zero-padding).

This means it only requires $O(N^2 logN)$ operations for NxN images, instead of the usual $0(N^4)$ for straight-forward cross correlation.

The images are 3-channel colour images, so this template matching was done on each channel separately resulting in three 1-channel images, and then averaged together at the end into a single 1-channel image. Each pixel represents the quality of the match at that the particular point by the following correlation equation:

$$R_{ccorr}(x, y) = \sum_{x^{`}, y^{`}} [T(x^{`}, y^{`}) \cdot I(x + x^{`}, y + y^{`})]^2 \tag{4.1}$$

where R denotes the resulting image, I denotes the input image and T denotes the template.

As you can see, this is not normalised, so a perfect match will be very large, and a bad match will be small or 0. The position of best match can be found by iterating through the image, and finding the pixel with the largest intensity in the resulting 1-channel image. When an object is guaranteed to exist in the image at any point in time, then this position can be returned without any further processing. However this is not the case for every game. With Whack-a-mole, there may be no moles in view at a particular point in time. This means a certain threshold needs be applied to this point, below which it is classified as ”no-match”.

This causes problems with the previous correlation equation, as it is not normalised; larger templates will always return larger numbers. I solved this by using a normalised correlation such as:

$$R_{normed}(x, y) = \frac{R_{ccor}(x, y)}{\sqrt{\sum_{x^{`}, y^{`}} T(x^{`}, y^{`})^2 \cdot \sum_{x^{`}, y^{`}} I(x + x^{`}, y + y^{`})^2}} \tag{4.2}$$

Figure 4.3 shows the template matching in action. The right image displays the correlation

[1] http://opencv.willowgarage.com/wiki/

matrix (inverted for display purposes) - the blacker the pixel, the better the correlation. You can clearly see 2 spots of high correlation where the 2 moles have appeared.

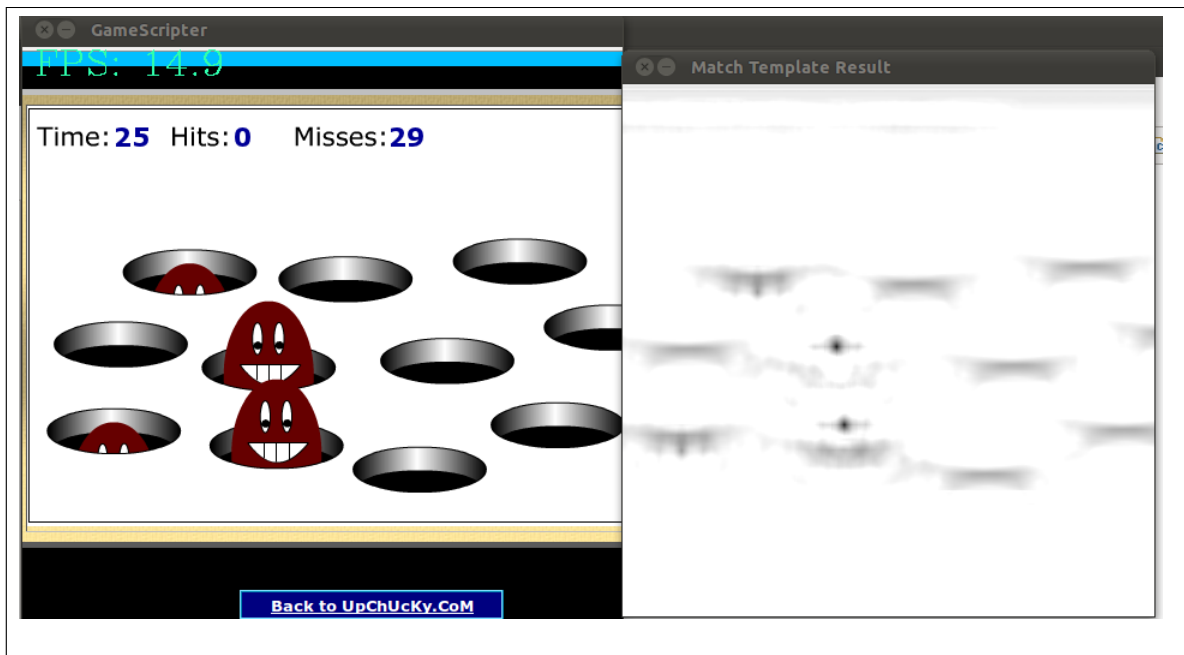

Figure 4.3: Whack-A-Mole script, with the debug view on

Results and Improvements

I tested my GameProfile on 2 whack-a-mole style games: Whack[2] and whac-a-mole[3].

Keeping with my iterative design schedule, I kept this implementation of *matchimage()* incredibly simple. It is neither scale or orientation invariant because it does not need to be, as the moles are all of the same size and always upright. Also it does not take into account multiple matches, returning just the single, highest quality match. For Whack-a-mole this is fine, as once a mole has been whacked, it will retreat back into it's hole.

Here are the results:

Game	Recall	Precision	Image Resolution	Template Size	FPS
Whack	97%	100%	500x500	74x73	10.6
whac-a-mole	90%	100%	500x500	82x70	10.1

Note, that recall & precision have been calculated in an in-game situation - if a mole pops-up and hides before it is hit, I have counted that as a positive in the dataset, but not a correct positive.

Interesting to note was that recall was not 100% like I expected it to be. In the first game, Whack, often multiple moles would pop-up at the same time. As the template matching returns only 1 result, only one of these moles would be hit in a single frame (which takes

[2]http://upchucky.com/flash-games-whack.html
[3]http://www.gamepoetry.com/blog/4k-flash-whac-a-mole-4k/

about 100ms). Towards the end of the game, the moles pop-up and in and incredible speed, where they are only out for a few 100ms, during which they may be missed be the hammer because other moles were prioritised before!

In whac-a-mole precision dropped further. As well as the reason above, this game having a large hammer as it's cursor as can be seen in figure 4.4. This often obscured the moles. The solution to this was simple - once you have whacked a mole, move the cursor out the way! This was just a single extra line in the GameProfile:

```
function callOuts.newframe()
  x, y = vision.matchImage("mole.png")
  input.clickOn(x,y)
+ input.moveCursor(0,0)
end
```

This increased precision to 98%.

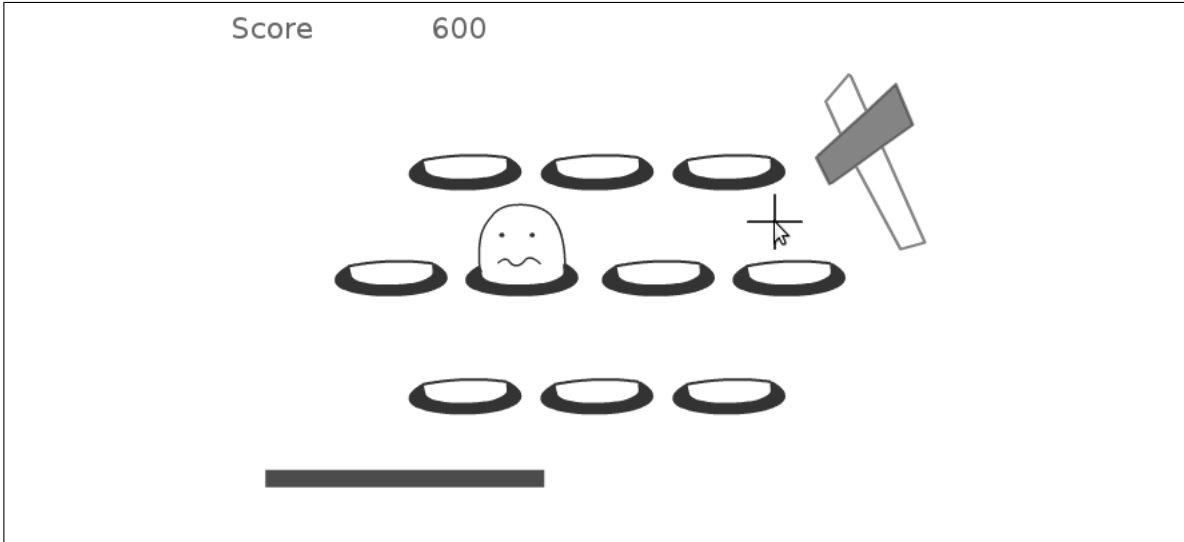

Figure 4.4: Whac-a-mole - a Whack-a-mole game with custom cursor

Performance

The performance of the template matching was disappointing. Even using the discrete Fourier Transform method, we were getting about 10 FPS for both games. It is interesting that whac-a-mole was slower by about 4.9%. The only difference between the two was a small difference in template size - about 5.6%. I investigated this further, and my results can be seen in figure 4.5.

The graph shows some peculiar behaviour. Most interesting was the sudden 100% performance increase going from 7000 (∼84x84) pixels to 8000 (∼89x89) pixels. Unfortunately I could not get to the bottom of why this happened. The OpenCV implementation of `matchTemplate` is over 400 lines long and the discrete Fourier transform implementation is over 600 lines. They both contain many low level optimisations and are designed for pure

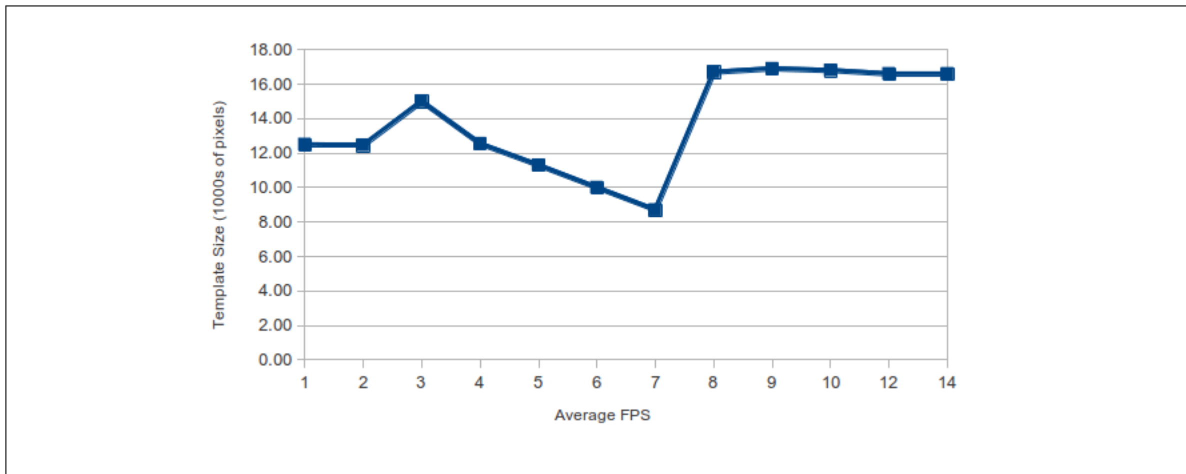

Figure 4.5: A graph showing the performance scaling as template size changes using OpenCV implementation

performance rather than legibility (no comments). For example they have hard-coded array of 1650 elements called `optimalDFTSizeTab[]` which is used in the method `getOptimalDFTSize` to return the optimal size the DFT matrix of the image. From my investigation, it seems to be due to the implementation of DFT rather than the template matching implementation.

Future Improvements

However from this investigation, I did learn many possible ways to dramatically improve the speed of template matching. Between frames, the image of the template does not change -in fact, I have optimised it so that the image is stored in memory for the entire duration, and only read once from the file - something the user does not need to worry about when using GameScript. On each call to matchTemplate(), the DFT of the template is recalculated - this is necessary. Doing this only once, storing the result and associating it with the original image should reduce computation time.

Although this will not effect this particular GameProfile, storing the DFT of the current frame can also lead to significant performance improvements. In the next section, Generic Whack-a-mole, there will be multiple template-matches per frame. As will be shown, using the naive implementation of using matchTemplate() on each template kills performance. By storing the results DFT of both the current frame and templates, template matching can be reduced to just 3 DFTs (1 per channel) and 3 matrix multiplies (1 per channel) per template, per frame.[4]

[4]This is a simplification - OpenCV actually works out an optimal "block" size and does many small block-wise matrix multiplications for correlation rather than a single large multiply for performance reasons

4.2 Generic Whack-a-mole

Introduction

This builds upon the previous game. The idea is to have a single script that will work with all whack-a-mole games, regardless of how the "moles" look or hole placement (figure 4.6). This means there should be no need for having a pre-taken image(s) of the mole(s) like in the previous GameProfile.

Figure 4.6: GraveDigger - This game looks very different from the others, but follows the same whack-a-mole style template. The moles are now creatures, and there's more than 1 type.

Desired GameProfile

The basic idea behind this GameProfile is that instead of having pre-made templates, the templates can be made on the fly by observing the player play. Here is a simple example of such a script, which collects images that the user clicks on, and then when the user clicks a predefined key (in this case "s"), it starts playing the game in a similar manner to the previous script.

```
images = {}
startPlaying = false

function callOuts.newframe(frame)
    if startPlaying == true then
        for i,v in ipairs(images) do
            x, y = vision.matchImage(v)
            input.clickOn(x,y)
        end
```

```
    end
end

function callOuts.buttonRelease(x, y)
    images[#images+1] = vision.getImage(x-30, y-30, 60, 60);
end

function callOuts.keyRelease(key)
    if key == 's' then
        startPlaying = true
    end
end
```

Here we introduce 2 new call-outs and 1 input call-in:

`callOuts.buttonRelease(x, y, button)` gets called, every-time a button is released (it's counterpart is `buttonPress(x, y, button)`). In this case, we are on a PC, so this is called every time a button on the mouse is released. It passes in the coordinates where the button was released and which button it was. Like **all** functions defined in GameProfile, these are all optional arguments, so if the user does not care about which button or coordinates, `buttonRelease()` can be used instead.

`callOuts.keyRelease(key)` gets called every-time a key is released. In this case it is a keyboard key, and a string of the key pressed is passed in - e.g. `'A'` or `'Up'`

`vision.getImage(x, y, width, height)` this gets a subset of the image, defined by the 4 arguments. It returns a "handle" to the image, which can be used in any GameScript function that accepts images. For example, the `matchImage()` has been extended to accept these handles as well as strings.

Background

Image Pyramids

An image pyramid [16] is a data structure was originally designed to support efficient scaled convolution through reduced image representation, but has found applications in a wide variety of vision applications. It consists of a sequence of copies of the original image that are successively down-sampled (usually in regular steps).

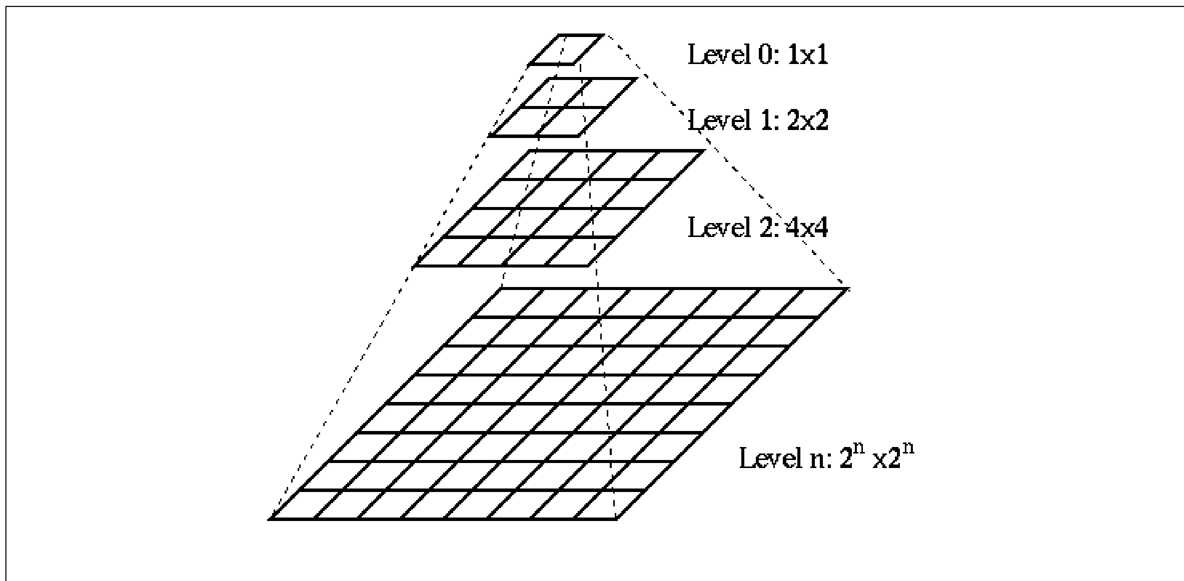

Figure 4.7: A simple Image pyramid

There are two types of image pyramids - the Gaussian pyramid and the Laplacian pyramid. The Gaussian pyramid is used to repeatedly filter and sub-sample the images to generate the sequence of reduced resolution images G_0, G_1, G_2 etc... The Laplacian pyramid is required when we want to reconstruct an up-sampled image from an image lower in the pyramid, e.g. reconstruct G_1 from G_2.

Gaussian Pyramids G_0 is the original image, at the bottom of the image pyramid. To produce level $G_i + 1$ from G_i, we first convolve G_i with a Gaussian kernel and then remove every even-numbered row and column thereby producing an image 1/4 of the area.

$$G_l(i,j) \sum_m \sum_n w(m,n) G_{l-1}(2i+m, 2j+n) \tag{4.3}$$

$w(m,n)$ is called the "generating kernel" and is usually Gaussian. We shall call this process as a REDUCE operation:

$$G_l = REDUCE[G_{l-1}] \tag{4.4}$$

The reverse process of REDUCE operation doubles the the size of the image in each dimension, with the new (even) rows filled with 0s. Afterwards, a convolution is performed with the filter derived from $w(m,n)$ (it's twice the dimensions, and normalised to 4) to fill in the values of the missing pixels. We can call this an EXPAND operation. Let G_l, k be the image obtained by expanding G_l k times. Then $G_l, k = EXPAND$[G $G_{l,k-1}$]

$$G_{l,k}(i,j) = 4 \sum_m \sum_n G_{l,k-1}(\frac{2i+m}{2}, \frac{2j+n}{2}) \tag{4.5}$$

Of course this process is just an approximation; it does not result in the original image as information is lost in the REDUCE stage. In order to restore the original image, we would require access to the information that was discarded by down-sampling. This is what forms the data of the Laplacian pyramid.

Laplacian Pyramids The most important property of the Laplacian pyramid is that it is a complete image representation: the steps used to construct the pyramid may be reversed to recover the original image exactly. The ith layer of the Laplacian pyramid is defined by:

$$L_l = G_l - EXPAND[G_{l+1}] \tag{4.6}$$

Another more intuitive way of defining the Laplacian pyramid (where the filter is a 5-by-5 Gaussian) is defined by the relation[17]:

$$L_i = G_i - UPSIZE(G_{i+1}) \otimes \mathcal{G}_{5x5} \tag{4.7}$$

Here the UPSIZE operator up-sizes the mapping of each pixel in location (x,y) in the original image, to pixel(2x+1,2y+1) in the destination image.
$\otimes$ signifies convolution and $\mathcal{G}_{5x5}$ is the 5-by-5 Gaussian kernel.

Uses Usually transforming an image from one representation to another is done for two reasons[16]:

- Transformation may isolate critical component of the image pattern so that they are more directly accessible to analysis
- Transformation may place the data in a more compact form so that they can be stored more efficiently

Laplacian pyramids serves both these objectives. Construction tends to enhance image features such as edges. Pyramid representation can also compress the data, however I am not interesting in this particular case.

Pyramid methods may be applied to achieve some level of template-matching scale independence. The template is convolved with each level of the image pyramid. As all levels of the pyramid combined contain just one third more nodes than there are pixels in the original

image - the cost of searching for a pattern at many scales is just one third more than that of searching the original image alone. This is much more efficient than scaling the template itself and searching the original image each time!

If the relative scale is too large between images, variants exist which can easily be defined with square root or smaller steps. However this of-course come at the cost of computational complexity.

Pyramid methods are also used in the estimation of properties within local image regions - for example image segmentation. These algorithms use fast initial segmentation which is done on the low-resolution images in the pyramid, then refined and further differentiated level by level.

Another class of analysis operations concerns fast course-fine search techniques. As was shown in section 4.1:Results and Improvements, template-matching is very slow. Rather than attempting to convolve the full template with the image, both the template image and the target image are made into Laplacian pyramids. The search can begin by convolving the low-resolution images in the pyramid, where rough estimates can be found. Higher-resolution copies of the pattern and image are used to provide more accurate estimates through subsequent convolutions. Substantial computational savings are achieved as search neighbourhoods are restricted to points identified at the coarser resolution.

The last use for Laplacian pyramids are image enhancement techniques such as noise reduction or sharpening[16]. The images from games are perfect and noise-free so this is of not much interest for GameScripter.

Implementation

The implementation of `vision.getImage()` was simple - all it does, is extract a region of interest from the current frame and stores it in the C++ core. A *handle* is passed back to the GameProfile, that can be used as a parameter whenever an image is expected. This "handle" is really a Lua lightUserData structure that contains a raw pointer to the C++ object that encapsulates the image.

As will be discussed in the evaluation (section 5.2), this in hindsight, was a poor choice as all the C++ core now has the responsibility of memory management, and does not know when it can be reclaimed.

Threading Issues

I remind the reader that IGameInputMonitor, which handles all the call-outs such as `callOuts.buttonRelease()`, runs in a separate thread (see section 3.2). This means that all call-outs need to be thread safe - including `vision.getImage()`.

The time it takes to query the X server for an image is about 5ms - this is what limits the FPS of the program, even when using an empty GameScript (see figure 4.8 with 0 templates). If during these 5ms, a call-in (such as `vision.getImage()`) was made, it was possible to crash

the program. Therefore I had to put mutexes and locks in all appropriate places for call-ins[5]

GPU implementation of template matching

The experimental version of OpenCV provides an early implementation of GPU accelerated template matching. It uses the same algorithm outlined as the CPU version, but it runs on a CUDA enabled, Nvidia GPU. As I already owned a fairly top of the range card (the GTX 570) and having never used it for any kind of kind of general-purpose computing on graphics processing units (GPGPU) tasks, I was interested in the performance gains I could achieve.

Implementation was fairly simple, as the interface was similar to the CPU version. As well as the template matching itself, I also ported all the other helper functions to the GPU - such as converting the colour space and finding the point of maximum correlation in the resulting single-channel matrix. I wanted to make the switching between the two easy for the user, so I associated a key that would switch between CPU and GPU acceleration at runtime (by default Alt-G).

The first implementation was simple - it followed the same principle as the CPU version. Performance increased massively, but not as much as I expected. When I profiled the code, I realised that act of moving the template and current frame onto the GPU memory actually took up over 60% of the processing time. Although I do not have any benchmarks for the original naive GPU implementation, performance improved significantly, especially with increasing amounts of templates.

Figure 4.8 shows the performance difference between the GPU and CPU versions. Both techniques have an approximately logarithmic decrease in performance with every new template added. This is to be expected as each call to match template will take the same amount of time, regardless of the number of templates. Through-out, GPU remains a constant 8.5x faster. This is an impressive performance boost, and makes a big difference as it it keeps fps in double figures below 12 templates, during which time the CPU struggles to get even 1 fps.

Results and Improvements

The script is successful in it's task. It was able to play both Whack-a-Mole games from the previous section, as well as many others I tested, including the GraveDigger game (figure 4.6 in the introduction). However there are a number of issues:

Performance Even with the massive performance boost due to GPU acceleration, FPS went down to single digits after only 12 templates. This was satisfactory for all games I played, as even the most complicated game, GraveDigger, had only 3 mole-types, and only 1 image was needed per mole. As I was satisfied with the results from the GPU accelerated template matching, I moved on to other areas of the project without implementing a version with image pyramids (although pyramids are used in other areas of the project, like optical flow section 4.6).

[5]For those interested, this was eased by the use of the boost threading library - `http://www.boost.org/doc/libs/1_46_1/doc/html/thread.html`

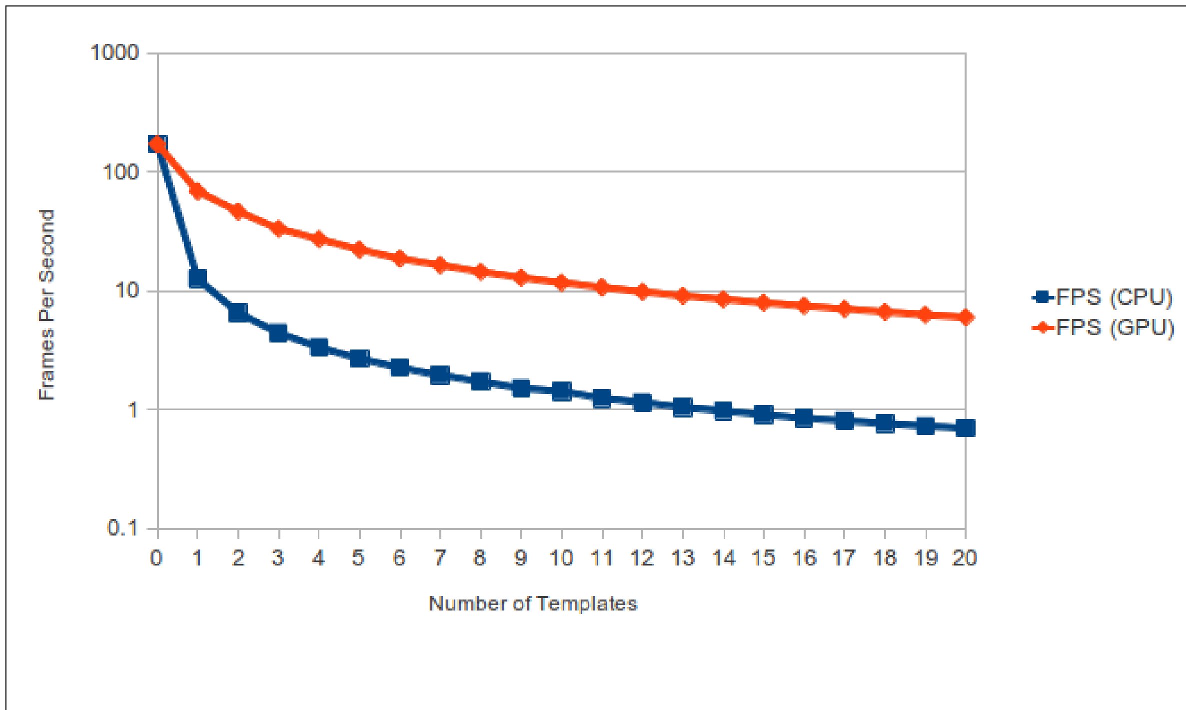

Figure 4.8: Graph comparing performance of CPU template matching vs GPU template matching. It was taken with a 500x500 image, and templates of size 60x60

However I believe I could achieve a order of magnitude better performance, using the techniques already discussed:

- Not recalculating the DFT of the templates each frame.
- Not recalculating the frame DFT between template matches in the same frame
- Using the image Pyramids discussed above

Image Sizes As you can see from the GameProfile code, the size of the image taken has been hard-coded (in this case it is 60x60). This means, the user needs to guess an appropriate size for template - something that should be avoided. Getting the template size wrong has two consequences:

Image too small If the image is too small, template match precision can take a hit as the template may not have enough information to uniquely identify the object

Image too large If the image is too large, template match recall can take a hit, because the template may contain too much background. This is not a problem in the whack-a-mole games describe in section 4.1:Results and Improvements as the background is a constant colour, but in GraveDigger, each hole looks different.

User manually stops learning period At the moment, the user has to press S to stop the learning phase. It would be ideal to have the system itself infer when it is finished learning, and take-over automatically. I have implemented a prototype of this in GameScript itself, although it is anything but user-friendly. In principle it is very simple - it runs the template matching in parallel to the learning phase and whenever the user whacks a mole, it checks if it would have predicted a match in the close vicinity too (I used a 20 pixel radius). Once it has predicted x clicks in a row (I chose 5), it takes over playing the game.

This works reasonably well, the major problem being template matching performance (which as discussed, can be solved). However I do not believe this type of learning can be placed into the core C++ because it is not generic enough. It is easy to define when to stop learning for Whack-A-Mole type games, however how do we extend this to the games discussed later such as Breakout, Tetris or advanced 3D games?

User Friendliness In this script there are two statements I dislike and think could be made more user friendly - the for loop and the images array.

- In the for loop: `for i,v in ipairs(images) do` the user is required to know the `ipairs` iterator (which iterates over arrays) and that it returns 2 variables - the index (`i`) and the value (`v`).
- The `images` is an array, and requires the user to know the length function `#` to append values to the end.

Fortunately both are easily solved as Lua is a fully featured programming language. We can create a new Lua library called `GameScripterLibrary`, and there we can create new functions and objects that can be used in GameProfiles. I have created simple containers and iterators so that the users can use them as follows:

```
for image in images:allImages() do
   ....

...
images:add(image)
...
```

Rather than requiring the user to import this file - i.e. `loadlib(GameScripterLibrary)` I have chosen to load the library automatically for them from within GameScripter.

The containers could have been implemented in the C++ core, however I felt it was a more elegant (and a lot less coding!) to keep the C++ core cohesive and deal with performance-intensive vision tasks only, while the GameScripterLibrary added user friendliness.

4.3 Breakout

Introduction

Breakout is originally an arcade game from the the late 1970s. In the game, a layer of bricks lines the top half of the screen. A ball travels across the screen, bouncing off the top and side walls of the screen. When a brick is hit, the ball bounces away and the brick is destroyed. The player loses a turn when the ball touches the bottom of the screen. To prevent this from happening, the player has a movable paddle (controlled by the mouse) to bounce the ball upward, keeping it in play.

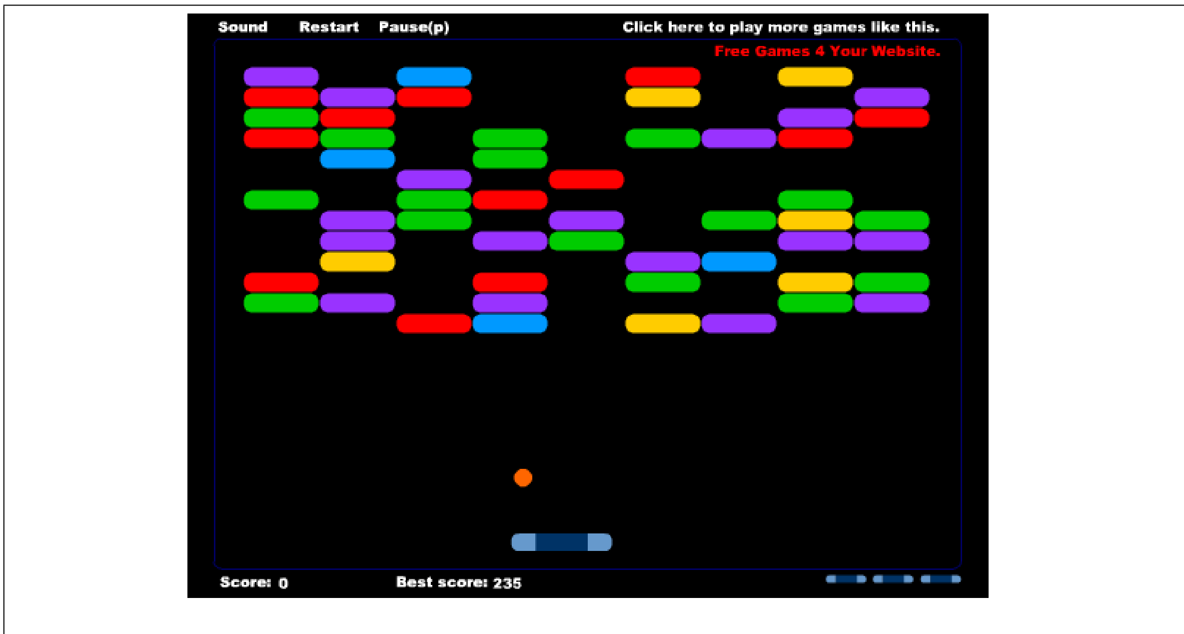

Figure 4.9: An breakout clone. The player controls the paddle at the bottom of the screen using the mouse

Desired GameProfile

A game-specific GameScript can be written using the already discussed functionality, by using template matching to find the ball position and moving the mouse (which controls the paddle movement) to x coordinate to where the ball is found. We also introduce the `input.moveCursor` call-in.

```
function callOuts.newframe()
  x, y = vision.matchImage("ball.png")
  input.moveCursor(x,nil)
end
```

`input.moveCursor(x,y)` - moves the cursor. If x is nil, moves just in the y direction and vice-versa.

Results and Improvements

In the first game I tested on, Breakout by 2DPlay[6] I managed to get a 95% hit ratio. The times the paddle did not manage to rebound the ball was because the ball speed was too fast; the program ran at 19 FPS which at times was not fast enough to keep up with the ball. The simple solution was to switch on GPU accelerated template matching, which increased FPS to over 120. However a more interesting solution was to change the script itself and add some very simple ball trajectory estimation:

```
lastXPos = 0    //global variable store the previous positions

function callOuts.newframe()
  x, y = vision.matchImage("ball.png")
  input.moveCursor(x + (x - lastXPos), nil)
  lastXPos = x
end
```

The ball trajectory is a simple line, so assuming the GameScripter runs at a constant fps (which it does in this case, as the computation per frame is constant), one can calculate where the ball be in the next frame with:

```
current x position + distance moved in x direction in 1 frame
```

This starts to show the power of GameScript being a fully fledged programming language, rather than a simple Domain Specific Language and results in a GameProfile which can play a perfect game of breakout!

The second game I test was Bounce Back[7]. Here I hit a snag - the GameProfile was *too* good. The mechanics of the game are different to the one in the first link, as the angle the ball bounces back from the paddle is not based on the angle of incidence, but actually on where ball hits the paddle. Both the simple GameProfile as well as the predictive one often get to a state after a few bounces where the ball *always* hits the middle of the paddle. This results in the ball just going straight vertically up and down ad infinitum. (see figure 4.10)

The simplest way to solve this would be to add some randomness to the script. The following example will move the mouse to a random position within 10 pixels of the ball.

```
input.moveCursor(x + math.random(-10,10), nil)
```

A more interesting solution would be to actually use this game mechanic to your advantage. One could find the position of any block and the paddle (using `matchImage()` again) and then calculate the position on the paddle where the ball should hit so that it rebounds with the correct angle so that it flies perfectly towards the target block. This is perfectly possible using the current framework, although FPS will take a hit due to the large number of calls to `matchImage()`.

[6]http://www.play.vg/games/10-Breakout.html
[7]http://www.agame.com/game/bounce-back.html

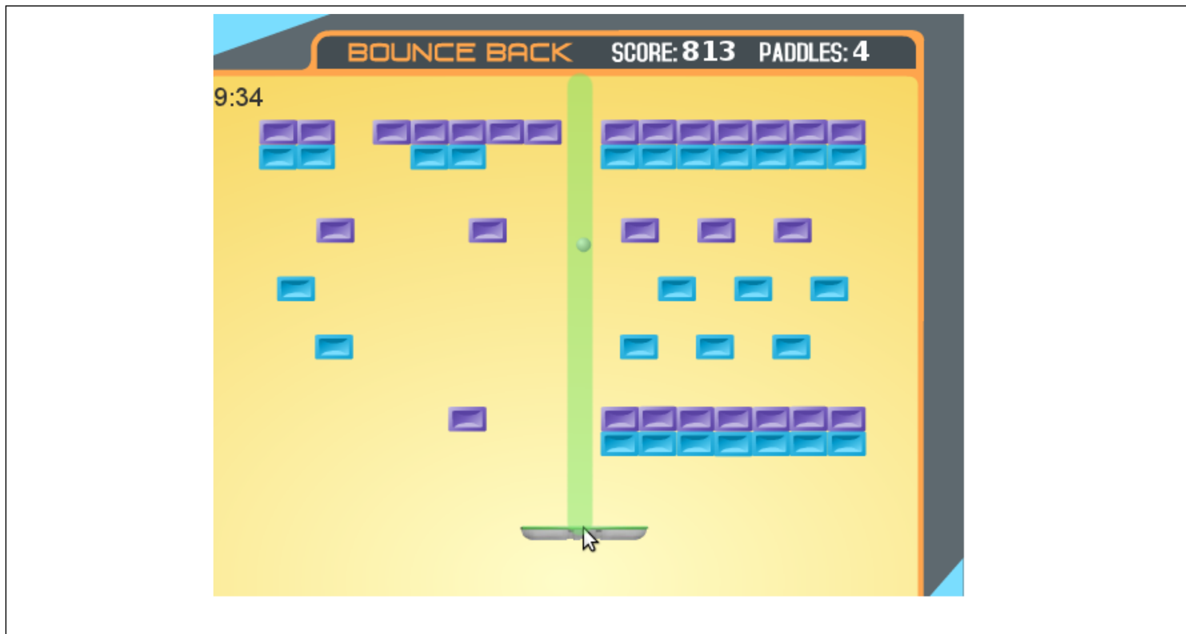

Figure 4.10: Bounce Back - here the ball stays never deviates outside the highlighted corridor

Further Improvements Although the GameProfile in this form is able to play the game, it does not take full advantage of all the features in the game. When destroyed, some blocks drop a power-up (or power-down) which usually take the form of a color-coded pill shaped object. If the paddle collects one of these power-ups, it will affect it in some way. Examples include, a larger/smaller paddle, multiple balls or guns which can shoot down blocks. Using the current framework with the simple `matchImage()` function and the powerful programming ability of GameScript allows us to create a GameProfile that uses these most of these features.

To tackle the case of multiple balls, I created the `matchMultipeImages()` function to return multiple matches. It also takes as an (optional) parameter "`numberOfMatches`" which causes the function to return only the best `numberOfMatches` number of objects. The results are returned as a simple array, each containing a table containing information about the object. The table structure remains constant between *all* functions that return multiple results, to keep the interface consistent (see Appendix D:GameScript Standard Object Table). A simple example is given below:

```
function callOuts.newframe()
  matches = vision.matchTemplate("ball.png")
  for i, obj in ipairs(matches) do
    print("Found a ball at: (" .. obj.x .. "," .. obj.y ..")" )
    print("Match Accuracy: " obj.matchAccuracy)
    print("Object size: (".. obj.width .. "," .. obj.height ..")" )
  end
  lowestObject = lowestYcoordinate(matches)
  input.moveCursor(lowestObject.x, nil)
  lastXPos = x
```

end

This handles multiple balls perfectly, by always tracking the lowest ball in the game. It also demonstrates that the print command works from within GameProfiles if the user wishes to print out information.

The `MatchAccuracy` field is always normalised to be between 0 and 1 for all matching types (such as histogram/contour matching in later GameProfiles).

4.4 Generic Breakout

Introduction

This is an improvement on the previous script; a single script that is be able to handle any breakout game.

Desired GameProfile

The common factor in every breakout game is that there is a ball that is moving. By segregating the objects into background (stationary) and foreground (moving) objects, we should be able to pin-point the ball position, regardless of how the ball looks. The desired GameProfile looks as follows:

```
function callOuts.newframe(frame)
  objects = vision.getForgroundObjects()
  if(isNotEmpty(objects)) then
    smallestObj = smallest(objects)
    input.moveCursor(smallestObj.x, smallestObj.y)
  end
end
```

This introduces one new call-in: `vision.getForgroundObjects()`

`vision.getForgroundObjects(numberOfObjects, minArea)` Returns an array of objects found to be moving. Has 2 optional arguments: the first specifies the maximum number of objects to return (ordered by size). The second specifies the minimum area needed for the object to be considered.

`smallest(table)` and `isNotEmpty(table)` are helper functions defined in the GameScripterLibrary.

Background

Background Subtraction

Background subtraction is a way for detecting and isolating moving objects from the rest of the image. Implementing a good background subtraction algorithm into my framework will be very useful; instead of using the entire game ”screen-shot” to run object detection, object detection can be confined to only moving objects. It can also provide an easy way to collect data to train classifiers on. In some cases, it also provides the basis of object tracking, which is another feature that could be useful.

In order to perform background subtraction, you first have to ”learn” a model of the background. Once learnt, this model can be compared against the current frame, and so all known background parts can be subtracted away. Everything that is left is a foreground object - usually signifying a moving object.

The simplest background subtraction methods involve just subtracting one frame from another and then any difference past a certain threshold is classified as foreground.[17].

$$|frame_i - frame_{i-1}| > threshold \tag{4.8}$$

Usually this is not enough as the background may be moving too (e.g. trees swaying in the wind) and illumination changes may distort the results. This requires more advanced techniques where we keep statistics about means and average differences of pixels in the scene. This is usually done for about 30 to 1000 frames, using only a few frames a second. More advanced methods are often based on fitting a Gaussian distribution (μ, σ) over the image histogram.[18] This gives use the background probability density function (PDF) which is update using a running average:

$$\mu_{i+1} = \alpha F_i + (1 - \alpha)\mu_i \tag{4.9}$$

$$\sigma^2_{i+1} = \alpha(F_i - \mu_i)^2 + (1 - \alpha)\sigma^2_i \tag{4.10}$$

Therefore to test if a pixel is in a background or not:

$$|F - \alpha| > threshold \tag{4.11}$$

where threshold is usually a multiple of the standard deviation ($k\sigma$)

The background model does not need to be constant, it should be kept regularly updated so as to adapt to varying luminance conditions and geometry settings. This is a valid need for 3D games with advanced lighting and features such as weather, but for 2D games will most probably be unnecessary.

Massimo Piccardi in his 2004 *"Background Subtraction Techniques"* [18] outlines the following most popular background subtraction techniques:

- Running Gaussian average
- Temporal median filter
- Mixture of Gaussians
- Kernel density estimation
- Sequential kernel density approximation
- Co-occurrence of image variation
- Eigenbackgrounds

These are the conclusions that were found from his results:

- *Gaussian averaging* and *median filters* offered high frame rates, with little memory requirements, at the expense of model accuracy.
- *Kernel density estimation* and *mixture of Gaussians* had high memory requirements; in the order of at least 100 frames, but much better model accuracy. *Sequential kernel density* approximation was nearly as accurate, but had lower time complexity and memory requirements by order of magnitude.
- *Co-occurrence of image variation* offered good accuracy and memory complexity, while explicitly addressing spatial correlation.

Ideally it would be useful to have background subtraction which allows arbitrary camera motion. This would be incredibly useful for 3D games, where the player is moving around, within a 3D scene. By using techniques based on ego-motion compensation and background estimation [20] one can achieve accurate foreground segmentation with a moving video, however this is probably out of the scope of the project.

Connected Component labelling

Background subtraction results in a similar image to the one given in figure 4.12. This is usually a binary image; the white areas represent foreground objects and the black represents the background. In it's simplest form, connected component labelling (also often known in computer vision as *blob extraction*) is a way of grouping together pixels into components based on pixel connectivity and pixel proximity - i.e. all pixels in a a connected component share similar intensity values and there exists a path of adjacent pixels between every pixel in a component. The resulting images from background subtraction are binary, so the measure of pixel similarity is easily solved - white pixels are similar to other white pixels. All other combinations are not similar.

A simple connected component labelling algorithm is defined below[21]. It works well on binary images (for each pixel, it's intensity value, $V = \{1\}$ or $V = \{0\}$) and can be used with both 4-connectivity or 8-connectivity. The connectivity level defines the neighbourhood that is taken into account when defining if pixels are connected or not:

For pixel p with coordinates (x,y), the set of pixels defining the 4-connectivity neighbourhood is given by:

$$N_4(p) = \{(x+1,y),(x-1,y),(x,y+1),(x,y-1)\} \tag{4.12}$$

The 8-connectivity neighbourhood includes the diagonals:

$$N_8(p) = N_4 \cup \{(x+1,y+1),(x+1,y-1),(x-1,y+1),(x-1,y-1)\} \tag{4.13}$$

The connected components labelling operator scans the image by iterating along a row until it comes to a pixel p with $V = \{1\}$. It then examines the four neighbours of p which have already been encountered in the scan (i.e. the neighbour to the left of p, above it, and the two upper diagonal terms). Based on this information, the labelling of p occurs as follows:

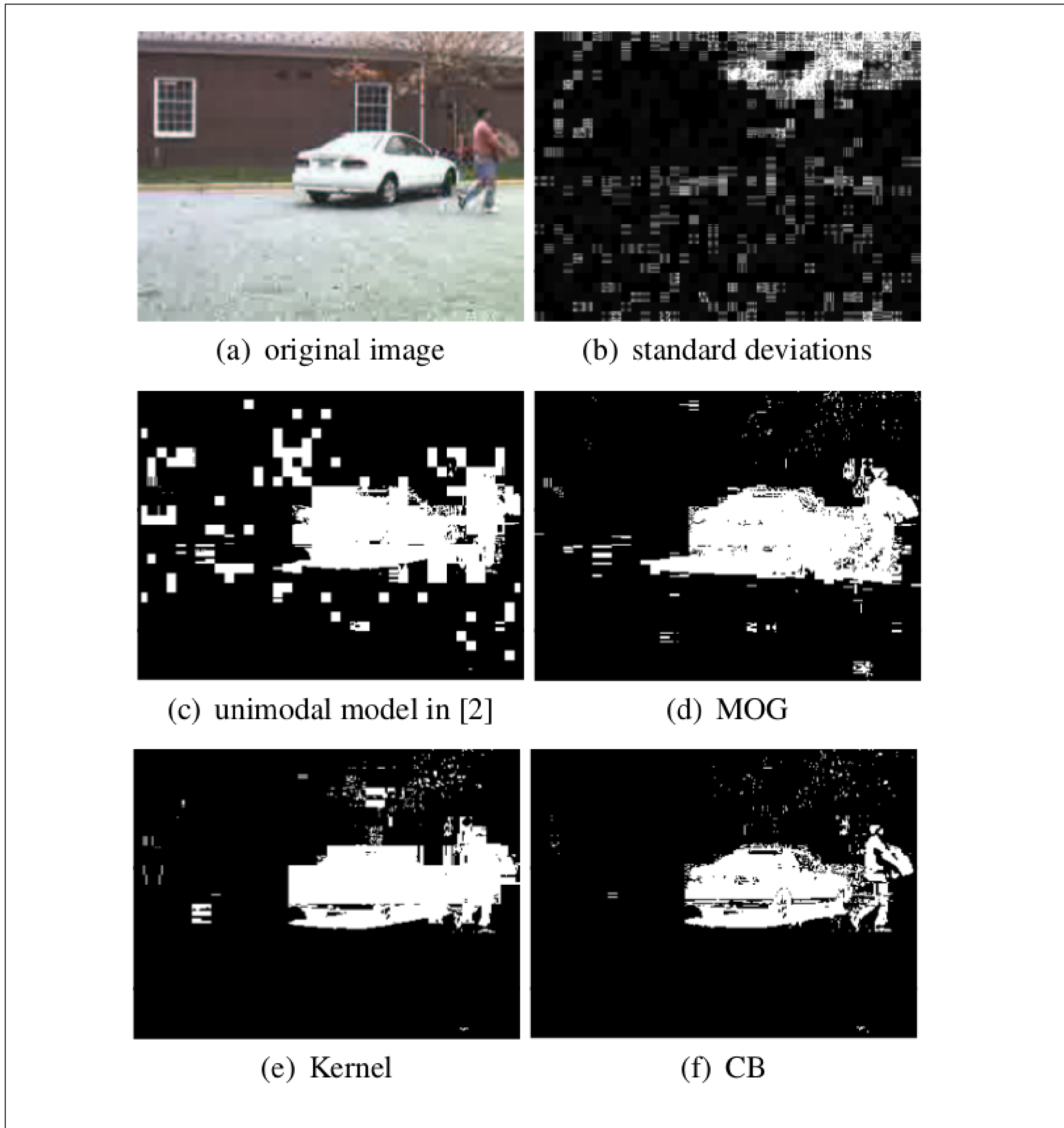

Figure 4.11: Examples of the different techniques above. The bottom right is the code-book method, which shall be discussed in the implementation. Picture taken from [19]

- If all four neighbours have $V = \{0\}$ - assign a new label to p
- If only one neighbour has $V = \{1\}$ - assign the neighbour's label to p
- If more than one of the neighbours have $V = \{1\}$ - assign one of the labels to p and make a note of the equivalences.

After completing the scan, the equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class.

Contour Extraction

Another way of extracting connected components from images is to trace the outline of objects to create contours. A popular method for contour retrieval is outlined by Suzuki[22]. The principle is simple and is similar to the component labelling algorithm listed above. The main difference is that it is a border following algorithm - the iteration doesn't expand "inwards" into the the component itself. There is also a labelling operator, except instead of labelling components, it labels borders.

The contour retrieval technique used by Suzuki results in a contour that has a point for every border pixel. This is very inefficient and often results in contours much bigger than they need to be. For example, a simple square only needs 4 contours to be represented perfectly, but with the previous method results there will be as many points as perimeter pixels.

Therefore there is a need to remove unnecessary points from the contour. Simple algorithms include compressing horizontal, vertical and diagonal segments, leaving only their

Figure 4.12: Background Subtraction - The woman is classified as foreground and so shows up white in the resulting image

ending points. There also exist some approximation algorithms such as Freeman Chain Codes [23] and Teh-Chin chain approximation[24].

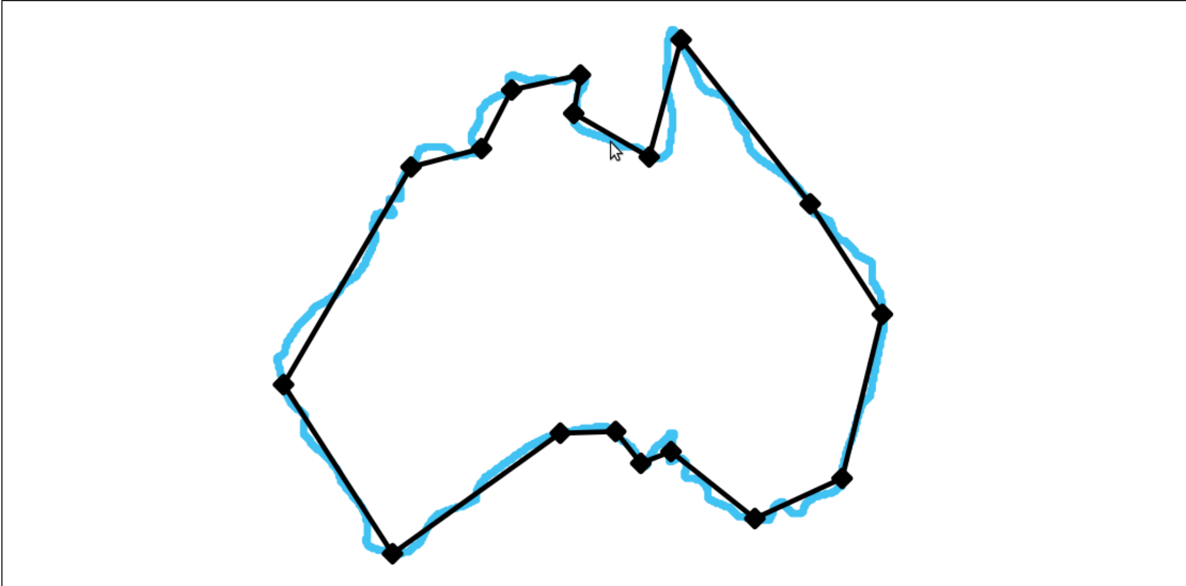

Figure 4.13: An example of Polygon approximation using Teh-Chin. The number of vertices has been reduced to only 18 from hundreds, while keeping error low

The Teh-Chin chain approximation algorithm guarantees approximation within a deviation threshold. It's approach is to use a monotonically increasing function of chord and arc length to form the initial set of approximation points. This is followed by a merging of those points. An example is shown in figure 4.13.

Freeman Chain Codes are interesting because they are a representation of the shape in terms of a sequence of lines where, each line is of a fixed length and in one of 8 directions (i.e: $0°$,$45°$,$90°$..). Each step can be represented by a number 1-7, resulting in polygon representations that are compact and easy to compare - e.g. $\{1,6,7,3,4,4,4\}$. However, Freeman approximation often suffers from quantization errors, missed points and redundant points[25]. (see Figure 4.26:Contour Matching with Moments for an example).

Implementation

Frame Subtraction

During implementation, I tested 3 background subtraction techniques. The first was simple frame-to-frame subtraction . For the purpose of tracking the ball in Breakout, this methodology sometimes actually produced better results than the other 2, much more computationally expensive & advanced techniques. This is because the background in Breakout is completely static, the usually the only thing that changes between frames is position of the ball and the position of the paddle. For the single frame after a block is destroyed, there will be also a difference. However using the assumption that the ball is smaller than blocks (which is true for all Breakout games I encountered), it was easy to find the ball.

There were however 2 disadvantages. Because of the computational simplicity of frame subtraction, the program was able to run much faster than the X server refreshed (the refresh rate was 60Hz, while my program could run at over 100fps). This meant that sometimes, frames were identical - subtraction would result in an empty matrix. This was easily solved by setting the targetFPS variable to something reasonable like 30 fps.

The other problem is that all this method does, is indicate regions of motion. If the ball in the first frame overlaps with the ball in the second frame, we actually highlight the sides of the ball (figure 4.14). This is because the pixels in the middle of the ball have not changed between frames. This is especially true for paddle, which always resulted in 2 disjoint areas of movement.

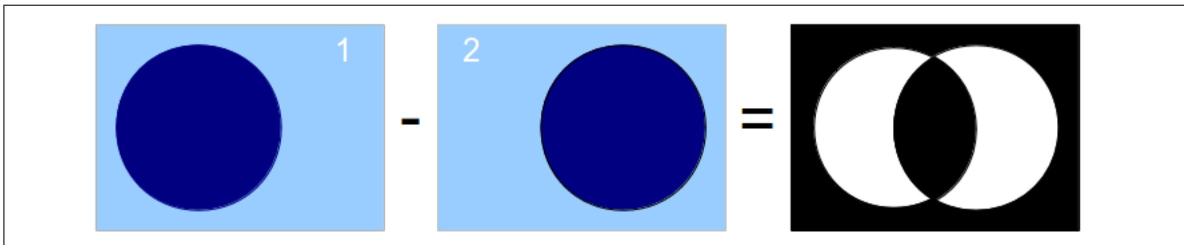

Figure 4.14: Background Subtraction using frame subtraction. Here the ball in frame 1 intersects, with the ball in frame 2. White signifies areas of difference

On the other hand, if the frame rate was too low or the ball moved too fast, the ball in the 1st frame did not overlap with the ball in the 2nd frame. This meant we actually got *2* balls in the image (figure 4.15). Here is no easy way to distinguish between the two - we do not know which area of movement corresponds to the balls current position as we have no background model.

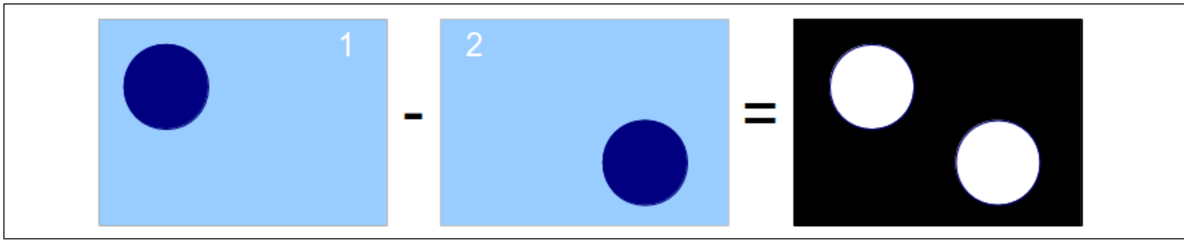

Figure 4.15: Background Subtraction using frame subtraction. Here the ball in frame 1 does not intersect, with the ball in frame 2. White signifies areas of difference

Mixture of Gaussians

The major problem was that frame subtraction does not generalise to 3D games. Like described in the background section, for effective background models in an environment where there is periodic movement and changing lighting conditions we need to learn a background model. I therefore started testing the generalised mixture of Gaussians (MOG) implementation provided by the OpenCV library.

Mixture of K Gaussians $(\alpha_i, \sigma_i, \omega_i)$ is a advancement on the running Gaussian average described in the background Section[18] to be able to cope with multi-modal backgrounds. The number of modes is arbitrarily pre-defined (I used 5). Every frame, the weight ω_i of each Gaussian is updated - the Gaussians that match the current value (e.g. have distance < 2.5 σ_i - i.e. the 99% confidence interval) have their α_i, σ_i updated by the running average.

This method actually models both foreground and the background. To be able to distinguish between which distributions only model the background, all the distributions are ranked according to $\frac{\omega_i}{\sigma_i}$ and the first X are chosen to represent background. Another alternative, is to use all distributions with a ratio above some threshold (this is what I used, with a ratio of 0.7).

With this I got some satisfactory results for breakout, and 2D games in general. As will be discussed in section 4.6:First Person Shooter, this also works reasonably well in 3D games. One disadvantage with this technique is that background with fast variations cannot be accurately modelled with just a few Gaussians. This causes problems for sensitive detection which as will be discussed later, which inspired me to write my own implementation, based on the paper "Background Modelling and Subtraction by Codebook Construction"[19].

I also felt that using a continuous distribution to model 2D games, where pixels often differ by large steps is not optimal. For example in breakout, the background is a uniform blue and the ball silver - the change between blue and silver is instantaneous between pixels. This is not necessarily true in the real world (and 3D games), where lighting creates many gradients and shades, and often there is a smoother between background and foreground (due to camera blurriness in real-life or anti-aliasing in games).

In interests of coherence, I shall give an overview of my implementation here.

Code Book Background Subtraction

CB adopts a quantization/clustering technique to encode and construct the background a pixel by pixel basis. Each pixel has 3 sets of *codewords*, which can be though of as a "box" with 2 thresholds: a minimum value *min* and maximum value *max* (see figure 4.16). Each set is associated with a single channel - I chose to use RGB as this removes the need to convert the images to a different representation. Together they comprise of a *code book* for that pixel. This is similar to the MOG implementation above, except that has Gaussian distributions instead of boxes.

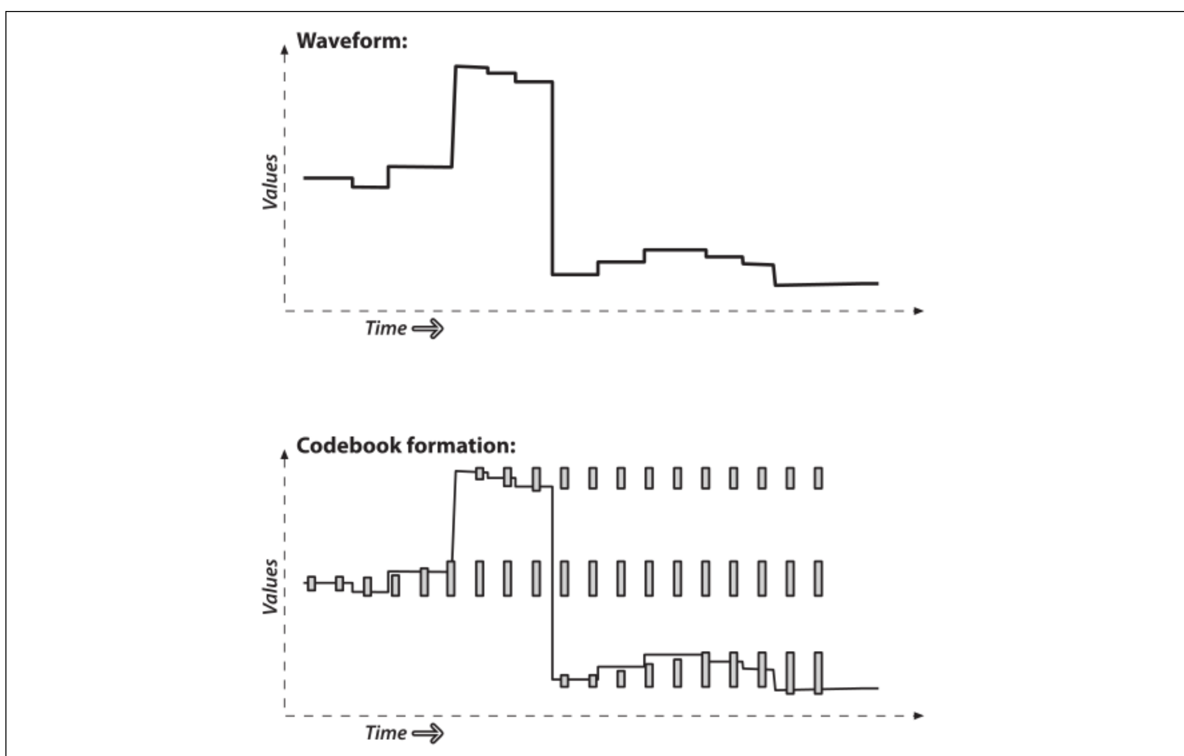

Figure 4.16: Codebook formation. The codewords expand to cover nearby values or new codewords are spawned if the there are none nearby. Taken from [17]

Background Subtraction Once trained, background subtraction with CB this is very simple and done on a pixel-by-pixel basis. For each channel we look at it's value and try to find a codeword that fits the following rule ({*min,max*}*Mod* are arbitrary thresholds):

```
(min - minMod) < value < (max + maxMod)
```

If such a codeword exists for *all* 3 channels, then this pixel is accepted as background, otherwise it is foreground.

Training Each Codebook for each pixel channel is initiated with an empty set of codewords. For each pixel channel in the training set:

- If there exists a codeword such that:

  `min < value < max`

  Do nothing

- If there exists a codeword such that:

  `min < value < (max + learnHigh)`

  set *max* to be *value*

- If there exists a codeword such that:

  `(min - learnLow) < value < max`

  set *min* to be *value*

- Otherwise create new codeword with:

  `min, max = value`

We also keep track of every time a codeword satisfies one of the above conditions, and record the frame that happened - i.e. this is the last access time. This is used periodically to remove seldom-used ”stale” codewords. This is what gives the CB algorithm good training performance with moving foreground objects.

The paper states the following advantages:[19]:

- **Unconstrained Training** allows moving foreground objects in the scene during the training period. When compared to MOG and Kernel density estimation, the CB method was able to obtain the most complete background model, when trained on a video with many moving people.
- **Training Speed** CB was 4.72x faster than MOG, but 0.04x slower than Kernel
- **Background subtraction** CB was 2.53x faster then MOG and 2.76x faster than Kernel once trained.
- **Adaptive and compressed** background models - The quantized codewords are very compact and take less memory than other techniques. They found on average they needed only 6.5 codewords per pixel for a 5 minute video at 30fps. This means that CB background models can be trained over many 1000s of frames, while others can only have a limiter number of frames in memory.

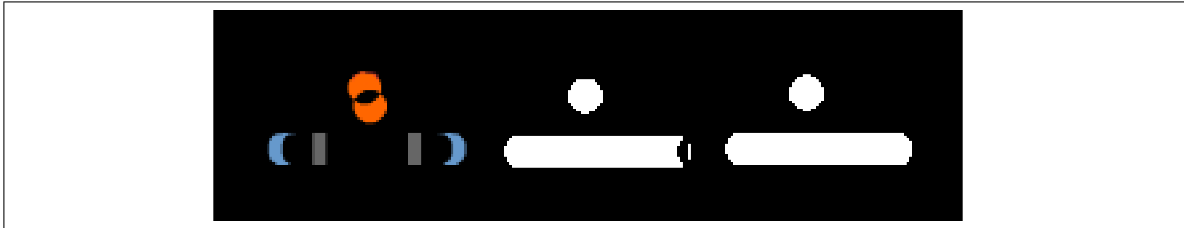

Figure 4.17: 3 Different types of background subtraction i) frame subtraction ii) MOG iii) Codebook

I also chose to analyse the 3 implementations I used. A drawback of the paper is that it only tested on 1 simple scene. I chose to test it on 2 scenes - 1 with lots of movement, the other with less.

Fairly stationary scene:

Method	Training Speed	Detection Speed
Frame Subtraction	n/a	140
MOG	47	65
CodeBook	72	74

Lots of movement:

Method	Training Speed	Detection Speed
Frame Subtraction	n/a	140
MOG	44	64
CodeBook	50	53

As you can see, my results did not match the paper's. Training-wise, Codebook was only 53% faster than MOG with a fairly stationary scene, but went that went down to only 14% faster when there was lots of movement. CB performance fell dramatically, because during the initial training period, it does not clear stale entries. This means that there are lots of stale codewords still hanging about, which means lots of comparisons and a performance decrease. Detection speed-wise, CB had 14% faster detection for a stationary scene, but actually 12% *slower* than MOG. This is because, when there is lots of movement, there is a higher number of codewords that are stored.

How much of these performance differences can be attributed to OpenCV's highly optimised implementation, I can't tell. Efficacy-wise, both implementations were about equal for 2D games.

Blob Detection

To find entire connected components in the resulting background subtraction image, I chose to use border-following method of Suzuki described in section 4.4. I then approximated these contours using Teh-Chin approximation for the final contours.

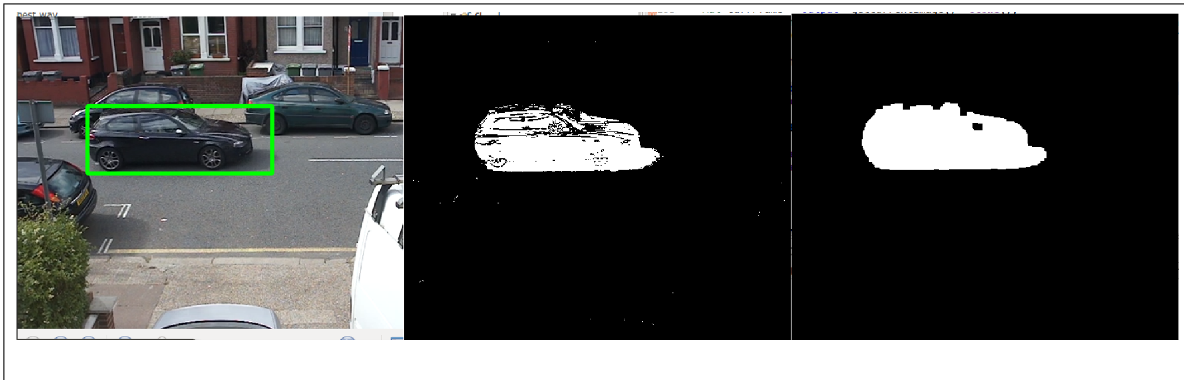

Figure 4.18: Background Subtraction, followed by closing, opening

However as you can see in figure 4.18, background detection produces some noise (white isolated spots) and we will see this feature prominently in 3D games. This means, we needed to do some preprocessing to remove noise. This was achieved using a combination of morphological operators - dilation and erosion.[8]

Dilation Dilation is a convolution of the image with a *local maximum* operator. This means the image pixel under the anchor point is replace with the maximal value in the kernel. Dilation causes the white areas of the image (i.e. foreground) to expand, and is usually used connect multiple components together.

Erosion Erosion is the opposite - it is a *local minimum* operator and causes areas of white to contract. Multiple iterations of erosion are often used to eliminate lone outliers - i.e. noise that is only a few pixels in size as these will disappear after the convolution.

Closing This is X dilations followed by X erosions.

Opening This is X erosions followed by X dilations.

Initially I tested erosions followed by dilations, under the impression that I should remove noise first, then restore the area of objects left. This worked well with CB subtraction, but caused problems with MOG background subtraction. This is because, objects were often "patchy" (figure 4.19)- lots of smaller areas of white covered the object, rather than 1 large "blob"). The dilation caused often caused these to disappear or severely distorted the shape or size.

Instead I settled on 3 iterations of closing (3 dilations followed by 3 erosions) and then 3 iterations of opening (i.e. 3 erosions followed by 3 dilations) with a 3x3 kernel. The closing served to connect nearby areas into larger blobs so they wouldn't disappear, and the opening served to eliminate elements that arose purely from noise. Both these are approximately area preserving, so the resulting image is mostly noise-free, while preserving outlines of moving objects (figure 4.19).

[8]http://www.mathworks.com/help/toolbox/images/f18-12508.html

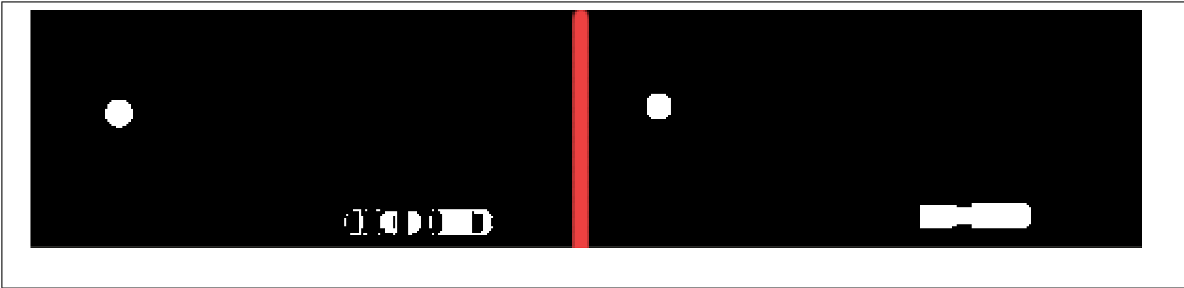

Figure 4.19: The image on the left is the result from MOG background subtraction, once it has started to accumulate the paddle into the background. The image on the right is once it has been closed then opened

Results and Improvements

After some tweaking, generic Breakout worked great with every game I tested on. Sometimes, when the paddle stayed in a single position for an extended period of time, it would start "blending" into the background if MOG or CB background subtraction was used with constant learning. This was easily solved with a number of possible solutions. The first and simplest is just framing the game, so that the paddle is not seen by the GameProfile. This is valid for breakout, as the paddle follows the mouse x coordinates regardless.

The second was similar to framing, except this was done in code. The following call-in was added:

`vision.excludeRegion(x, y, width, height)` This excludes *all* vision algorithms from computation on the rectangular region defined by the arguments. This includes `matchImage()`, `getForegroundObjects()`, `matchShape()`, `matchColours()` and `getTrackedObjects()`

The third was to stop the background subtraction from learning (i.e. the background model stays fixed). The following call-ins were added:

```
vision.stopBGSLearning()
vision.startBGSLearning()
```

`stopBGSLearning()` could be called after X number of frames, or when the user feels fit by adding associating a key with it:

```
function callOuts.keyRelease(key)
    if key == 's' then
        vision.stopBGSLearning()
    end
end
```

4.5 Generic Tetris

Introduction

Tetris is a puzzle video game released in 1984. It has become one of the most popular games of the 20th century and rarely needs an introduction. Nevertheless, here is a short description taken from the internet[9]

"A random sequence of tetrominoes (see figure 4.20); shapes composed of four square blocks each fall down the playing field (a rectangular vertical shaft, called the "well" or "matrix"). The objective of the game is to manipulate these tetrominoes, by moving each one sideways and rotating it by 90 degree units, with the aim of creating a horizontal line of ten blocks without gaps. When such a line is created, it disappears, and any block above the deleted line will fall. With every ten lines that are cleared, the game enters a new level. As the game progresses, each level causes the tetrominoes to fall faster, and the game ends when the stack of tetrominoes reaches the top of the playing field and no new tetrominoes are able to enter. Some games also end after a finite number of levels or lines."

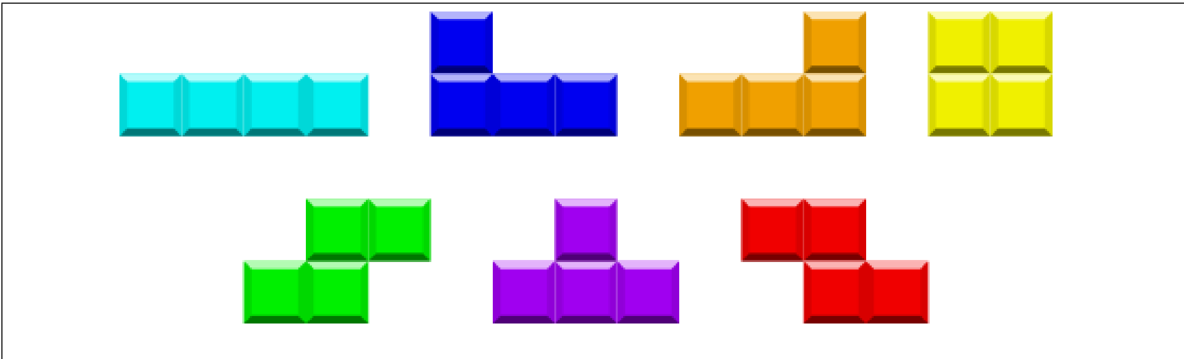

Figure 4.20: The seven one-sided tetrominoes in their Tetris Worlds colors. Top row, left to right: I, J, L, O. Bottom row: S, T, Z.

Desired GameProfile

Here the idea is to parse Tetris game-state. To simplify the GameProfile, the only important part is actually the type, rotation and position of the falling Tetrimino. This is because the GameProfile can internally store all blocks that have fallen (i.e. the blocks at the bottom). This however does mean that unlike the previous games, this GameProfile cannot start mid-game.

The AI for Tetris is out of scope for this project. In theory however, once we have a GameProfile that can parse state correctly, we could plug this state into the AI. The AI would find the best next move, and the GameProfile would act on this to move the Tetris blocks by using call-ins.

[9] http://en.wikipedia.org/wiki/Tetris

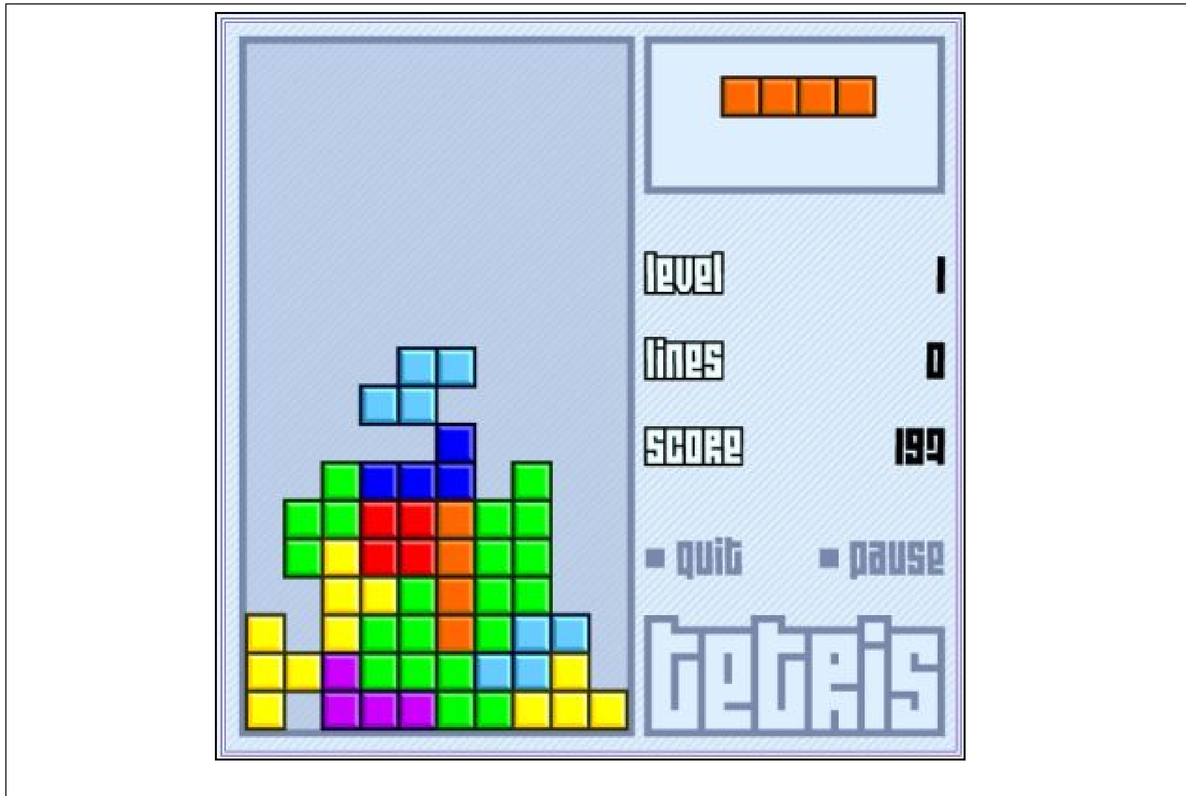

Figure 4.21: An online clone of the original Tetris game

The GameProfile needs to know about the pieces themselves. Rather than having code that represents Tetriminos, this can be achieved by taking images of each one and storing them in a table:

```
pieces = {"Z.png", "BZ.png", "Cube.png", "Long.png", "L.png",
          "BL.png", "T.png"}
```

A simple solution would be to use the `matchImage()` function. Unfortunately, there are $2+4+4+1+2+4+2 = 18$ different permutations of the possible orientations the pieces could be in. With the current template matching algorithm, this would run slowly.

We can apply the background subtraction techniques to identify the contour of the falling piece. Now, we know the bounding box of the tetrimino, can constrain template matching to a single iteration - it would be extremely fast and simple!

However, hopefully by now you may have realised that I am trying to get the GameProfiles as generic as possible. Template matching is less useful in such a situation, as the colours of both the pieces and the background will differ (remember, at the moment template matching is done on the entire bounding box, not just the contour). Also the sizes might be different, although this is not much of a problem, as we know the scale now, so template could be scaled to the same size before matching.

Instead I have decided to match the contours of the shapes. This will work with *any* tetris game, as the tetriminoes are always the same shape. This gives rise to the following

vision call-in:

`vision.matchShape(image, image)` Returns 2 values: The first is a value between 0 and 1 that indicates how well the shapes in the two images match. 1 indicates a perfect match, while 0 indicates no match. The second is the rotation difference between the shapes. It is between 180 and -180 degrees.

This results in the following GameProfile, which prints out the shape and the rotation:

```
pieces = {"Z.png", "BZ.png", "Cube.png", "Long.png", "L.png",
              "BL.png", "T.png"}

shapeAccuracy = {}
rotation = {}

function callOuts.newframe(frame)
   objects = vision.getForgroundObjects(1)
   if objects ~= nil then
      handle = vision.getHandle(objects[1])
      for i, v in ipairs(pieces) do
         countourResults[i], rotationResults[i] = vision.matchShape(v, handle)
      end
      maxValue, maxIndex = findMax(contourResults)
      print("Found object: " .. pieces[maxIndex] ..
                " with rotation: " .. rotation[i]);
   end
end
```

The GameProfile finds the largest moving object (the falling tetrimino). It then compares it with every stored tetrimino using `matchShape()` and prints out the result of the best match and it's rotation. Now that it knows what object is falling, it can pass this onto the AI, which will tell it which position to move it into. Then using `input.PressKey()` it can rotate and move the tetrimino into position.

`vision.getHandle(object)` - This call-in returns a "handle" to an object, which can be used in all GameScript call-ins, where an image is expected (e.g. `matchShape()`) . This is the same type of handle that `getImage()` returns (see section 4.2)

`findMax` is a GameScripter library function that returns the maximum value in an array, and it's index.

Background

Histogram Creation and Matching

On a general level, histograms are simply collected counts of underlying data organized into a set of predefined bins. The bins can be populated by counts of features computed from the data, in our case from the colour.

It is usually best to first convert the image from an RGB image to a HSV image as this is more useful interpretation of colour. An RGB image splits the image into 3 channels for the amount Red/Green/Blue each pixel has. Our aim is to match colour which means we would need to take into account all 3 channels, as what we see, is a combination of these 3 base colours.

Converting to a HSV image gives each channel its own properties:

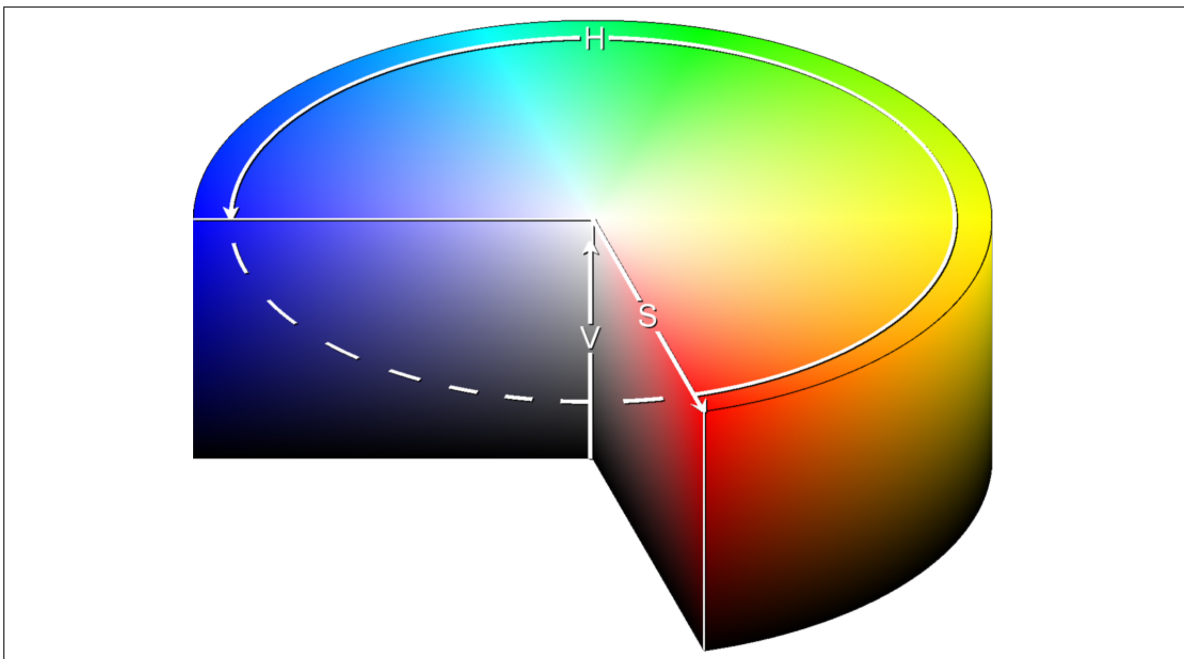

Figure 4.22: HSV Representation

Hue this can be thought of as the base colour of the object. *The angle in the diagram.*

Saturation is the perceived intensity of a colour. *The radius in the diagram.*

Value corresponds to the amplitude (or "intensity) of the colour (reflects the subjective brightness perception of a colour for humans along a lightness-darkness axis). *The vertical axis in the diagram.*

For 2D games, this particular representation does not provide us with much benefit over RGB, as colour of objects usually does not change for the duration of the game. However in 3D photo-realistic games, we need to take into account variable lighting conditions and shadows. In the HSV representation, shadows have no effect on hue or saturation, just the

value. Specular reflections will have constant hue (though different saturation and intensity). Therefore it is easier to design algorithms for histogram matching using HSV.

The first task of matching histograms, is to normalize them. This is an important step as it enables us to match objects of different sizes and match only based on the distribution of colour. The normalized count is the count in a particular bin, divided by the total number of observations (multiplied by the class width - but in our case all widths are the same so this can be ignored). If we didn't normalise, an identical object that is twice the size will have twice as many counts and so distort most matching algorithms even though the distribution of counts across the bins is identical.

The ability to compare to histogram in terms of some specific criteria for similarity is outlined by Schiele and Crowley[26]. There are a number of possible distance metrics:

Correlation (perfect match = 1, total mismatch = -1):

$$d_{correlation}(H_1, H_2) = \frac{\sum_i H_1^{'}(i) \cdot \sum_i H_2^{'}(i)}{\sqrt{\sum_i H_1^{'2}(i) \cdot \sum_i H_2^{'2}(i)}}$$

where $H_k^{'}(i) = H_1(i) - (1/N)\left(\sum_j H_k(j)\right)$ and N equals number of bins in the histogram.

Chi-Square (perfect match = 0, total mismatch = unbounded):

$$d_{chi-square}(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) + H_2(i)}$$

Bhattacharyya (perfect match = 0, total mismatch = 1):

$$d_{bhattacharyya}(H_1, H_2) = \sqrt{1 - \sum_i \frac{\sqrt{H_1(i) \cdot H_2(i)}}{\sqrt{\sum_i H_1(i) \cdot H_2(i)}}}$$

The problem with Schiele and Cowley's algorithm is that changes in lighting conditions can shift the histogram left or right as follows:

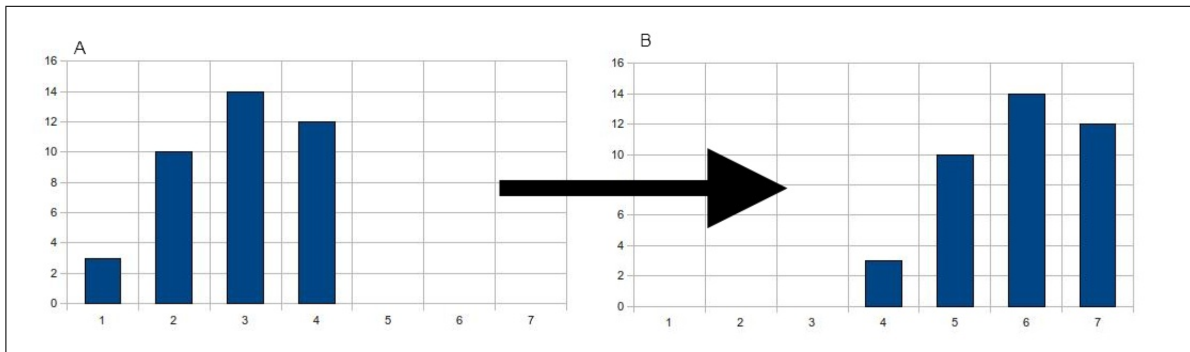

Figure 4.23: Histogram Matching

Our algorithm would give very poor matches for the two above histograms as it does bin for bin (represented by columns) matching (e.g. bin3 for A = 14 and bin3 for B = 0, therefore the match is very poor).

Earth movers distance Algorithm [27] can be informally described as the amount of work needed to change histogram 1 to histogram 2 through the process of shovelling dirt between bins. The further the shovelled dirt needs to move the higher the cost. This means that this measure would be more appropriate for cases like this as the amount of shovelling needed is quite small, even if the bins do not match very well.

Edge Detection

Edges occur at boundaries between two different regions in an image, where there is significant change in image intensity. Edge detection examines the rate of change of intensity near the pixel - sharp changes with steep gradients are good evidence of an edge, while slow changes suggest the contrary.

Usually, edges can be classified into the following categories as shown in figure 4.24:

Step Edges - image intensity abruptly changes from one value to a different values

Line Edges - image intensity abruptly changes value, but then changes back again soon after.

Roof Edges - in real images, there is usually a smoothing effect causing roof edges, which have less of an abrupt change in value.

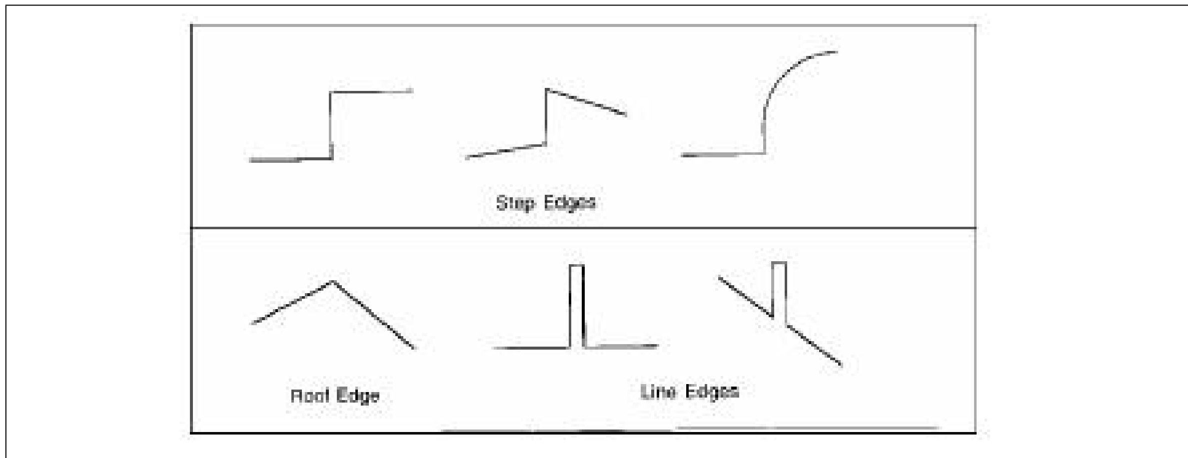

Figure 4.24: Types of Edges

An edge detector is an algorithm that produces a set of edge points from an image. Most edge detection techniques have been around for decades. They involve convolution operators that estimate the 1st or 2nd order derivatives in an image. Most have multiple convolution masks to find edges in vertical, horizontal and diagonal directions. 1st order derivative operators include: Roberts, Sobel, Prewitt, Robinson, Kirsch - which are all $3x3$

matrices except Roberts which is $2x2$. The Laplacian operator is a commonly $3x3$ used 2nd order derivative operator. The 2D version, diagonally discriminating version is given below as an example:

$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix} \tag{4.14}$$

Figure 4.25: Example of edge detection. Taken from `http://en.wikipedia.org/wiki/File:EdgeDetectionMathematica.png`

A more advanced and interesting edge detector was proposed by J. Canny[28]. The Canny edge detector builds upon the previous detectors and tries to assemble individual edge candidate pixels into complete contours (a sequence of points, usually closed). There are 2 main steps:

First Derivatives The 1st derivatives are computed in x and y and then combined into four direction derivatives. Like in the simpler edge detectors, points where these directional derivatives are local maxima are candidates for assembling into edges.

Contour Construction Instead of using a single threshold at which a given intensity gradient switches from "edge" to "not-edge", Canny uses thresholding with hysteresis. This requires 2 thresholds - high and low. The following rules are applied to each pixel:

- If a pixel has a gradient larger than the high threshold - it is accepted
- If a pixel has a gradient below the low threshold - it is rejected
- If a pixel has a gradient between high and low thresholds, it is accepted only if it is connected to a pixel that is above the high threshold.

This leads to a simple algorithm that "traces" object boundaries, with more accuracy than the other operators as it is not affected as much by noise and blurriness in the images. These boundaries can then be treated as polygons for further processing (usually they are simplified with techniques described in section 4.4:Contour Extraction)

Contour Matching with Moments

We need a way to summarize the shape of an object so that it can easily be compared to another, invariant of both size and rotation. One way to achieve this is by using Hu invariant moments. [29]

A moment is a gross characteristic of the contour computed by integrating over all the pixels of the contour. i.e:

$$m_{p,q} = \sum_{i=1}^{n} I(x,y)x^p y^q$$

A *central moment* is the same as the moments mentioned above except that the value of x and y are displaced by mean values.

$$\mu_{p,q} = \sum_{i=0}^{n} I(x,y)(x - x_{avg})^p (y - y_{avg})^q$$

This then needs to be normalised so that the moments are size invariant and also coordinate system invariant. This is done by diving by m_{00}^X, where X is the power needed for the resulting normalised moment to be independent of scale:

$$\eta_{p,q} = \frac{\mu_{p,q}}{m_{00}^{(p+q)/2+1}}$$

Hu invariant moments of a contour are linear combinations of central moments. They are invariant under translation, changes in scale and rotation. Therefore they a perfect tool for comparing contours. Please refer to Appendix A.1 for full list of Hu invariant moments. We are able to compare 2 shapes by comparing each Hu moment individually and then combining these into a single figure with which summarizes the quality of the match.

We could also use other discriminants, based on geometric features[30], for matching. For example, number of vertices, distance of each vertex from the centre of gravity, angle subtended by the vertices at the centre of gravity or convexity defects.

Freeman Chains (as described in Figure 4.4:Contour Extraction) can also be used to match contours by using pairwise geometrical histograms. Histograms, quantised into 8 bins, are created by counting the number of each kind of line in the freeman chain code. The standard Histogram matching techniques (described in section 4.5:Histogram Creation and Matching) can be used to match these histograms. The problem with this is even though shapes give perfect histogram matches when rotated by a multiple of $(360/x)^\circ$, when they are in-between these rotations matching gets very inaccurate.

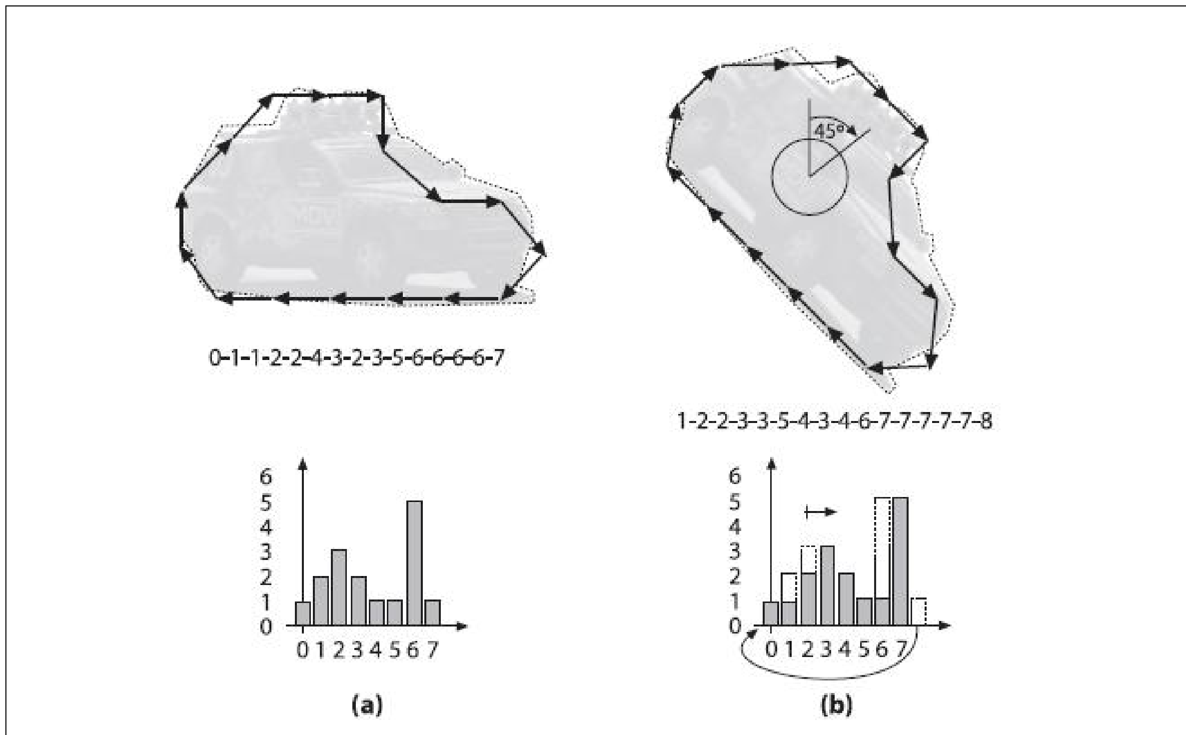

Figure 4.26: Freeman Chain approximation and matching. Matching is very accurate when rotations in multiples of 45°

Implementation

Tetrominoes are only ever rotated in 90° degrees, so if was going to implement a tetris specific algorithm, I would probably use Freeman chain approximation matching as it would be fast and efficient. However, in the general case it is inadequate, so instead I chose to use Hu Moments. Finding the contours for the stored images is roughly the same as for the foreground segmentation objects, except there is a need for edge detection.

Edge detection To find the edges in the colour images I chose to use the OpenCV implementation of the most effective edge detector - Canny. I had the choice of either applying Canny on each colour channel separately, or combine all the channels into a grayscale image and use Canny on that. The latter methodology would work fine in this case as the stored tetriminoes are on a perfectly white background, however I planned for the general case and decided to do it on each colour separately.

To combine the results of the 3 resulting edge maps, I simply added them together. I felt this gave me the best results I kept as much information in the final edge map as possible. Other possible choices were:

- Use the average of the gradients.
- Use a weighted average based on the quantity of each color at each location.

Contour Finding For the stored images, the Canny edge map is used for contour finding and follows the same algorithm as the connected component labelling (see section 4.4:Blob Detection) i.e. Suzuki border-following algorithm, followed by Teh-Chin approximation. The contour for the foreground object has already been found in the process of connected component labelling.

The contours are now stored together with associated image so that they do not have to be calculated again[10].

Contour matching The approximate polygons can now be compared using Hu Moments. All 7 of the Hu moments are calculated for both polygons and then compared as follows:

$$match(A,B) = \sum_{i=1}^{7} |\frac{m_i^A - m_i^8}{m_i^A}| \tag{4.15}$$

where

$$m_i^A = sign(h_i^A) \cdot log|h_i^A|$$
$$m_i^B = sign(h_i^B) \cdot log|h_i^B|$$

[10]This is encapsulated in the *GenericObject* class in the source-code. This is used for objects found using all of the techniques described in this report

and h_i^A and h_i^B are the Hu moments of A and B respectively (see section A.1).

This means that a perfect match would be 0 and the worse the match the higher the number.

We can find the orientation of an object can also be found using the following combination of moments:

$$angle = 0.5 \cdot tan\left(\frac{2 * m_{1,1}}{m_{2,0} - m_{0,2}}\right) \tag{4.16}$$

In this case, we want to find the *relative* orientation of the matched objects, so we calculate the orientation for both and subtract on from the other.

Results and Improvements

From my testing on 3 online tetris games, the profile was able to identify the falling blocks with 100% accuracy and the rotation with 100% accuracy. Colour information was not needed; contours were enough to uniquely identify each shape. This means that if the script was hooked up to an intelligent AI, it would be able to play with great efficiency!

The contour matching done here was very simple - I wanted to investigate how effective it would be on more complicated shapes. I created a quick test program that took in a picture and rotated it in steps of 1 degree, and calculated the rotation. Here is an extract of the table produced (using the outline of a car, Figure 4.27):

Degrees Rotated	Degrees Estimated	Match Quality
83	82.9401	0.994307
84	83.9819	0.992483
85	84.9635	0.99487
86	85.9753	0.994044

The average error for rotation was a tiny 0.02 degrees, while the match quality was on average 0.994.

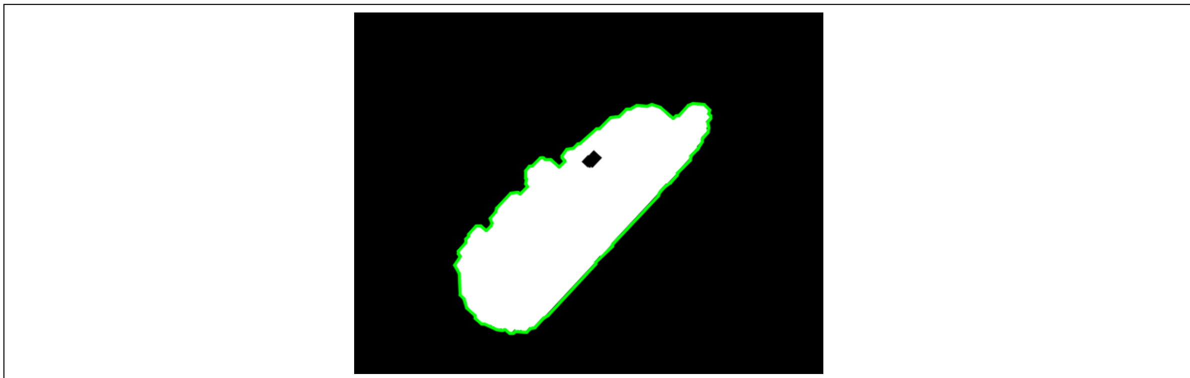

Figure 4.27: The image was rotated in steps of 1 degree, and rotation/match quality was calculated using moments

Removing the hard-coded playing field size

In the GameProfile, the horizontal size of the Tetris field is hard-coded to 10 squares. However this can be calculated from the image - divide the width of the playing field in pixel, by the width of the *shortest* side of the falling object. All tetriminoes have a 2 square, shortest side. The exception to the rule is the I Tetrimino, which is 1, but this can easily be tested for.

To find the width of the field I have implemented a new vision call-in:

`vision.findSquares(maxNumber, minSize)` returns a list of objects representing squares. Returns *maxNumber* or less of squares (ordered by size), with a minimum size of *minSize*.

This allows us to find the largest square on screen, which 99% of the time, is the boundary of the game. It works as follows:

- Use pyramids (see section 4.2) to down-scale then upscale image to filter out any noise.
- Blur the image with a median Blur. I found this reduces any textures that interfere with the latter stages, while keeping edges sharp.
- Do canny edge detection (section 4.5) on all channels of the image and a grayscale version. This works better than just on the grayscale as finds it squares in all colour planes.
- Find contours using Suzuki algorithm and approximate with Teh-Chin (section 4.4).
- Iterate through contours with 4 edges. Calculate angles between edges - if all angles within some threshold (I used 5 degrees) count contour as square
- Sort found squares by size and set return objects in the GameScript standard object table.(Appendix D)

The alternative was to use a Hough line transform[11] to find the straight lines parts of the edge map instead of the Suzuki border-following algorithm. I found however the above worked extremely efficiently, using methods I had already implemented, so I stuck with that.

Now we can find the border of any game, and set the GameScripter to only parse that part of the game:

```
function findAndSetBorder()
   squares = findSquares(1)
   if squares ~= nil then
      vision.setGameBorder(squares[1].x, squares[1].y
                           squares[1].width, squares[1].height)
   end
end
```

This is a method I added to the GameScripter library.

[11] http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm

Other Flash Games

Other than the ones documented here, I have a tested a number of other 2D games, such as with online pool (mostly used template and histogram matching). The call-ins/call-outs provided above, serve their purpose and provide the necessary tools to parse game state. However many are puzzle-based, so the AI is most involved part of the GameProfile. For example, very popular games like Bejewelled and chess are easily parsed, but writing a good AI is complex.

4.6 First Person Shooter

Introduction

> "Does it play Quake yet?"
>
> Chris Emery

A first-person shooter[12] is a video game genre where the player experiences the action through a first-person perspective - i.e. through the eyes of a protagonist. Game-play usually centre around combat with the protagonist's gun, with enemies varying between games - anything from terrorists to zombies and flying robots.

Popular games include Doom, Quake and lately the Call of Duty games - which is valued as one of the largest entertainment franchises in the world.

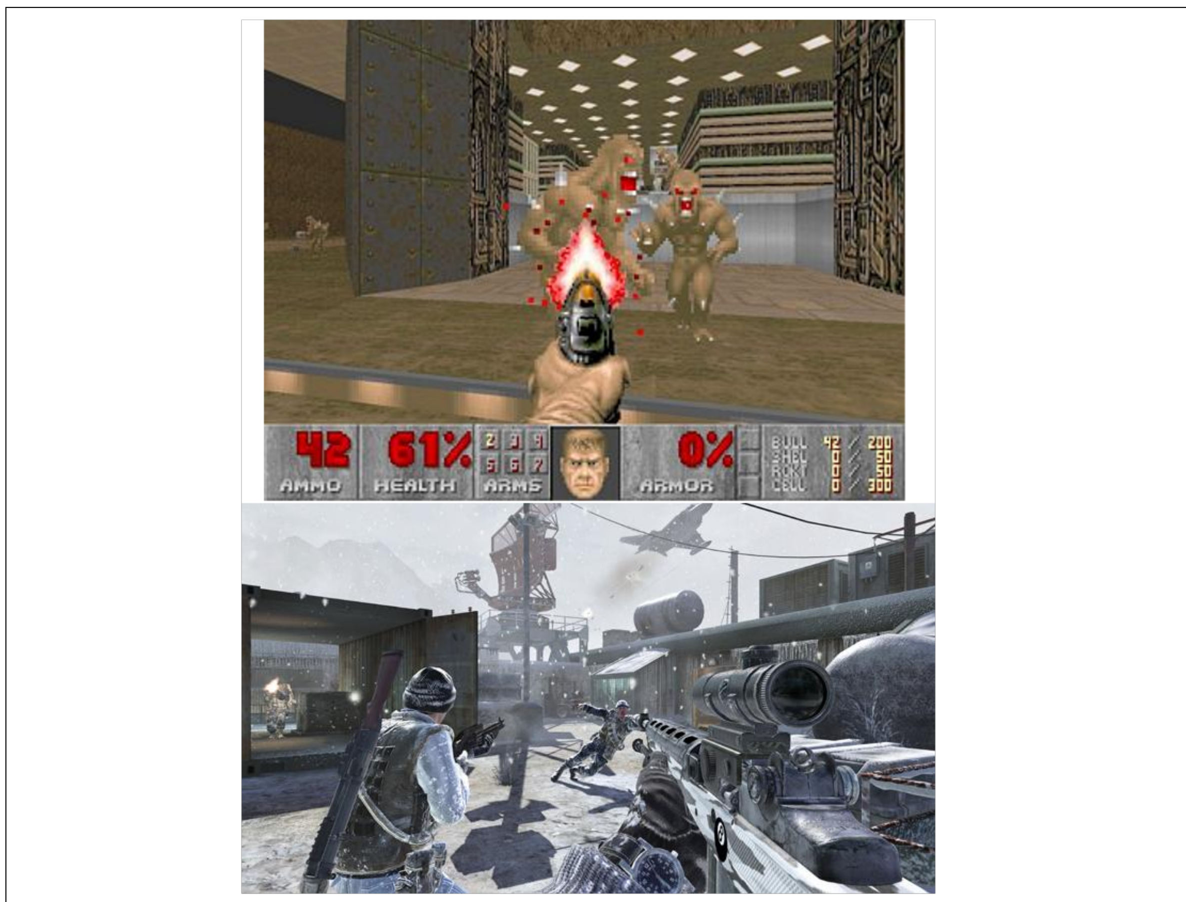

Figure 4.28: First-Person Shooters - Doom (1993) is on top, Call of Duty Black Ops (2010) is on the bottom

[12] http://en.wikipedia.org/wiki/First-person_shooter

Desired GameProfile

In an ideal situation, the GameProfile we would like:

```
function callOuts.newframe()
  x, y = vision.matchObject(EnemyPlayer)
  input.clickOn(x,y)
end
```

Here, the idea is exactly the same as the original whack-a-mole - you find the enemy player on screen, and move your cursor there to click (or in this case that will mean shoot). However, how do we define the variable `EnemyPlayer`? It can't just be a single image like we did for 2D games, as a single image is not enough to define what an enemy looks like. This brings us to object detection frameworks

Background

Object Detection Frameworks

Paul Viola and Michael J. Jones report on "Rapid object detection using a boosted cascade of simple features" [31] sets out a great framework for real-time object detection, especially relevant in for my project. Although it concentrates on face recognition (like a lot of object detection papers), it applies to all object detection.

Unlike some other object-detection procedures[32], this one uses features rather than pixels directly. There are 2 reasons for this: using features is faster, and features can encode ad-hoc domain knowledge that is difficult to learn using a finite quantity of training data. The features they use are in figure 4.29.

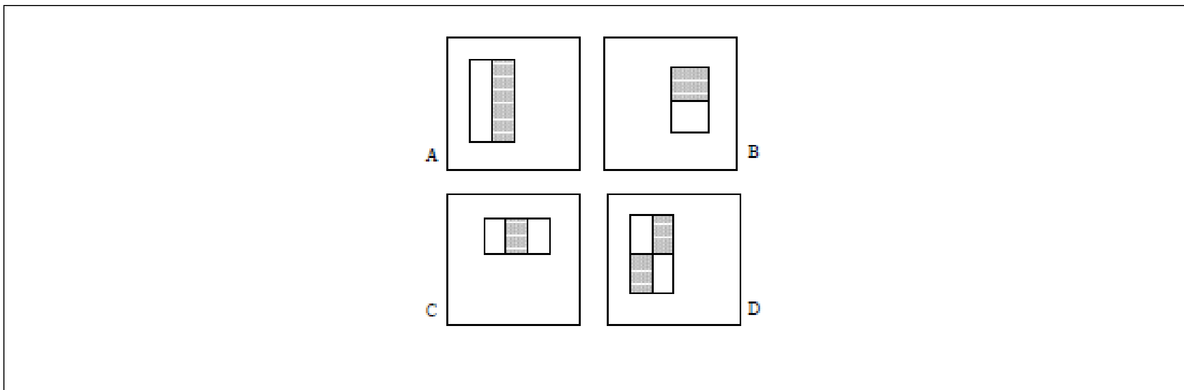

Figure 4.29: Example rectangle features shown relative to the enclosing detection window. The sum of the pixels which lie within the white rectangles are subtracted from the sum of pixels in the grey rectangles. Two-rectangle features are shown in (A) and (B). Figure (C) shows a three-rectangle feature, and (D) a four-rectangle feature. [31]

The features are very primitive compared to alternatives such as steerable filters [33] as they only work in 2 orientations: vertical and horizontal. However they trade-off this flexibility and potential loss of accuracy, with a significant computational efficiency.

Conventionally a detector is created using features which can be scanned over the input image. However to try and retain some scale-invariance, these fixed size detectors have to be run on different resolutions of the image; usually facilitating the need to create an image pyramid. The creation of this pyramid is computationally intensive and prohibitive in a real-time scenario.

To combat this P. Voila and M. Jones introduce a new intermediate representation called an "integral image" with which rectangular features can be easily computed. Essentially they are a summed area table [13] which can be computed in one pass over the original image. The benefit is that any rectangular sum can be computed in four array references, which greatly speeds up the calculation of rectangular features.

In any given image, the complete set of rectangular features exceeds the number of pixels, so learning these features outright using techniques such as support vector machines[34] or neural networks[32] would be incredibly slow. Instead, they propose using a variant of the AdaBoost[35] learning algorithm to find a small amount of features which can combine to form an effective classifier. The idea behind AdaBoost is simple; the learning process is split into a series of rounds, and at each round, the feature classifier with the highest correct classification rate is chosen. In the subsequent round, the weights of each incorrectly classified example are increased so that that they are favoured. This results in a technique where training error approaches zero exponentially with the number of rounds [36].

However AdaBoost is not the only technique for feature selection. Other possibilities such as feature variance[37] and Winnow exponential perception learning[38] are possible, although they are left with larger number of features than AdaBoost.

The final innovative technique they used was the use of an attentional cascade, which also was one of the reasons for choosing AdaBoost as the learning algorithm. Rather than running all classifiers simultaneously, one can construct a cascade of classifiers which achieves increases detection performance while greatly reducing computation time. The cascade can be thought of as a "pipeline" where smaller, more efficient classifiers are evaluated first, gradually increasing in complexity, until last come the more complex many-feature classifiers which are there to achieve low false positive rates (and therefore not sacrificing on Precision). If at any point in the pipeline a classifier is negative the sub-window is immediately rejected. For tasks such as object detection, where the negatives far outnumber positives, this technique works extremely efficiently. The cascade was constructed using AdaBoost, starting with a two-feature strong classifier, however this could be applied to other machine-learning algorithms too, like neural networks[32].

Combined, these techniques produce a very efficient object detection framework; on a 700mhz pentium III processor, a frame-rate of 15 was achieved on 384x288 image with a 77.8% recall rate. Since then, processing power has increased by an order of magnitude, and so by using a similar technique, object detection speed should not be an issue. In most cases, classifiers with more features will achieve higher precision and recall rates, so it is easy to tweak this procedure by tweaking the number of classifier stages and the number of features of each stage.

The major downfall of this report, and this technique in general, is that it concentrates

[13] http://en.wikipedia.org/wiki/Summed_area_table

on detection speed and ignores the consequences on training speed and training sample sets. They used 4916 positives and 10,000 negatives. This is a fairly large sample set size - it would take a while to get such a large sample size in a game, however not unreasonable if playing the game for extended periods of time. More importantly, the training time for a 32 layer detector was in the order of weeks on a single 466Mhz AlphaStation XP900. Even with the rapid increase in computing power, performance improvements in this aspect of this procedure would have to be taken to be applicable to my project, as one of the key requirements is real-time learning.

Tracking and Motion

All the previous techniques discussed such as template matching and cascade classifiers are applicable only to single images - we isolate particular shapes on a frame-by-frame basis. Tracking is a related problem - we use inter-frame relationships to track objects which do not have to necessarily be identified. Tracking and optical flow is a very large and fast-moving field of research in computer vision as of this moment. In this background section, I will only concentrate on the areas of optical flow that are directly related to the implementation of GameScripter; I forgo going into any detail of some vast areas of research such as dense optical flow, although I may mention them when necessary.

Lukas-Kanade Method

Optical flow Optical flow is the distribution of apparent velocities of movement of brightness patterns in an image. [39] It is caused by relative motion of objects and the viewer.

In **Dense optical flow**, a velocity or displacement is associated with *each* pixel in the frame. The simplest form of dense optical flow is *block matching* - attempting to match windows around each pixel from one frame to the next[17].

In **Sparse optical flow** algorithms, velocity or displacement is associated with only a small subset of pixels in the frame. These pixels usually have desirable properties that allow them to be as unique as possible and therefore easily identified between frames.

Although dense optical techniques are usually more accurate, they are extremely computationally expensive. This is because each individual pixel must have a velocity associated with it - however not many pixels can be easily identified between frames. For example, pixels in the centre of a large blank wall cannot be distinguished from each other. This means that the surrounding pixels, and their associated velocities must also be taken into account, leading to costly computation. Sparse optical techniques have the advantage that only a subset of points need to be associated velocities - these points rely only on local information that is derived from some small windows around them. Lukas-Kanade is an example of one of these techniques, and it's derived pyramidal version[40].

Finding Invariant Features The first task is to find points which can easily be found between frames. Edges found using the edge detectors discussed are not suitable, because they usually only have a gradient in 1 direction. This is because of the aperture problem,

which is demonstrated in figure 4.30. As you can see, in the left image, the square is travelling right, but at the points highlighted, we can only detect motion diagonally.

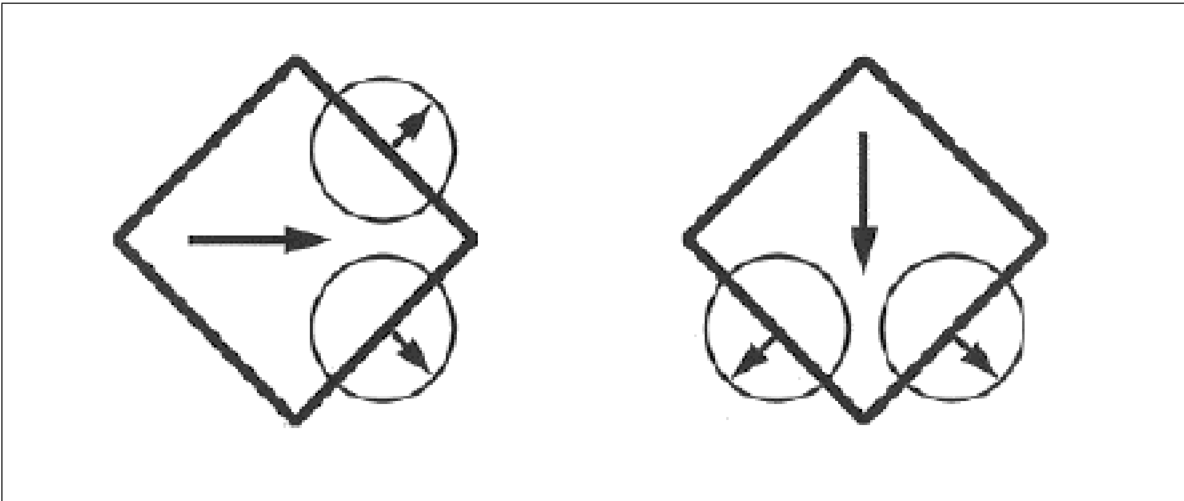

Figure 4.30: The Aperture Problem

If however we tracked the corners of the square, we could track the object in all directions. This means that ideal points, are ones with strong derivatives in 2 orthogonal directions - or *corners*.

The most common definition of a corner is provided by Harris[41] and relies on the matrix of second-order derivatives of the images. Second-order derivatives are used because they ignore uniform gradients. Informally, Harris corners are places in the image where the there is a texture or edge going in at least 2 separate directions, centred at that point. For a excellent explanation of the maths behind Harris Corners, I refer the reader to Utkarsh Sinha's online explanation.[14]

Sub-pixel Corners Now that we can find corner locations on a raster image, it is useful to refine these locations to sub-pixel level for greater accuracy. This can be done in several ways, but the simplest is through the use of a set of dot product equations[15]:

- First take two points **p** and **q**. **q** is the estimate integer corner position (i.e. in this case the Harris corner). **p** is a any other point around q.

  Point **q** is a good estimate of the corner, so lies very close to it. **p** can only lie at one of two places:

  1. A flat region
  2. On an edge

- If **p** lies on a flat region, the gradient at **p** is zero. Therefore the dot product of the gradient at p, and the vector **q-p** will also be 0, regardless off the position of q:

[14] http://www.aishack.in/2010/04/harris-corner-detector/
[15] http://www.aishack.in/2010/05/subpixel-corners-increasing-accuracy/

$$< \nabla I(p), q - p >= 0$$

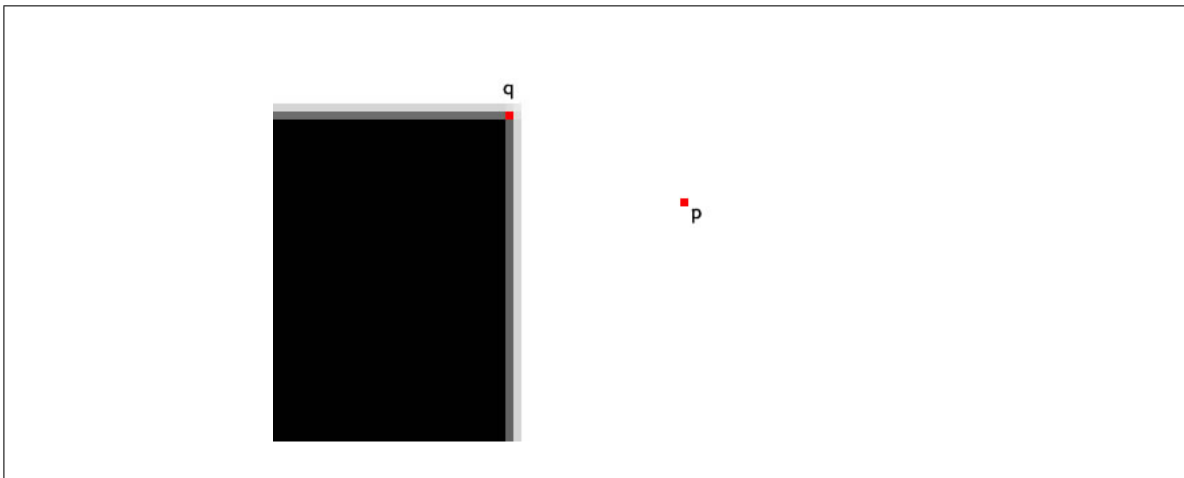

Figure 4.31: Point p is on a flat regions

- If the **p** lies on the edge, the gradient at **p** will be some defined vector. However this gradient will be perpendicular to the vector **q-p**. This therefore means we have the same dot product equation again:

$$< \nabla I(p), q - p >= 0$$

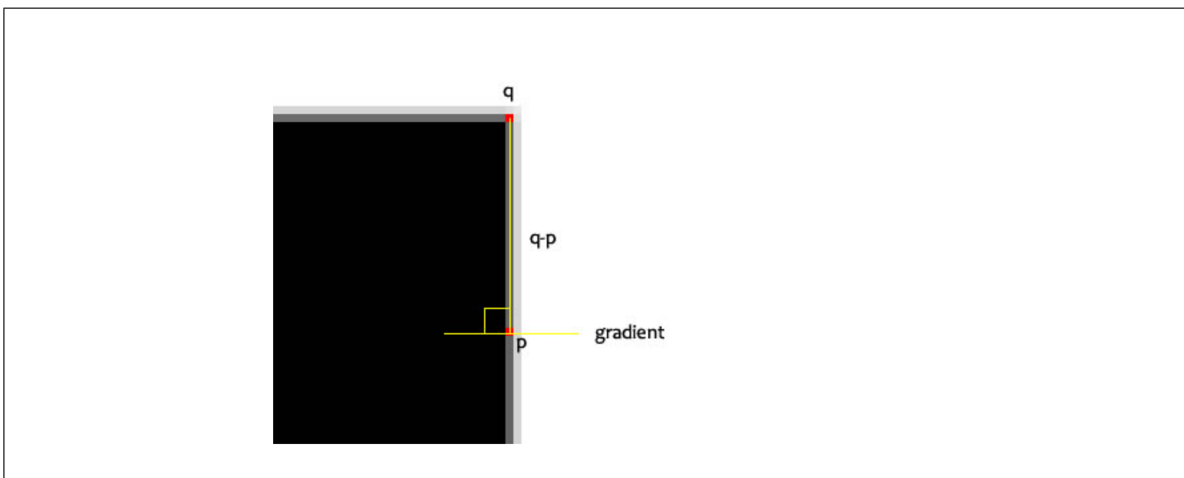

Figure 4.32: Point p is on a gradient

- We can now assemble many such pairs of gradients and form a system of equations. Each equation equals 0 and if you solve for **q**, you get a sub-pixel estimate of the corner!

Assumptions The Lukas-Kanade method has 3 major assumptions:

Brightness Constancy Image measurements (usually image properties such as brightness) in a small region remain the same, although their location may change:

$$I(x+u, y+v, t+1) = I(x, y, t)$$

Spatial Coherence .

- Neighbouring points in the scene typically belong to the same surface and so therefore typically have similar motions
- They also project to nearby points in the image so we expect spatial coherence in image flow

Temporal Persistence The image motion of a surface patch changes slowly over time - i.e. objects do not move much between frames.

These assumptions can lead us to an effective tracking algorithm. The maths has been extremely well documented, and so I refer the reader to the LearningOpenCV[17] book for the best explanation.

Gaussian Pyramid Application Lukas-Kanade uses small local windows to decrease computational complexity, however large motions can move points outside the local window. One can increase the window size however this causes a significant performance hit (the extent is investigated in the implementation section). Instead, we can make use of image pyramids (section 4.2) and tracking can start at the highest level of the pyramid (with the lowest detail) and work down to lower levels (highest detail).[40]

K-Means Clustering

The K-means algorithm attempts to find clusters in the data. In it's most general form, a set of N data points is grouped into k clusters in I-dimensional space.[42]. K-means is a very simple iterative refinement process and consists of the following steps:

1. User input is N, I-dimensional data points, and a chosen k for the number of clusters
2. Randomly assign cluster centre locations for the initial iteration
3. Assign each data point to cluster $\hat{k}^{(n)}$ with the closest mean:

$$\hat{k}^{(n)} = argmin_k(distance(m^{(k)}, x^{(n)})) \tag{4.17}$$

4. Update the clusters' centres (mean) to be the centroid of their associate data points:

$$m^{(k)} = \frac{\sum_{x_i \in \hat{k}^{(n)}}}{|\hat{k}^{(n)}|} \tag{4.18}$$

5. Repeat steps 3-4 until centroids stop moving (i.e. convergence has been achieved)

Problems Although the K-means algorithm is an effective clustering algorithm it has it's problems[17]:

- K-means is not guaranteed to find optimal solution. (Although it does always converge to some solution)
- The number of cluster centres have to be chosen in advance; the algorithm does not specifying the optimal number of clusters.
- K-means does not take into account the covariance in the I-dimensional space.

My solutions to these problems will be discussed in the implementation section.

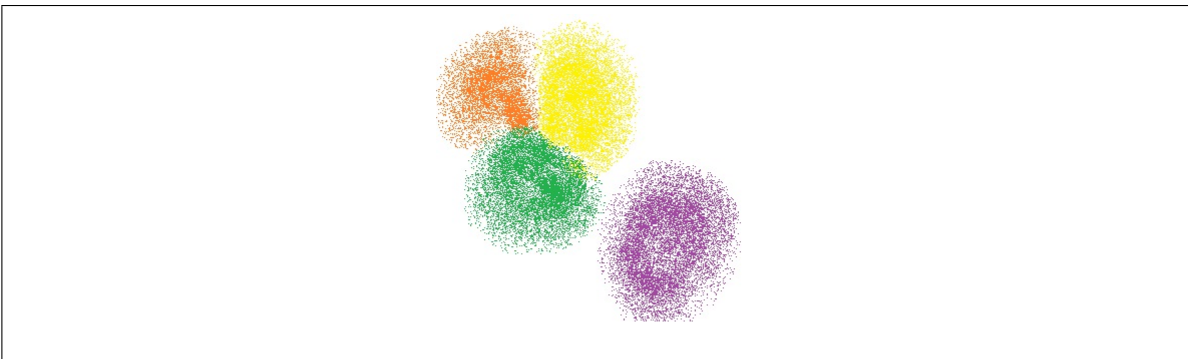

Figure 4.33: Result of KMeans clustering with 4 cluster points. The points are colour coded with the cluster they belong to

Implementation and results

Application of previous techniques

Template Matching As I have shown in previous games, template matching is a very simple and easy way to find objects. Unfortunately, it only works well on 2D games, where objects are only seen from 1 perspective. In 3D games, where objects can be close (and therefore large), far away, rotated, observed from any angle and affected by lighting conditions, template matching is completely ineffective. One would have to use templates of varying scale, angles and lighting conditions, which would be computationally infeasible..

Contour Matching This has more potential as I have shown that there are effective rotation and scale invariant matching techniques. However, once again we are not matching a single image, but a 3D object which will have a different contour from every angle (see figure 4.6).

Histogram Matching (Back-projection) Back-projection builds upon the histogram matching techniques discussed int the previous section. It is a way of recording how well patches of pixels (or a single pixel) fit the distribution in the histogram, by sliding this patch across the image in a similar way to template matching. This has more potential than the previous techniques and is often used in real world applications such as skin-tone detection.

Enemies in FPS games tend to have a similar colour distribution from all angles and it tends to be fairly uniform (e.g. the legs are the same about the same colour as the upper body and so on). This suggests using a small patch with back-projection. However, very often the games try to be as photo-realistic as possible, and so enemies actually wear camouflage - severely reducing the effectiveness of the methodology.

Background Subtraction The background subtraction techniques were directly transferable to a 3D environment without needing much tweaking with the (major) drawback - the viewpoint of the user cannot move. The algorithms used work on a pixel-by-pixel basis, and are not designed to cope with any kind of relative motion between the camera and the view (as discussed in the Background section). A rudimentary GameProfile could be written to find moving foreground objects and then shoot that position. (The player would have to be stationary and wait for enemies to come to him (figure 4.35) and the act of moving the weapon cross-hair towards the enemy will mean that the background will change (figure 4.36 and the model will usually have to be re-learnt before the script is effective again. This is not an effective solution.

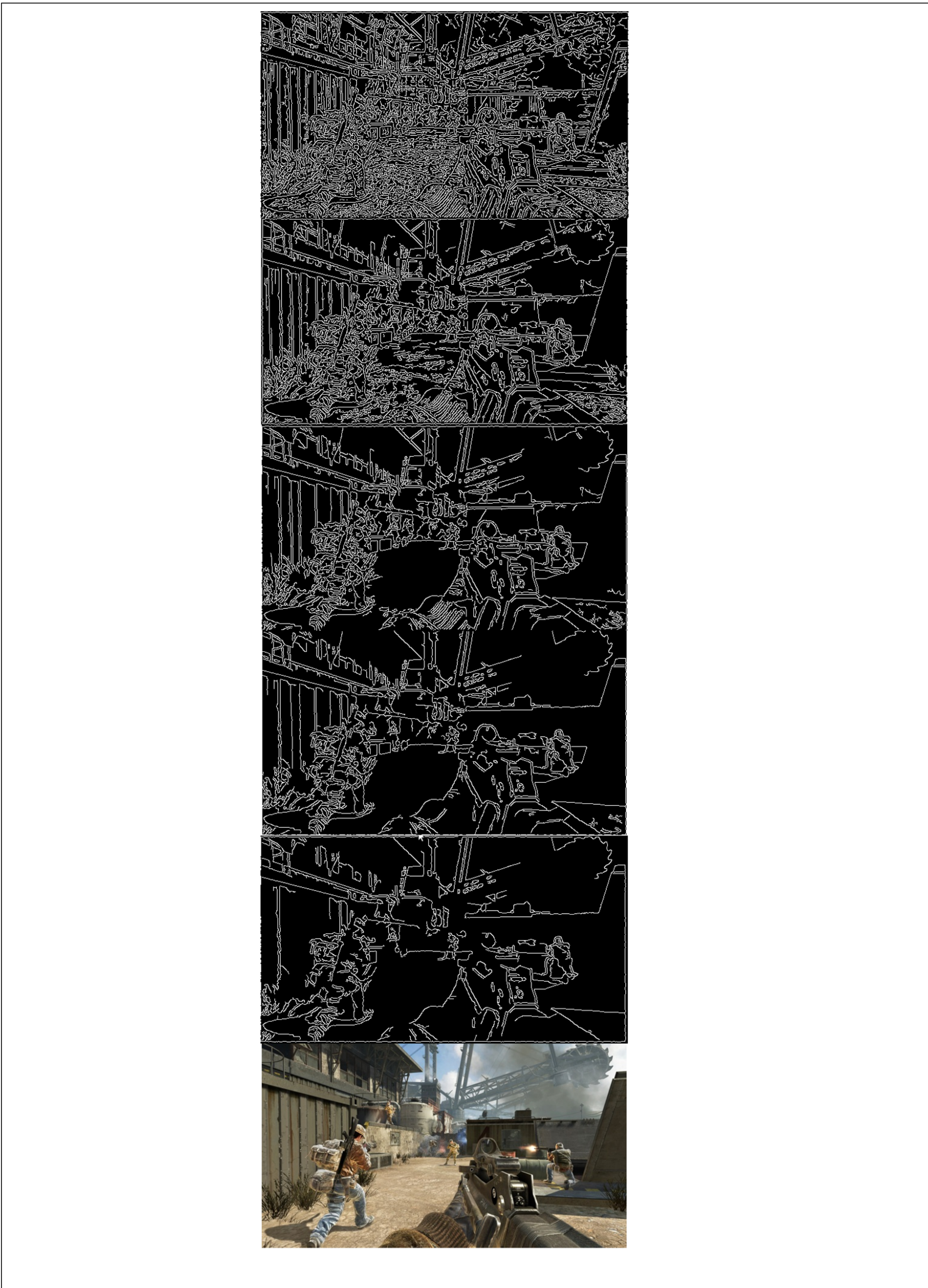

Figure 4.34: Can you spot the enemy players from the Edge Maps? Using Canny edge detector with varying thresholds

Figure 4.35: Background subtraction in 3D environment. The Enemy has quite clearly been identified. However the viewpoint cannot move.

Figure 4.36: Background subtraction in 3D environment when the player is moving. Resulting image is indecipherable

Object Detection Frameworks

One area I originally had high hopes for, was using the Viola-Jones' boosted rejection cascade technique outlined in the background. I created a GameScript that would observe the player - when they shot, an image of what they shot at was recorded and this were used as a positive. When they weren't shooting, images were periodically taken and used as negatives. These images were then trained using a modified version of OpenCVs provided *cascadeTraining* facility, which implements an extension of the Viola-Jones' technique to include diagonal features/citeLienhart02anextended. The modification was to modify the tool to be able to run on images "as they become available", instead of having to have the whole data-set available immediately.

Unfortunately this completely failed - I was not able to build any classifiers that had any kind of analysable results. I believe the problem with this approach has 2 main problems:

1. It required a large, high quality data set - such as those often used for face detection[31]
2. The use of only one classifier is not enough, when considering object detection from all angles.

Problem 1 was gathering data was a harder task than I initially thought. My technique

of taking images everywhere where the person shot created a large number of images, but quality was insufficient for object detection training. This is because:

Not every image contained an enemy In my findings, I found often players would shoot speculatively hoping to hit an enemy - e.g. shooting at popular areas where enemies usually hide. Other times, players used features of the game-mechanics shooting at doors or boxes instead of enemies themselves, knowing that bullets are able to pass through them, hitting the enemy player. Having even a small number of these images can poison a data-set quite easily, although the solution is equally simple - just ask the player to refrain from speculative shooting where the enemy is not in view.

Many enemies where partially hidden Often enemies were partially obscured by boxes, cars or even other players.

Scale of enemies in images greatly varied Enemies sometimes were very far away, sometimes close. For a classifier to work, the training set of the enemies should be normalised to the same size.

Not all enemies were equal Not everyone who was shot at was an enemy player. In some games there were tanks and helicopters. These of course, would need another classifier.

Even if all the problems listed above were solved, (i.e. have a large set of perfectly cropped images of enemies), the 2nd major hurdle is the use of only 1 classifier. The reason why object recognition frameworks work is because they find common invariant features in the dataset which can be used in the detection stage[31]. Combining images of enemies from all angles is undesired, as then we are trying to find invariant features from *all* sides. In face recognition, separate classifiers are used for frontal face, right profile of the face & left profile of the face (which is usually just the mirror of the right profile). Each classifier is very different from the other - for example the frontal face often has very strong features resulting from the placement of both eyes; such as the fact there are 2 of them, of very similar size, at a similar horizontal level. Trying to train a classifier on enemies from all angles is akin to trying to train a classifier of all 3 face types simultaneously; the resulting classifier would be useless.

Optical Flow

I was eager to try and get a prototype GameProfile for a first-person shooter, without any of the drawbacks of background subtraction or the need for hand-compiled training sets. I wanted a system that could do foreground/background segmentation with arbitrary camera motion and in real-time. A quick bit of research showed that solutions are still very much in their infancy and it is currently a major area of research in computer vision. Time was limited, so I took the approach of using whatever tools were provided by OpenCV and easily accessible.

The idea was to model the average apparent motion of the visual scene and then find areas areas of the image that differed. OpenCV provides a 1 sparse optical flow algorithm (Lukas-Kanade) and 3 dense optical flow algorithms - Block Matching, Horn-Schunck and Farneback. The 1st two are fairly old techniques that are slow and not suited to real-time

systems. Farneback is newer and uses image pyramids to speed up performance. It can be customized with parameters such as number of pyramid levels, the scale between pyramid levels, window size and iterations which would give me a lot of manoeuvrability to tweak it to run well in a real-time environment.

However I decide to stick with optical flow I had researched and understood - Lukas-Kanade (LK) (section 4.6). Anything I learnt from testing with LK could be transferred to a dense optical flow algorithm in the future.

Implementation

In the first frame, I find the initial Harris corners that I wish to track, on a grayscale image. I chose the maximum number of points to find as the square-root of the image area (therefore I will try to find 500 points in a 500x500 image), as I felt this was a good starting point. Points within 10 pixels of each others were rejected in favour of a more uniform distribution. The coordinates of these points were then improved upon, using the sub-pixel cornering algorithm (discussed in section 4.6) and implemented in opencv by `cornerSubPix()`

I used the pyramidal LK algorithm provided by `calcOpticalFlowPyrLK` with a window size of 10x10 and 3 levels of pyramids. In the second frame, I use the corners in the first frame as the input into the LK algorithm, and as a result, get back a list of positions where the corners have moved to. The size of this list is equal to, or smaller than the original corner list as some corners may not have been found due to rapid changes in lighting, movements or occlusion by other objects. An error is also calculated for each point - it is equal to the difference between the patch around the point in the 1st frame and the patch around the corresponding tracked point in the second image. I use this to prune away points whose local appearance has changed too much.

The process can then be re-started for each frame - the resulting tracked points are fed back into the LK algorithm on the 3rd frame , matched then fed into the 4th and so on. This gives us a nice optical flow at about frames per second for 500 points in an 500x500 image.

Finding new Harris Corners

As the player pans around, these points will quickly move out of frame, and therefore are no longer matched. These need to be replaced with new points. I first tried to brute-force it - every single frame, I cleared all the points, and found new Harris corners. Unsurprisingly, this killed performance - Performance slowed to 10 frames per second.

The elegant solution would be to find areas with a low density of corners, and then search those areas for new corners. However, the engineer in me spotted that the fps was about half, which meant the two tasks took about the same amount of time. Instead I took the simple option of threading the workload - new corners were found on one CPU, while optical flow was performed on the other. Performance increased by 80% to 18 fps.

Interestingly, the main reason why I didn't get a better performance boost, is because of Intel Turbo Boost[16] technology, which increases the CPUs frequency depending on the load

[16] http://www.intel.com/technology/turboboost/

on the CPU. When a single core of the CPU is running at a 100%, turbo boost was able to boost the core's frequency to 3.04GHz, but with 2 cores, it only went up to 2.66GHz (up from the standard 2.4GHz rating).

Estimating flow

To first see if this was a viable technique, I decided to tackle lateral flow first -i.e. when the player is still and moves the cursor left and right to pan the screen. As you can see from figure 4.37, the vector directions are pretty uniform, opposite to the direction of movement. To find vectors which did not behave in this manner, I calculated the average vector in a frame. All points with vectors that differed more than some threshold were marked as "suspect" (for the curious reader, skip ahead to figure 4.40 to see the results)

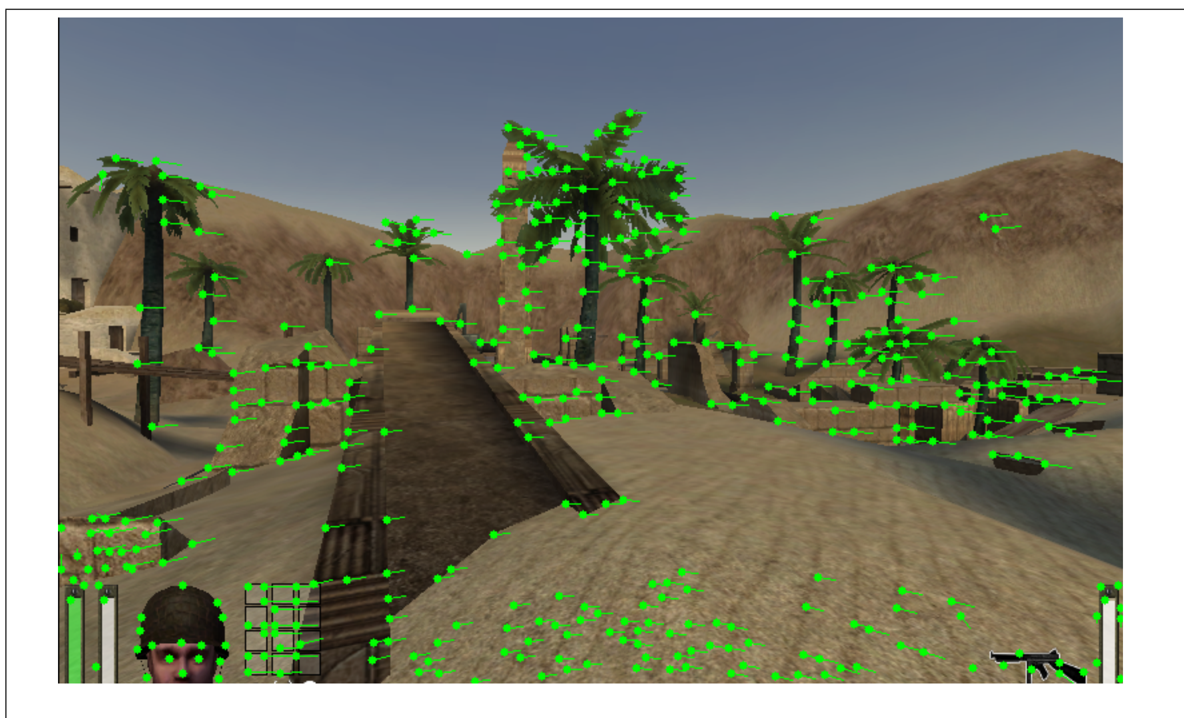

Figure 4.37: Panning - vectors are pretty uniform

Although games do not use a physical real world camera they still have image distortion "programmed" in. There are two types of distortions[17]:

Radial Distortion This arises as a result of the shape of the lens. A perfect lens should be a parabola, but in practice are not perfect parabolas. This causes radial distortions - rays farther from the center of the lens.

Tangential Distortions These arise from the assembly process of the camera as a whole

Games are of course not prone to tangential distortions. Radial distortion however, is more often than not, present in games as it is deemed a "feature". It is usually called *Field*

of View within games and frequently is user adjustable. The larger the Field of View, the larger the distortion (figure 4.38)

Figure 4.38: The left is the image when the FOV is selected to be a minimum, the right is the maximum. The player position has not changed!

The simplest and most commonly used camera model, is the pinhole camera model. Light enters through a tiny hole and is projected onto the image surface. The perpendicular distance from the hole to the image plane is the *focal length*. This can be modelled with the following equation:

$$-x = f\frac{X}{Z} \tag{4.19}$$

where f is the focal length, Z is the distance from the camera to the object, X is the length of the object, and x is the object's image on the imaging plane.

With radial distortion, the distortion is 0 at the centre of the image. Therefore it can be modelled with the first few elements of the Taylor series expansion around $r = 0$:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^2 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^2 + k_3 r^6)$$

where $x_{corrected}$ is the non-distorted location of x, on the original distorted image.

Ideally we would like to correct this. There is research into the estimation of radial distortion, but most of the methods use some pre-defined object in the view (e.g. a chessboard or a line of LEDs). With games, we don't have this option[17]. Others are semi-automatic - they require user input (usually select lines the image that should be straight).[43]

Intuitively I believe it should be possible. We know the distortion is 0 at the centre, and is *perfectly* radial, and we have a well-defined optical flow. If we say, constrict the movement to

[17] At least without creating "in-game" chessboard!

the x axis, and over a number of frames, use an iterative process to minimise the variance of vector radial distortion by modifying the parameters above, I believe we could easily converge to a good approximation of a minimum. However this is pure speculation.

As this was still classified as a prototype, to temporarily solve this problem, I chose to instead ignore vector magnitude and concentrate on direction instead. This was done by averaging the *normalised* vectors, and all comparisons were done on normalised vectors. This means that it won't be able to distinguish between enemies moving in the same direction as the general flow, even if moving at wildly different speeds.

K-Means

Now I had a set of points which didn't "fit" with the rest, I needed to group them into clusters, with each cluster representing a separate moving object. I use the highly effective but simple K-means algorithm outlined in the Background (section 4.6).

To solve the problems outlined in the background section:

No guarantee of best possible solution - I run the algorithm 5 times (with each cluster count), with different random initial starting positions, then select the solution with the lowest variance.

Not knowing how many clusters to use - I run the algorithm with 1 cluster point, up to 10 cluster points. I record the best variance recorded at each cluster count. If you plot the total variance curve, you will get an "elbow". To find the point where this elbow occurs, I draw a line from the origin, to the last point (i.e. with 10 clusters) and find the variance point that is furthest away from the line. To do this, I use the "distance from a point to a line" formula:[18].

Assuming a point (m, n) and a line $ax + by + c = 0$:

$$distance = \frac{|am + bn + c|}{\sqrt{a^2 + b^2}} \tag{4.20}$$

I use the cluster count which produced this variance point (see figure 4.39).

Does not take into account the covariance in the I-dimensional space. - this does not effect me, as the variance in both directions in an image can be treated as equal.

Once I have found the clustering, I filter out any clusters that have a low number of points (I was using 3 at the time of writing, but once again, this is an arbitrary number and should be based on other parameters such as the total number of LK points in the image and maybe image size).

[18] Here is a proof of this formula, using very basic maths: http://www.intmath.com/plane-analytic-geometry/perpendicular-distance-point-line.php

Finally, I draw bounding boxes around the clustered points, take a "snapshot" of the image at that point, and finally return the results back to the GameProfile, using the GameScript standard object table format (Appendix D).

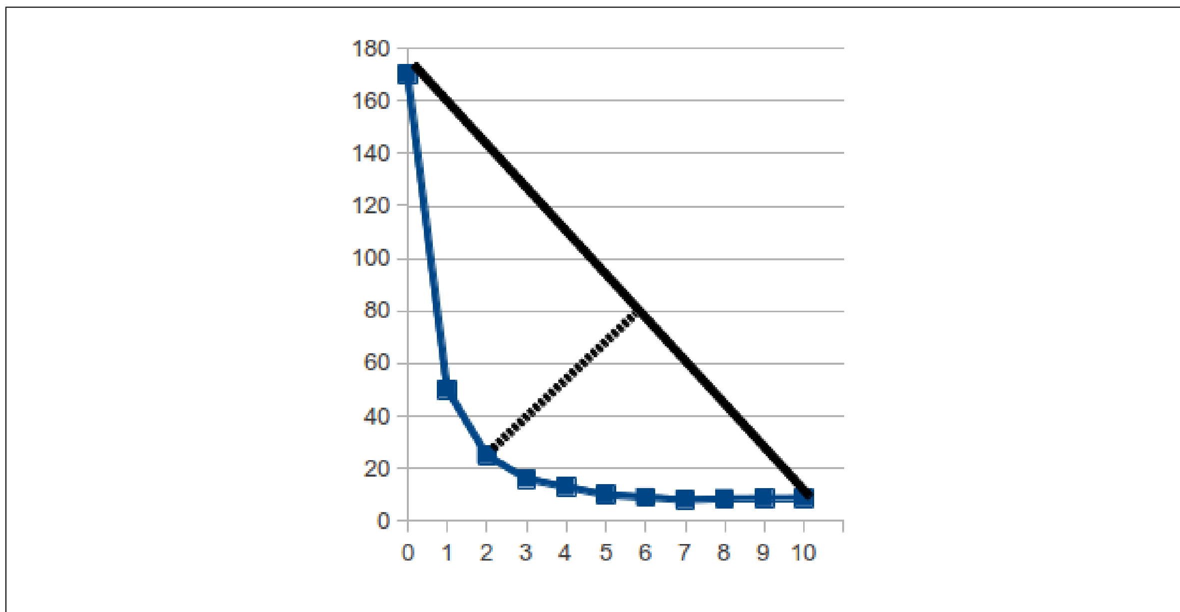

Figure 4.39: Finding the elbow of a graph, using the shortest distance from a point to a line mathematical basis. Here, 2 cluster points are chosen

Results

When stationary, the algorithm works better than background subtraction. There is no need for any learning, and so has no "down-time". Single "rogue" vectors that sometimes appear, are filtered out by the minimum cluster size threshold. (see figure 4.40)

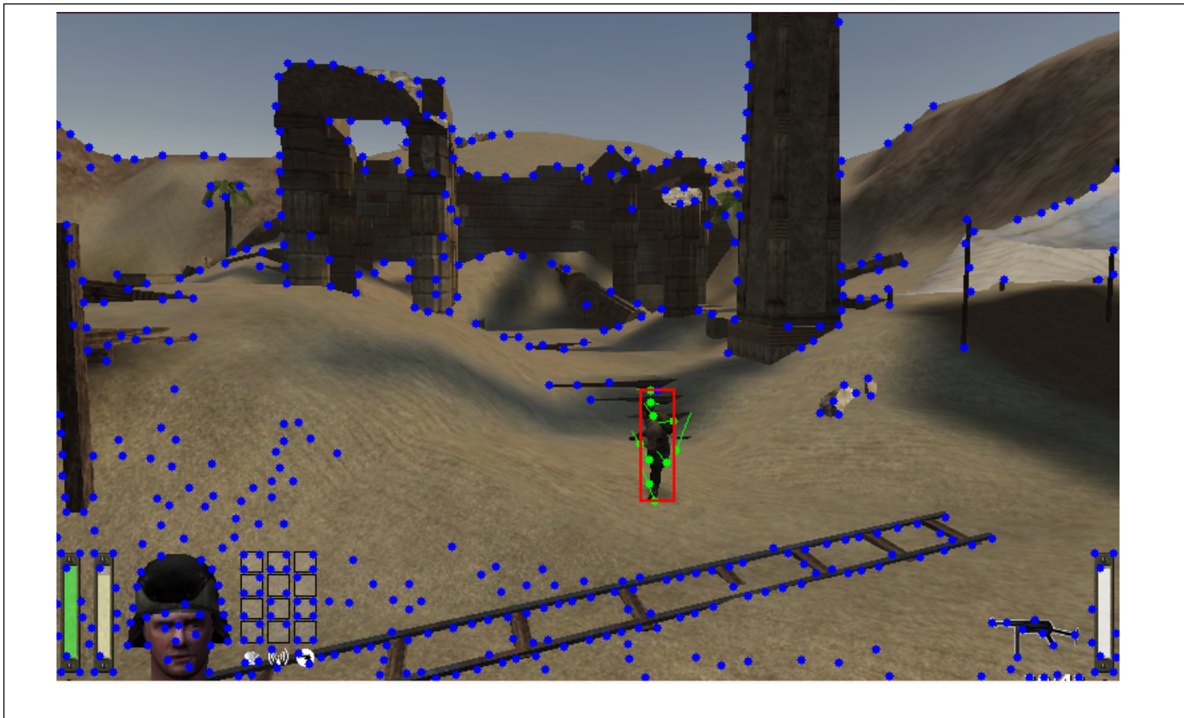

Figure 4.40: An enemy is tracked. The green dots signify vectors that don't fit within a threshold of the average

When panning, we get a pretty uniform optical flow. Enemies that run into view are most often identified and clustered, however we still get some false positives (see figure 4.41). This I found this often was because the corners used are not high quality enough - they often have 1 gradient is a lot stronger than the other, so often jump around along straight edges (e.g. the green vector on the tree - the sides of the trees are not great "corners" as they suffer from the aperture problem). Precision can be increased by requiring better quality Harris corners, but this goes hand in hand with reducing recall, due to less points being used. However from preliminary testing this doesn't seem to be too much of an issue, as Harris corners are usually much higher quality on enemies, due to well define edges.

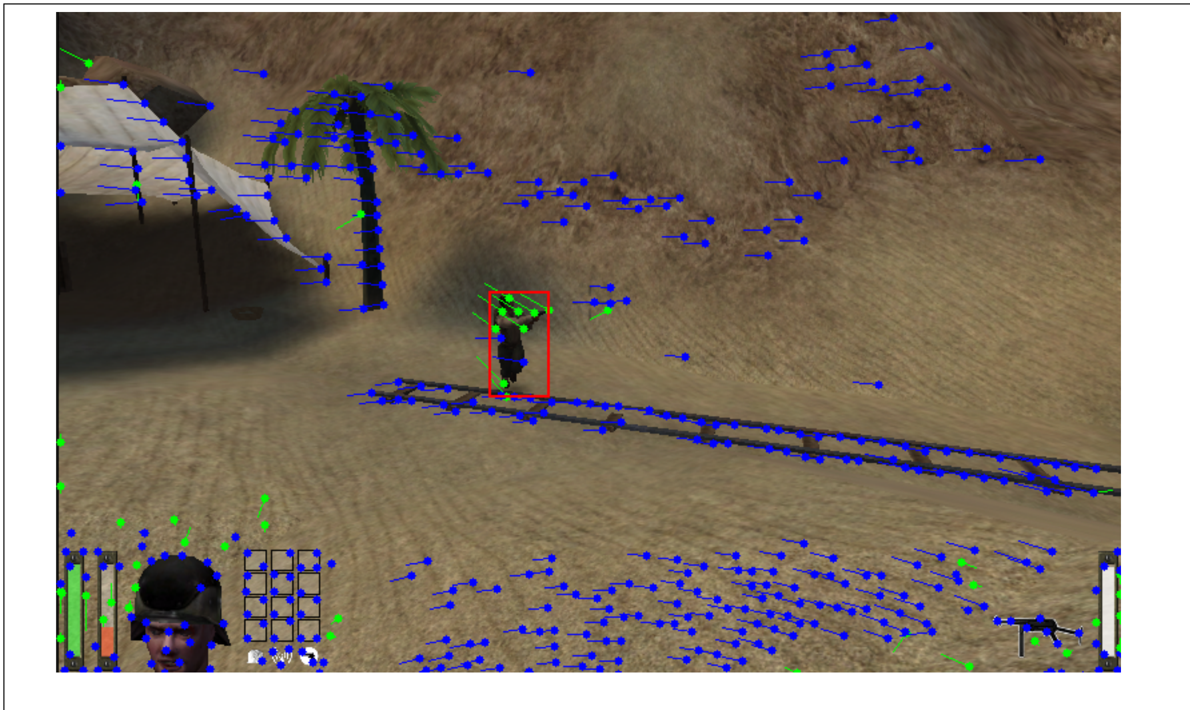

Figure 4.41: Foreground/Background segmentation

When moving, this entire algorithm stops working. This is because vectors are no longer parallel to each other, but all point to a "Focus of Expansion". Moving foreground objects also have their own focus of expansion - the focus of expansion is the single point on the projected image where the object appears to be coming from. (figure 4.42

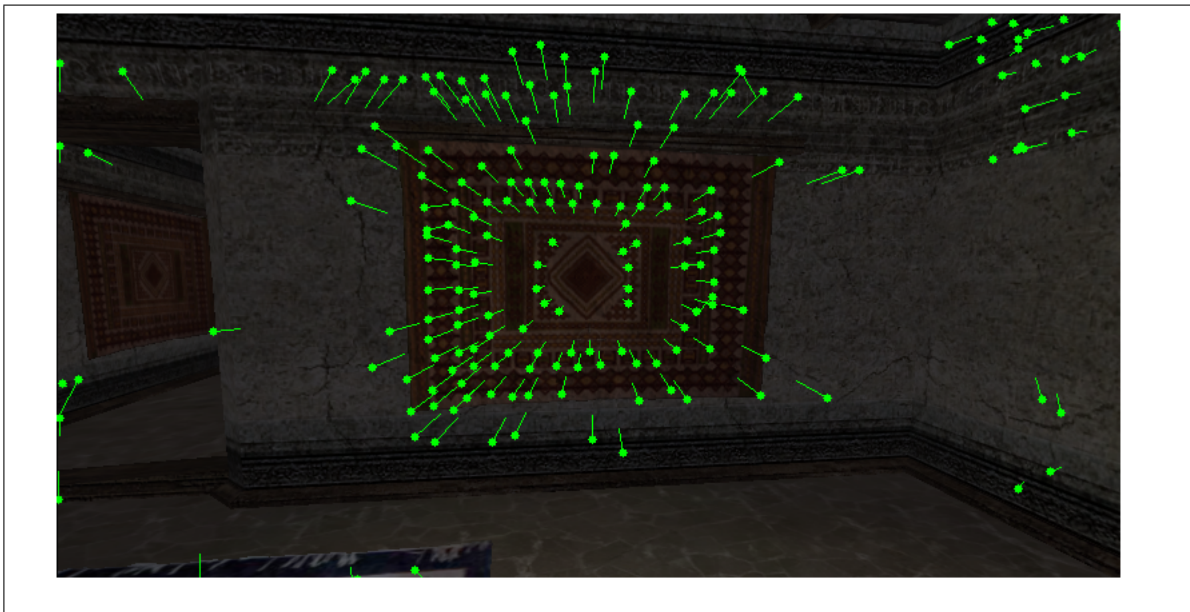

Figure 4.42: Focus of expansion - all (accurate) vectors on an object should all point to a single point in the image plane

Another interesting movement is the "lean left", "lean right" movement. This rarely features on newer FPS games, but it actually provides vectors that "rotate" around the middle of the image, as can be seen in figure 4.43. This is the gives the same effect as camera rotation in real-life.

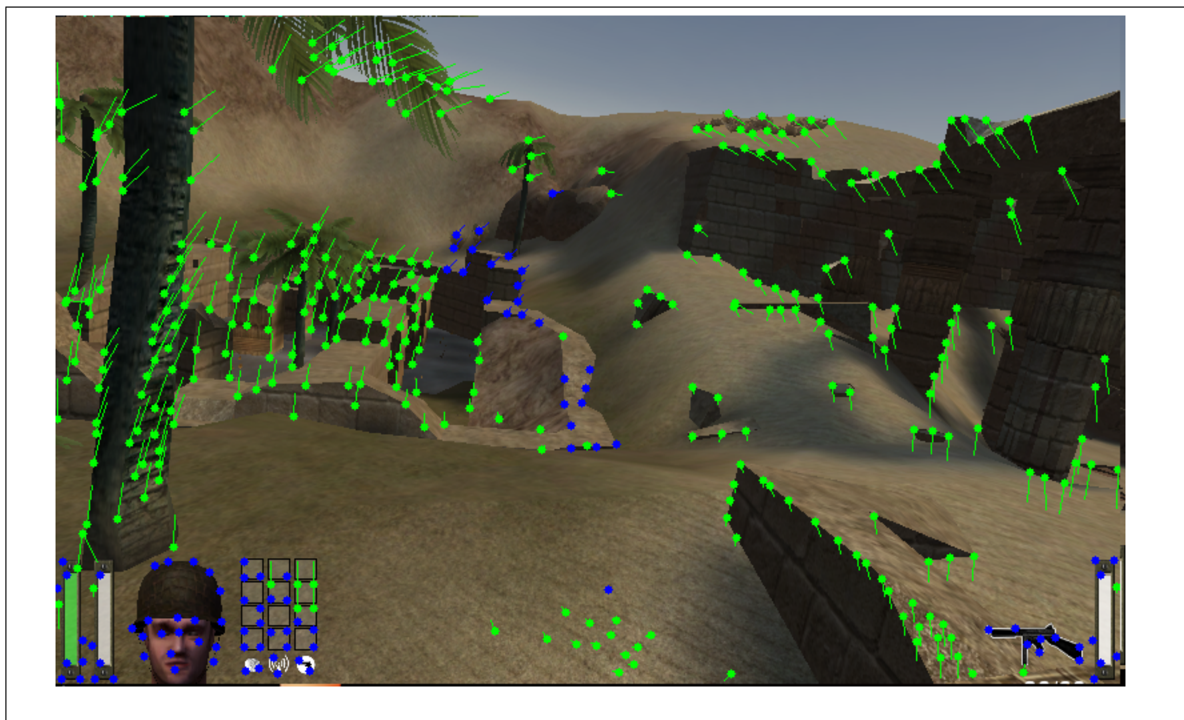

Figure 4.43: F

Further improvements

Performance

One issue I came across was that game-play in FPSs is very fast, and panning speed is often too fast for the LK algorithm used. This means that during period of fast panning, most Harris corners have been lost, and the few that remain are usually anomalous (figure 4.44).

Figure 4.44: Fast panning - most Harris points are lost and the ones that remain are anomalous

To solve this, we can do 2 things:

- Increase the window size for LK matching.
- Increase the frames per second that GameScripter runs at.

The problem with the first choice is that increasing the window size for LK matching actually made it *worse*. This is because the matching is not done "offline", but in real-time. Increasing the window size, decreased the fps that GameScripter ran at and consequently, the time (and therefore translation) between the consecutive images increased. Increasing the window size from 10x10 to 15x15, increases the search space by 225%, so theoretically performance should decrease by the same amount. A pixel can now travel 5 pixels further before being lost - this is only a 50% increase. This is counter-productive as in a uniform panning motion, the pixel will travel 225% further in a single frame, but can only be matched at distances 50% higher only! A similar argument can be used to justify decreasing window size to *increase* the tracking accuracy.

In practice, I found that the speed limiting factor was actually the finding of Harris corners. I remind the reader that in my implementation these are done in parallel and so the fps is the minimum of these 2 task. Decreasing the the number of corners resulted in a speed benefit for LK tracking, but none in corner finding. Fortunately both these tasks lend themselves to parallelism, as they assume that the points are independent. Therefore, both tasks should be gain linear performance benefits with the number of cores, by splitting the

image into sections (with overlapping sizes equal to window size) and running the algorithms concurrently. Results can be then merged together without needed to worry about conflicts.

Arbitrary Motion - Speculative Future work

Using average vectors clearly doesn't work in the general case. Instead we need to find the focus of expansion (FOE), and then we can mark all vectors that do not pass through this point as "suspect". I can think of 2 ways of achieving this:

Voting in Hough Space Finding the intersection of many lines reminds me of the process of voting in 2 dimensions when implementing the Hough transform[19] in $m - c$ space. Of-course this has the same drawbacks - the parameter space is infinite. Therefore we could do it in $r - \theta$ space, where parameters a constrained.

K-Means with pair-wise vector intersection We find the points of intersection of all combinations of vectors. Then we can apply K-Means clustering, and find the largest cluster to be the FOE. Also has the benefit that smaller clusters may be indications of other objects, moving in a different direction!

Both the above methods are very slow, and unlikely to be able to run well in real-time. For example, for the K-means method, there are ${}_nC_2$ points, which must be found, then clustered (for 500 points this is 124750 points!). However for the purposes of an FPS, we are not trying to get very good accuracy - we only need an estimate of the FOE as we are using arbitrary thresholds anyway to mark vectors as "suspected". Therefore I believe a probabilistic version of either of the 2 methods listed above would work well, where we only sample a small part of the population until we have a good enough estimate (i.e. a low enough variance).

Arbitrary Motion - The GameScripter Way

The above are numerical techniques, with a basis in mathematics to find the FOE. I however think the most exciting and novel way to find the FOE is to not use the optical flow itself, but calculate it from the players movement. This should be completely possible from within the GameScripter environment, and does not require API access to the game. We can record the players actions since the X frames, and estimate where the FOE should be!

There are 3 assumptions:

1. The in-game protagonists movement in relation to the game world is defined *only* by the players input actions
2. The in-game protagonists movement is deterministic.
3. The fps is fast enough so that cursor movement between frames can be approximated by a single line. This assumption is there to simplify the algorithm, rather than a necessity.

[19]http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm

The above assumptions are pretty major, and there are obvious flaws. For point 1, a player could pushed by another player, and therefore movement that hasn't been recorded through the IGameInputMonitor. For point 2, the terrain has a big say to the players movement - if a player is trying to walk into a wall, he/she of course will not get anywhere. However in the general case, where the player is walking/running on the ground, these assumptions should hold.

Let's for example take a simple case where at frame i we wish to find the FOE. We inspect at all input data between frame $i-1$ and i and find during that time, the user has held the "w" key pressed (which represents forward movement), with no other input. We can immediately identify the FOE - the centre! No extra processing is necessary! If the inter-frame time is short, it may be necessary to hold more history of the user input, but this does not change the idea.

The next example uses assumption 3 - Since frame $i-1$, the cursor position has changed by the vector (x, y). This therefore means the optical flow vectors should have a direction of (x, y). If there is no radial distortion, all vectors should also have a magnitude of $\omega \cdot \sqrt{x^2 + y^2}$. ω is a weight that could be found very simply by experiment - in a single frame, divide the average of n optical flow vector magnitude, by the cursor magnitude $|\vec{v_c}|$:

$$\omega = \frac{1}{n} \frac{\sum_i^n |\vec{v}|}{|\vec{v_c}|}$$

If ω is not linear (i.e. it is a function of the cursor vector magnitude), more experimentation can be done over many frames to estimate the function. This is often the case in FPS games as they use what is called "mouse acceleration", where small movements give accurate small cursor changes, but larger movements give very large cursor changes, so that players can quickly turn around in-game.

Lastly, and most interestingly, it should be relatively easy to handle both simultaneous keyboard and cursor movement. The FOE could be calculated by the addition of vectors, which we have shown can be calculated from the users input. This is shown in figure 4.45

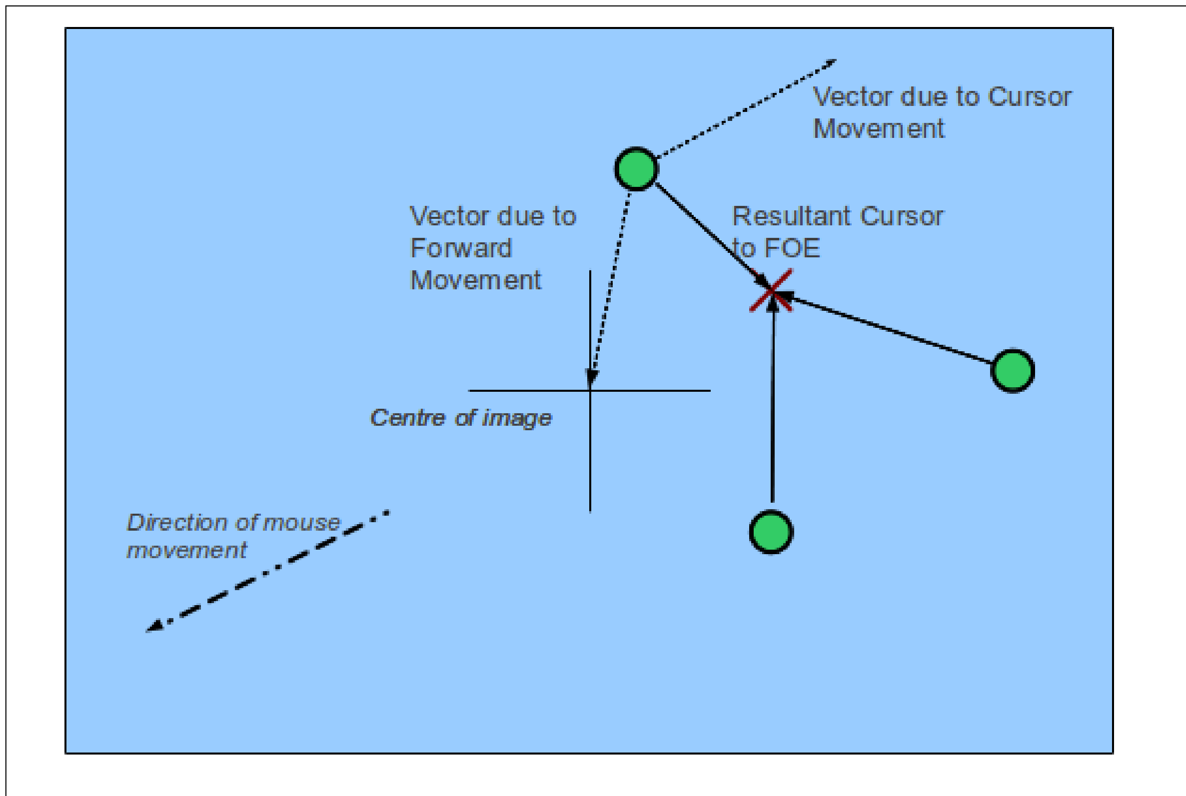

Figure 4.45: Adding together the vector from moving forward and the vector resulting from cursor movement (the dotted arrows) results in the vector towards the FOE

Even if it turns out that this technique is not completely accurate (or mathematically sound), it could be used in unison with the aforementioned Hough space and K-Means techniques, to reduce the search space.

4.7 Real-world Application

In this section, I investigate the use of GameScripter not on games, but on the real world. Modifying the software to work with a camera was extremely simple because of my modular design - all that was needed was a concrete implementation of IGameOutput (section 3.2:Overview of System Architecture) that supplied images from the camera rather than from a games console. Of course a concrete implementation of IGameInput was also needed, but the functions all just threw an error - you can't control real life from GameScripter![20].

Task

As a short demonstration, I chose a task that is an active area of research right now in computer vision - car counting in computer vision. I chose this because data was easy to come by - I was able to take a video from my bedroom window.

Desired GameProfile

The idea behind the profile is that once we identify a car, we track its movement through the frame so as not to count it again. The identification of a car uses background subtraction - all the foreground objects larger than a certain size are assumed to be cars (the smaller ones usually were caused by people walking past).

```
numOfCars = 0

function callOuts.newframe()
   objects = vision.getForgroundObjects()
   for i, obj in ipairs(objects) do
      if obj.area > 5000 and obj.isTracked == false then
         vision.trackObject(obj)
         numOfCars = numOfCars + 1
         print("Found new car, running total: " .. numOfCars)
      end
   end
end
```

We introduce a new call-in :

`vision.trackObject(obj)` - Tracks the object passed in the argument. This object which can then be queried at any time using `vision.getObjectPosition(obj)`.

I also added another call-in (although it is not used in the script):

`vision.getObjectPosition(obj)` - Returns the GameScripter standard object table (Appendix D) if obj is a tracked object, otherwise nil

[20](yet)

Implementation

I chose to build upon already existing techniques in implementing `vision.trackObject(obj)`. The algorithm is very rudimentary, based on my own experiences with computer vision, not on any papers or mathematical basis. It assumes the object does not change much between frames, but tries to adapt over longer periods of time to an object's changing appearance (such as colour, shape or size).

Initialisation On the initial call to `vision.trackObject(obj)`, corners are found *within* the objects boundary and are associated with the object as the object's "tracking points". A colour histogram of the object, it's contour and it's size is recorded.

Every Frame For each object that is to be tracked:

- Using Lukas-Kanade, try to match the inner-corners to the new frame. Record the number of corners found **A**, and the average error **B**
- Test **T**: Perform background subtraction. If there exists a connected component at the mean position of all **A** corners:
  - Find the ratio between the size of the last recorded size and the current size (**C**)
  - Match new histograms using techniques already described with the last recorded histogram. **D**
  - Match new contour using techniques already described with the last recorded contour. **F**

Now we work out a measure of how well the LK optical flow has matched the object (i is the current frame):

$$\omega_1 \frac{\mathbf{A}_{i-1}}{\mathbf{A}_i} + \omega_2 \mathbf{B} \tag{4.21}$$

Each ω_1 and ω_2 are a predefined weight. If this is below some threshold, we can say that we have successfully tracked the object, otherwise we have lost it, and remove the object from the tracking queue.

If test **T** fails, we just keep track of the inner-corners and use their average position for object position.

If test **T** passes, check if $\mathbf{C} < someThreshold$ before carrying on. This is checking if the object is still about the same size as it was before. If there is a massive jump, it is likely that it has overlapping with another foreground object.

If $\mathbf{C} < someThreshold$ then match histograms and contours. Calculate the following:

$$\omega_3 \mathbf{D} + \omega_4 \mathbf{F} > someThreshold \tag{4.22}$$

This threshold should be quite high, so we can be sure it's a good match. If this test passes we are pretty sure this is the same object. This means we can redo initialisation - i.e. find new inner-corners, update the stored size, update the stored histogram, update the stored contours.

Results

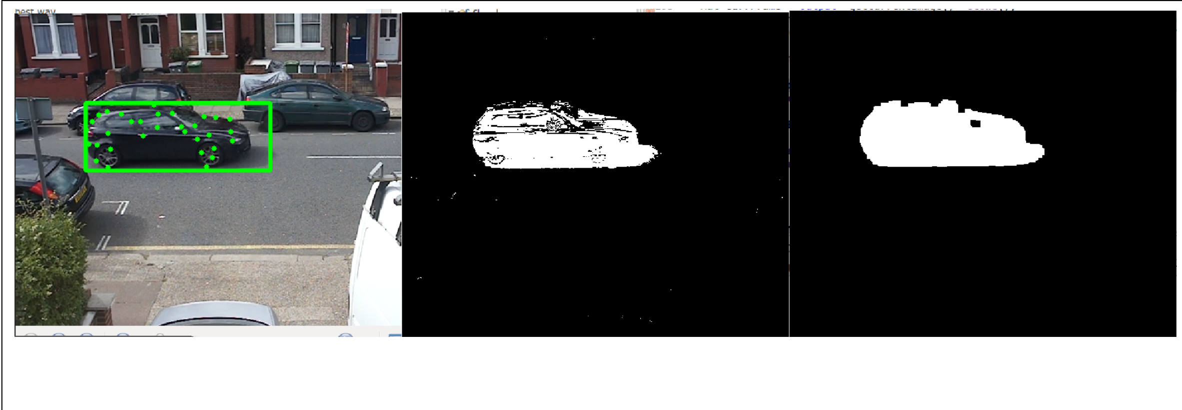

Figure 4.46: The car is tracked using LK and corners. The middle is the raw background subtraction while the right is the closed then opened version

The script was successful. In a 25 minute video I took from my bedroom window, it attained 67/67 cars - a 100% accuracy rating [21]. The re-initialisation part of the algorithm meant that it didn't matter if some of the feature points that were tracked were lost (or "attached" themselves to background instead), as they were often reset.

For the algorithm to work well, I found that I needed to set the contour matching weight very low. Contours deform quite a bit between frames, and are especially deformed by the closing/opening morphological operators. In the future, it would be interesting to investigate some machine learning algorithms, to dynamically adjust the weights, or even use an alternative to the weighting system! This is discussed in section 6.2:Future work

However there are many obvious flaws with the algorithm. When two cars pass each other, the closer one often occludes a portion of the other one. When this happens, the algorithm resorts to using just feature points to track the objects. Often, the algorithm recovers once the silhouettes become disjoint again because the sizes of the vehicles haven't changed much, which triggers a re-initialisation step. However this is only true because of the angle of the test video. From a lower angle, one car could completely cover the other one, and tracking would be lost as the algorithm relies only on tracking between adjacent frames.

Overall, I don't think this algorithm will generalise well. I would have have liked to test more advanced tracking techniques such as mean-shift, but time was short. I believe I would be able to get better results than this my home-grown algorithm.

Overall, I think it was a success in demonstrating that algorithms that have been developed on games can be transferred into real-life!

[21] Yes I watched it all... not the most interesting 25 minutes of my life

Chapter 5

Evaluation

I have the discussed the GameScripter's efficacy and user-friendliness on a game-by-game basis in chapter 4:GameProfiles. In this section I try to summarise the points conveyed in the that section, and review the success of the project as a complete application, including it's strengths and weaknesses. Like in the previous section I try to add some quantitative analysis, but with nothing to compare against, this is difficult.

5.1 Case-study

One of major project goals was making GameScripter easy and accessible to end-users. No evaluation would be complete without testing this theory in real world, hence I recruited of two of my friends: Jack, a seasoned Imperial College DoC student, also in his fourth year and James, a musician who doesn't understand the difference between google and the internet, let alone written a single line of code. I gave them the task to create a bot that could play a game called **Volley Brawl** . The rules of the game a very simple and can be figured out from Figure 5.1. You control a simple character, and you have 4 controls - up/down/left/right (although the key-scheme is W/A/S/D) and are playing a game of volleyball against a computer opponent (or another human player).

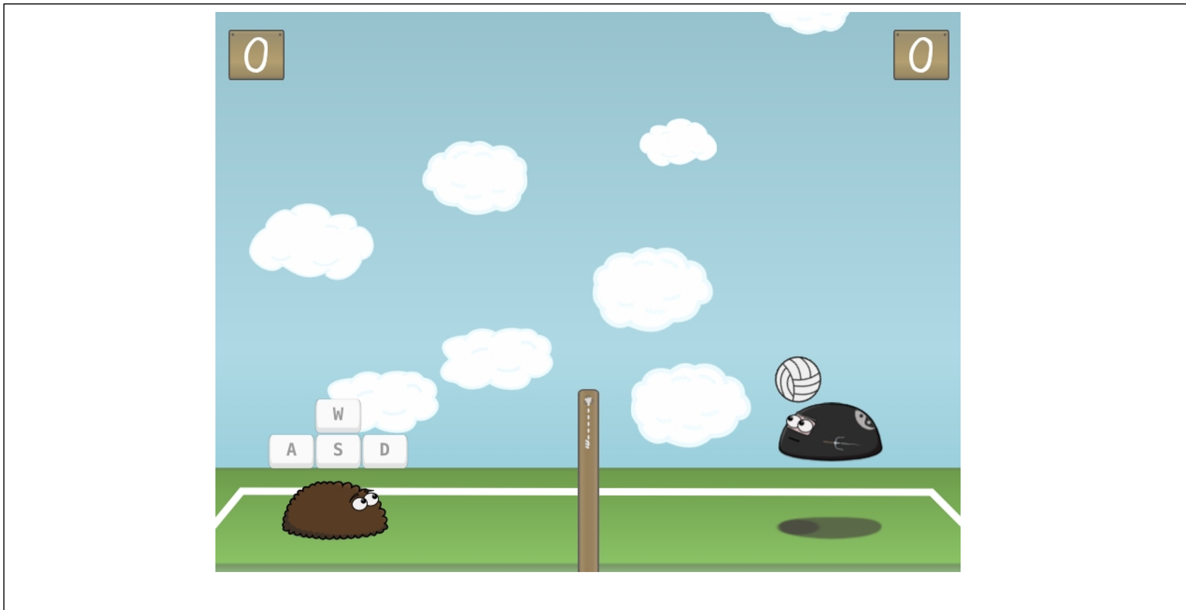

Figure 5.1: Volley Brawl: a ”slime” volleyball game

I left Jack on his own, with the documentation in the appendices of this report, but I sat by James so I could answer any of his questions, but refrained from giving him direct help. The different approaches taken were interesting:

James

- James's biggest problem was that he didn't know where or how to start. When I provided him with a skeleton file, defining the parsing box boundary and a few other finished examples, he had no problem in proceeding.
- At first I though James was going to take a simple template-matching scheme, because he was taking screenshots of the slimes and the ball. But what he did instead is only use template matching to find the initial positions of the ball and then actually used `vision.trackObject()` to track resultant object. When I asked him afterwards why he didn't keep template matching every frame to find location of the ball and slimes, he said it was more ”natural” to track it, as he could just query the object's location at any point.
- James didn't use any user defined functions or other call-outs - everything was done in `callouts.newframe()`
- The control scheme was very rudimentary - but surprisingly effective - he followed the ball everywhere, but tried to keep 15 pixels to the left of it, so that the ball hits the side of the slime when it falls (thereby pushing it over the net instead of straight up).

  Controlling the slime is more complicated than just moving a mouse, as you need to keep track of the slime too. The main problem was the call `keyPress()`. James at first thought that it meant a key press and a key release. When he wanted the slime to

change direction, he never released the key, and so the slime wouldn't respond. Once he figured it out though, he had said it was not a nice solution, having to "remember" which keys had been pressed and not yet released.

Jack

- Jack went for an entirely different technique for tracking the ball and slime - `getForegroundObjects()`. However instead of using relative sizes of the objects to classify the objects (as I did with Breakout - section 4.4), he used relative positions - the ball will always be higher than the slime. The clouds you see in the figure are not stationary - they actually move at a very slow pace. Jack filtered these out using the minimum size criteria (as because they moved so slowly, most of the clouds actually blended into the background, with only the edges coming up in background subtraction)
- Jack also used `vision.excludeRegion()` to exclude the region where the opposition slime was in, so that it didn't come up in the background subtraction.
- Jack went a step further than James and tried to predict the trajectory of the ball, however this was not too successful, because the ball's trajectory was not linear. In the end, he settled for a similar control scheme to James, except with an extra if statement that would make the slime jump if it was in a certain position in relation to the slime. He also complained about the control scheme of `releaseKey()` and `pressKey()`. He wanted to be able to specify how long to press the key, before releasing (in milliseconds).
- 1 really good idea Jack had was used the `callouts.keyPress()` to bind keys to increment and decrement variables in his program, such as the ideal height of the ball at which the slime should jump. This meant that he could watch how they affected game-play, rather than having to change the number in the GameProfile and reloading every time.

Both scripts faired well the early stages of the game, however as the game progresses and the ball speed increases. In these situations, Jack's GameProfile was better, as it was able to jump to intercept balls, whereas James's slime was not always fast enough to follow the ball along the ground.

All in all, I think I learnt from this experience. The current documentation, combined with some examples, is just about acceptable for people with programming experience, but a good tutorial would be useful for those without. The `pressKey()` and `releaseKey()` call-ins were not well thought out, and need some modification (to include an optional timed release). Also the idea of using increment, decrement keys I thought was a great idea, and I believe I could take this a step further, by providing a language construct that simplifies the task - `bind(key, variable, increment)`.

Lastly, I hope that you can see that using a DSL has made the software more accessible than it would ever have been with a more complicated programming language.

5.2 GameScript

The key success of this project was the use of a Domain Specific Language. GameScript is what moved the project from being just a piece of exploratory programming, into a full featured product that can be used by an end-user.

GameScript makes GameProfiles surprisingly effective and very easy to write. Users usually know what they want to achieve, but not sure how. The call-in, call-out structure makes it intuitive, as it guides the user through the process. They do not need to worry about any complicated language features such as type systems or memory management. Even simple concepts such as variables do not need to be explained more than "places where you can store stuff", as I did with James.

Another major advantage of GameProfiles are kept completely separate from the core program - they are just text files. There is no compilation phase and they can be dynamically loaded and unloaded, without shutting down the application. They can be modified while running, and then reloaded at the press of a button, giving way to a very fast iterative scripting style and fast feedback. This is very important to the end-user and even benefited me, the developer, because it meant that I could batch test many scripts quickly and easily when testing for regressions.

GameScript vs C++ Comparison

Here, I compare the speed of a sample GameProfile written in GameScript with one written in pure C++.

I have chosen Generic Breakout and Whack-A-Mole as the benchmarks. As with most GameProfiles, no performance-heavy algorithms are executed in the GameProfiles themselves unless the user decides to implement a complicated AI in GameScript itself. Therefore this example should be indicative of overall performance loss of using interpreted code and passing values and calling through the Lua stack (see section 3.1:GameScript as a Domain Specific Language using Lua).

I ran both version 10 times, each time taking the average of 1000 frames.

	Whack-A-Mole		Breakout	
Game	Average FPS	Standard Dev.	Average FPS	Standard Dev.
GameScript	10.31	0.04	34.52	0.10
C++	10.31	0.04	34.56	0.11
Improvement (%)	0	-	0.12	-

These results show that there is a negligible difference in performance using GameScripts and pure C++. This is unsurprising, since there is not much communication through the Lua Stack - per frame there is only 1 call-out, and 1 or 2 call-ins. The overhead of this is tiny compared to the very performance heavy computer vision algorithms. Even the apparent 0.12% speed increase is dubious, and is within experimental error (interesting to note that although S.D. was low between runs, S.D. between frames was high - 0.38 for Whack-A-Mole and 3.0 for Breakout).

Disadvantages

There is however much room for improvement. The major disadvantage of GameScript is the upfront cost of implementation. Each new piece of functionality that I added to GameScripter needed to be wrapped around in code to correctly expose it to GameScript. This often was not a trivial task, involving the design of new types and more complicated memory management. This is because I could not discard data that a GameProfile may still wish to use.

On the plus side, all the extra work of wrapping code made GameScript more reliable and safer to use. This code isolates the GameProfiles from the core C++ code which means the user is not be able to crash the program as a result of bad coding. Instead the Lua interpreter fails to interpret the GameProfile correctly, leaving the core functioning correctly and can inform the user of the error. There is no need to worry about problems like segmentation faults or null pointer exceptions.

Memory Management I made the incorrect decision to have memory management fully managed by the C++ core. At the moment we have call-ins such as `vision.getImage()` or `vision.getHandle()` that return a *handle*. The core however, has no way of tracking which handles are still in use so that the memory can be reclaimed - it has to keep *all* of them in memory. This has not been a problem, as these are called sparingly, but an inexperienced end-user may have a faulty GameProfile that can quite easily cause GameScripter to run out of memory by having a loop that makes thousands of handles.

The solution is to place memory management in hands of the Lua interpreter, which is garbage collected and use Lua userData. The handles are then no longer just pointers, but areas of memory, where full objects that can reside and can be manipulated from both Lua and C++. When the GameProfile no longer uses one of these objects (i.e. there no longer exists a reference to them), they can be easily garbage collected.

GUI support Another, smaller disadvantage is the loss of GUI-based help during coding. Integrated Development Environments such as eclipse[1] help the user by providing auto-complete hints and documentation on demand. This would be very useful for an end-user with GameScripter, as it lessens the need to refer to documentation.

5.3 Efficacy of vision

5.3.1 2D Object recognition

The individual efficacy of the vision techniques used in GameScripter has been extensively covered in chapter 4:GameProfiles. Overall, in 2D games, object recognition has often achieved 98+% recall rates while keeping precision rates at 100%. This is entirely due to the nature of 2D games. 2D games usually have very simple visuals, with no lighting changes or motion blur. Objects are often just images that are just translated across the screen,

[1]http://www.eclipse.org/

making finding them very easy using extremely simple techniques such as template matching (section 4.1) Sometimes, some objects can be rotated (Figure 4.21) or a scaled larger or smaller. As these transformations do not actually change the object shape or colour - they can still be easily detected, the only difference is the search space explodes rapidly leaving techniques such as template matching computationally too slow. This means further techniques such as background subtraction (section 4.4), edge detection (section 4.5) and contour matching (section 4.5) can be employed in unison to produce scale and rotation independent ways of matching objects in real-time.

5.3.2 3D Object recognition

3D object detection becomes much harder; my initial attempts to use object cascade classifiers were unsuccessful (section 4.6). This is not an indication that real-time object detection is impossible - there have been numerous successful demonstrations of these techniques working successfully in real-world situations. However these all used very good training sets, often hand-sorted and verified - something I could not do.

I hypothesise that, with the use of some preprocessing, Voila-like cascade classifiers could be adapted to be of use in a 3D game environment. Positive images gathered during game play could be segmented into separate classes based on other features before training phases, with each class having it's own classifier. Edge detection and watershed[2] techniques could be used to find the object's outline contour, leading to correct and consistent cropping of images. Images could also then be classed by contour similarity before being trained on. For example the profile of a helicopter from the front would be vastly different from a side contour. The question is that if we are already able to segregate images well into these separate classes, is there any need for cascaded object detection in the first place?

5.3.3 Background/Foreground Segmentation

Background/Foreground segmentation in 2D was a relatively simple task, with even simple frame subtraction working well(section 4.4) for 2D games. The more advanced codebook and mixture of Gaussians also worked well in 2D, but had the advantage that they also worked well in 3D environments with little to no changes needed (section 4.6). However neither were of any use when the viewpoint was moving. Although there seems to be research into the field of background subtraction with arbitrary camera motion, I did not have time to investigate this area.

5.3.4 Optical flow

The prototype I created showed that it is possible to get some decent background/foreground segmentation using sparse optical flow, in a real-time environment (section 4.6). My implementation was flawed, as it could only deal with lateral movement - not forward or backward movement. However I explained why and proposed a number of solutions, including a novel algorithm to calculate the focus of expansion without the use of optical flow.

[2] `http://cmm.ensmp.fr/~beucher/wtshed.html`

I also used optical flow as the core of my novel object tracking algorithm (section 4.7:Real-world Application). It was a combination of many of the researched techniques in this report, and tried to use the best characteristics of each algorithm, while avoiding the drawbacks.

5.4 Performance

I have already discussed in-depth the performance of many of the features as they were introduced. Overall, most algorithms were single-threaded and performance was satisfactory for most tasks. Only when it came to optical flow was I slightly hindered by performance. Part of the reason may stem from the fact I chose algorithms in advance I knew could be used in a real-time environment (e.g. sparse optical flow vs dense optical flow 4.6). Adding a GPU gave an order of magnitude increase in performance in areas they were implemented in (section 4.2:Implementation). That said, I feel there massive performance gains that could be attained by tuning algorithms to be more suitable for GameScripter. Many areas were outlined, but template matching was one I believe could be most improved (section 4.2), along with parallelising the LK implementation.

5.5 Game Applicability

For GameScripter to be successfully used on a game, it needs to be able to accurately and easily parse the game state and game events. In the GameProfiles section, I have shown that for many games this is possible. The reason for this is that all the game state (or at least the game-state we care about) is displayed to the player - in **Breakout**, all game state can be parsed in a single screen-shot, the same being true for **Tetris**. There is little advantage of having an API hooked into the game, as state through vision produces near identical results. However this is not true for many other game, which leads to difficulties.

The simplest game I can think of where game-state cannot be parsed simply, but requires "memory" is **Concentration**[3] (also often know as **Pairs**). This is a card game in which cards are laid face down on a surface and the players take turns to flip face up 2 cards. If the cards match, they are removed from the playing field, otherwise they are turned back over. The object of the game is to remove all pairs of cards from the table. Here game-state cannot be parsed at any one point in time by scraping the screen. For this game it would be fairly simple to write a GameProfile, as GameScript supports the saving of objects, and their positions through function such as `vision.getImage()` or handles, however it demonstrates the need of having to store game-state in the GameProfile itself. As games get more complex, this need gets to a point where vision technique severely fall behind API access.

Real-time strategy (RTS) games are an example where parsing game-state just through vision is all but impossible. In an RTS, players position and manoeuvre units and structures under their control around areas of the map usually with the aim to destroy their opponents' assets. The player only has a small window of view of the entire playing field at any one-time (see figure 5.2). Some state be parsed fairly simply, such as building and unit positions, but

[3] http://en.wikipedia.org/wiki/Concentration_(game)

only relative to the current view-point. Getting the absolute position on the playing field is considerably harder although could be achieved to a certain extent through the use of mini-maps. If you refer to figure 5.2, you can see the mini-map in the bottom-left hand corner. The little trapezium displays your current view onto the entire playing field. By parsing the current view with reference to where this trapezium is, one can get rough estimate of the absolute coordinates of buildings and units in view

However, this is a tiny portion of state in the entire game. How do we parse unit way-points, a building's current production queue or even the terrain itself. These problems are usually a non-issue with a game API, where these can easily be queried through simple functions such as *getBuildQueueForBuilding(buildingID)*. Even if we do not take into account the fact that game AI severely falls behind human players at such games already[44], I predict that a computer's ability to play games such as these through vision techniques will not be viable in the near future.

Figure 5.2: Starcraft 2 - currently the world's most popular RTS. Only a subset of the entire field can be seen at any point in time. The mini-map is displayed in the bottom left corner

Chapter 6

Conclusions

I have shown that, by exposing computer vision techniques with a simple domain specific language, I have been able to successfully produce an novel, easy to use tool to parse game state and script game-play. Many 2D games can be scripted in less than 10 lines by people with little programming experience and play the game perfectly (section 5.1). I have also shown that this approach can be extended to augment some 3D games (section 4.6). However, I have also investigated the limitations of such an approach - not all games are susceptible to parsing of game state with vision, because much of may be ”hidden” from the player.

The use of the simple scheme of call-ins and call-outs, has provided a friendly way of guiding a less technical user in writing GameProfiles. The up-front cost of implementing such a system is easily mitigated in the long run, both to the end-user and the developer (section 5.2). The end-user benefits as it allows them to think about the problem itself, rather than worry about writing any boiler-plate code or the need to know any complicated programming theory such as type systems or memory management. Any errors in GameProfiles are usually the result of faulty program logic rather than misunderstanding of language nuances. By removing the need for a compilation stage, even the developer benefits during testing. All this is achieved without sacrificing performance 5.2. I can see this technique being extended to other applications in a similar fashion, exposing functionality in a safe and simple manner.

I believe I have done something that no-one else has done before, and apply vision techniques on games themselves rather than on tangible, real-world images. Through this, I have demonstrated that computer vision techniques used in the real-world can be very easily modified to be used in computer games. In fact many algorithms can be slightly simplified as you no longer have to worry about imperfections in cameras and camera lenses or poor data, resulting in a quicker (and more enjoyable) development. Although on the face of it, the usefulness of in-game vision to research is not obvious, in section 4.7 I believe I have also shown that techniques learnt from vision in games can be adapted back to real-world scenarios.

6.1 Things I have learnt

Through-out this project I have come to appreciate the value of thorough background research as it helped me make full use of existing solutions that were indirectly related. I benefited greatly from getting a overall "feel" of the major areas of computer vision before starting implementation. I needed to do this because nothing existed even closely similar to my project, which I could base development on, or compare to. When I came across a new problem, I was quickly able to narrow down applicable areas of computer-vision to research. I made excellent use of the open-source OpenCV library allowing me to have prototype algorithms working very quickly, aiding my iterative approach to programming. This, as a result, vastly broadened the scope of my project. It has surprised me how many very simple ideas, such as image thresholding, blurring and image subtraction can be built upon each other to get really impressive results.

6.2 Future work

In this section I outline possible future work. First, I would go about tackling a lot of the limitations, such as performance and the GameScript shortcomings outlined in the Evaluation and GameProfiles, before going to implement new features. The particular part of the project I would most savour working on, would be to implement and test some of the outlined optical flow techniques from the first-person shooter GameProfile section (4.6). It was unfortunate that this was but a small part of my project, and I could not put more time into that area. Now that the GameScripter framework is well established and standardised, I would be able to concentrate more on the underlying algorithms and less on the framework itself.

6.2.1 New Features

There are also a number of new features I would like to see in GameScripter:

Hook-ins At the moment we have call-ins and call-outs - I propose a 3rd type of interface - Hook-ins. This is a method of associating a GameProfile function with an event. Instead of having a single function that is called when an event happens (e.g. `callouts.newframe()`), we can split up game logic into separate functions. This has the benefit of making Game-Profiles more modular; one script can easily build upon another without having to modify the original GameProfile. An example is given below, which can extend *any* GameProfile to support pausing and starting of background subtraction, without modification to the original profile.

```
loadScript("OriginalProfile")

hookins.associateKeyPress("p", pauseBackgroundSubtraction())
hookins.associateKeyPress("s", startBackgroundSubtraction())

function pauseBackgroundSubtraction()
```

```
  vision->pauseBGS()
end

function startBackgroundSubtraction()
  vision->startBGS()
end
```

Higher layer of abstraction At the moment there exist many ways to match objects - contours, histograms & templates - the user must decide on the best methodology (or a combination of methods). I would like to add another level of abstraction on top of this, and amalgamate all these techniques into a single "super" `findWithFB()` call-in, which stands for "find with feedback".

This would use a weighted combination of all the above techniques, with the possibility of background subtraction and cascade classifiers too, to find the object. Of course the system would not know what weighting to use, so initially it would have a default weighting which would then be refined by actually getting feedback from the user. This would be in the form of a simple pop-up window that would display side-by-side the candidate object, and the object it wants to match it with. The user would then quickly press keys to say match or no-match (e.g 'y' or 'n'). This builds up a training set of positives and negatives which can be used to adjust the weighting for all the different methods.

Of-course there is no reason why with a good training set we should be limited to just a simple weighting. We can now use supervised learning techniques for a better detection rates such as decision trees or Bayesian networks.

Algorithms adapting to real-time Although I have a `TargetFPS` variable that can be set, and is propagated through-out the program, many algorithms do not use it. In the future, I would like to see all the algorithms to adapt to this parameter in a transparent manner to the end-user.

Support for Sound Some games rely on sound as a integral part of game-play mechanics - something GameScripter cannot handle right now. I would be interested to investigate how accurate one could locate an enemy in an First-person shooter, by listening to footsteps from the 5 sound channels all modern games provide. This is not a simple task as it requires a whole investigation into a whole new field of research - sound recognition.

Testing Framework Robust testing was something that was missing in my iterative design process. I sometimes broke previous GameProfiles due to changes to the framework interface, or added performance regressions. The testing framework would be very simple - it would consist of a new call-in `setTestVideo()`, that can set the "game" to instead be a pre-recorded video or image. Then a series of asserts could be used, to make sure the code is doing the correct thing. Below is an example of how I imagine a single testProfile would look like, testing both the accuracy and the performance of the `matchImage()` function:

```
setTestVideo("myVideo.mp4")

function callouts.newFrame(frame, deltaTime)
  if frame == 1 then
    x, y = matchImage("testImage.jpg")
    assert(x == 301 and y == 234)
    assert(deltaTime < 50)
  else if frame == 2 then
    x, y = matchImage("testImage.jpg")
    assert(x == 305 and y == 230)
    assert(deltaTime < 50)
  end
end
```

6.2.2 Hindsight

> "Before, you are wise; after, you are wise. In between you are otherwise"
>
> David Zindell (The Broken God)

With the benefit of hindsight, I would have done the following things differently:

Too open ended Although I am very happy with what I have achieved, I believe the project had too much "breadth". I feel that although GameScripter is capable of a lot of things, it doesn't do any of them to the best of my ability. I would have liked to narrow the scope of the project in order to concentrate on a few less features. For example, I would have liked to improve the performance of template-matching based on the ideas discussed in the report, but I lacked time. I also think the jump from 2D games to 3D games was a bit too large, and it would have nice to been to concentrate just on a single 3D game (e.g. an FPS) to get much better results.

Test-Driven Development As described in the previous section, I would have taken a more test-driven approach to minimise regressions, especially when testing multiple algorithms.

Memory-Management As described in section 5.2, I would have moved some of the memory management to the Lua environment, where it can be managed by the Lua garbage collector, and remove the need for handles.

6.3 Closing Remarks

I hope that through this report you can see that the novel approach of using a domain specific language has decreased the barriers of entry for users to use advanced computer vision techniques, in a completely new & unexplored field. The techniques applied during

development enabled rapid prototyping of algorithms, provided easy access to the large datasets that games provide and fast testing. These could then be applied back into real-life scenarios. Most importantly, through this project I have managed to excite and enthuse my peers at university with computer vision, by providing an accessible tool that encroaches into a field that many of them genuinely care about - games.

Appendix A

Additional Figures

A.1 Hu Invariant Moments

$$h_1 = \eta_{20} + \eta_{02}$$
$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$
$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$
$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$
$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$
$$h_6 = (\eta_{20} - \eta_{02})((\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$
$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$

Figure A.1: Hu Invariant Moments

Appendix B

GameScript Variables

These are set at the global level of the GameProfile - i.e. anywhere in the script

`rootX` - the x coordinate of the top-left corner of the parsing window

`rootY` - the y coordinate of the top-left corner of the parsing window

`width` - the width of the parsing window

`height` - the height of the parsing window

`targetFPS` - the desired speed at which the script will run

`maxFPS` - the maximum speed at which the script will run

Appendix C

GameScript call-outs

For all these call-outs, the arguments passed in are optional. Therefore you can only use them without no parameters, 1, 2 or 3 parameters.

`callOuts.newFrame(frameNumber, deltaTime)` is called every new frame. Note that this not every new frame of the game itself, but every time 1 iteration of the control loop has finished, and a new image has been received from the game. It passes in the `frameNumber` and `deltaTime` - the time in ms since the last call to `newframe()`

`callOuts.buttonRelease(x, y, button)` gets called, every-time a button is released. It passes in the coordinates where the button was released and which button it was.

`buttonPress(x, y, button)` gets called, every-time a button is pressed. It passes in the coordinates where the button was released and which button it was.

`callOuts.keyRelease(key)` gets called, every-time a key is released. A string of the key pressed is passed in - e.g. `'A'` or `'Up'`

`callOuts.keyPress(key)` gets called, every-time a key is pressed. A string of the key pressed is passed in - e.g. `'A'` or `'Up'`

`callouts.cursorMovement(startX, startY, endX, endY, motionTime)` gets called every time there is some cursor movement. It passes where the coordinates where the motion started from, where it ended, and how long it took in ms.

`callouts.closeGameProfile()` gets called just before GameProfile shuts-down.

Appendix D

GameScript Standard Object Table

For all call-ins which contain multiple object results, the same structure of table is used. The table returned is an array, indexed from 1, with a number of field

`centreX` - the centre X coordinate of the object

`centreY` - the centre Y coordinate of the object

`x` - the top left hand corner X coordinate of the object

`y` - the top left hand corner Y coordinate of the object

`width` - the width of the object, in pixels

`height` - the height of the object, in pixels.

`matchAccuracy` - context sensitive - if there was no matching involved, this would be -1, otherwise it is the quality of the type of matching technique used. E.g. if the result is from template matching, this will hold the quality of the template match. 0 signifies a bad match, while 1 is a perfect match.

`isTracked` - boolean if the object is being currently tracked or not.

Appendix E

GameScript call-ins

For all call-ins, the arguments passed in are optional. Therefore you can only use them without no parameters, 1, 2 or 3 parameters.

E.1 Input call-ins

`input.moveCursor(x,y)` - moves the cursor to coordinates `(x,y)` If `x` or `y` is `nil` or absent, the movement is constrained to only the 1 axis - e.g. if `x` is `nil` the cursor moves to the specified y coordinate.

`input.PressButton(button)` - presses `button` at the current cursor coordinate. If button is not specified, it uses the default button for your platform

`input.ReleaseButton(button)` - releases `button` at the current cursor coordinate. If button is not specified, it uses the default button for your platform (e.g. left mouse button for a PC)

`input.PressKey(keyString)` releases the key `keyString` (e.g. "a" would release key a, whereas "A" would release shift-A)

`input.ReleaseKey(keyString)` - releases the key `keyString`

`input.clickOn(x,y, button)` - moves the cursor to position `x,y` and then clicks there, using the `button` specified. If button is not specified, it uses the default button for your platform . If `x` or `y` is `nil` or absent, the movement is constrained to only the 1 axis - e.g. if `x` is `nil` the cursor moves to the specified y coordinate, before clicking.

`input.click(button)` - a shortcut for `input.clickOn(nil,nil, button)`

E.2 Vision call-ins

`input.matchImage(image)` - `image` can either be a string or a handle. This returns the x,y coordinates of the best match, or nil if no match is found. Uses template matching.

`input.matchMultipleImages(image) - \verb`image— can either be a string or a handle. Returns the GameScripter standard object table (Appendix D)

`vision.getHandle(object)` Returns a handle for the object passed in. This handle can then be used by any call-in that accepts an image.

`vision.getImage(x, y, width, height)` this gets a subset of the image, defined by the 4 arguments. It returns a `handle` to the image, which can be used in any GameScript function that accepts images.

`vision.getForgroundObjects(numberOfObjects, minArea)` Returns the GameScripter standard object table. If the background model learning has not started, this will start it too. Has 2 optional arguments: the first specifies the maximum number of objects to return (ordered by size). The second specifies the minimum area needed for the object to be considered. Uses background subtraction.

`vision.getForgroundObjectsOpticalFlow(numberOfObjects, minArea)` Returns the GameScripter standard object table. Uses optical flow to return foreground objects. Has 2 optional arguments: the first specifies the maximum number of objects to return (ordered by size). The second specifies the minimum area needed for the object to be considered. Uses background subtraction.

`vision.learnBackground(numberOfFrames)` - learns a background model over the next `numberOfFrames` number of frames. Uses code book method.

`vision.startBGSLearning()` Starts running learning the background model.

`vision.stopBGSLearning()` Stops learning the background model.

`vision.excludeRegion(x, y, width, height)` This excludes *all* vision algorithms from computation on the rectangular region defined by the arguments. This includes `matchImage(), getForeground`
and
`getTrackedObjects()`

`vision.matchColours(image, image)` Returns a value between 0 and 1 that indicates how well the colours in the two images match. 1 indicates a perfect match, while 0 indicates no match. Uses colour histogram matching.

`vision.matchShape(image, image)` Returns 2 values: The first is a value between 0 and 1 that indicates how well the shapes in the two images match. 1 indicates a perfect match, while 0 indicates no match. Uses Hu moment contour matching. The second is the rotation difference between the shapes. It is between 180 and -180 degrees.

`vision.findSquares(maxNumber, minSize)` returns list of objects representing squares. Returns *maxNumber* or less of squares (ordered by size), with a minimum size of *minSize*.

`vision.startTracking()` Starts tracking optical flow. Uses Lukas-Kanade optical flow

`vision.trackObject(object)` - Tracks the object passed in the argument.

`vision.getObjectPosition(object)` - Returns the GameScripter standard object table (Appendix D) if obj is a tracked object, otherwise nil

`vision.setGameBorder(x, y, width, height)` - GameScripter will only screen-scrape the part of the image of defined by the parameters.

Bibliography

[1] Bob Scott. *AI Game Programming Wisdom, Illusion of Intelligence.* Charles River Media, 2002.

[2] Alexander Nareyek. Ai in computer games. *Queue*, 1:58–65, February 2004.

[3] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, pages 1–9, New York, NY, USA, 2005. ACM.

[4] Roberto Ierusalimschy. *Programming in Lua, Second Edition.* Lua.Org, 2006.

[5] Cisco dynamic access policies. `http://www.cisco.com/en/US/docs/security/asa/asa80/asdm60/user/guide/vpn_dap.html`.

[6] Shivani Agarwal and Dan Roth. Learning a sparse representation for object detection. In Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen, editors, *Computer Vision ECCV 2002*, volume 2353 of *Lecture Notes in Computer Science*, pages 97–101. Springer Berlin / Heidelberg, 2006.

[7] Murat Kunt. Digital image processing. In Dionys Baeriswyl, Michel Droz, Andreas Malaspinas, and Piero Martinoli, editors, *Physics in Living Matter*, volume 284 of *Lecture Notes in Physics*, pages 73–75. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0009210.

[8] P Aschwanden and W Guggenbhl. *Experimental Results from a Comparative Study on Correlation-type Registration Algorithms.* Wichmann, 1992.

[9] A. Merigot, P. Clermont, J. Mehat, F. Devos, and B. Zavidovique. *A pyramidal system for image processing*, pages 109–124. Springer-Verlag, London, UK, 1986.

[10] W. Krattenthaler, K.J. Mayer, and M. Zeiller. Point correlation: a reduced-cost template matching technique. In *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, volume 1, pages 208 –212 vol.1, November 1994.

[11] T.K. Leung, M.C. Burl, and P. Perona. Finding faces in cluttered scenes using random labeled graph matching. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, June 1995.

[12] H.J. Wolfson and I. Rigoutsos. Geometric hashing: an overview. *Computational Science Engineering, IEEE*, 4(4):10 –21, 1997.

[13] S.X. Liao and M. Pawlak. On image analysis by moments. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 18(3):254 –266, March 1996.

[14] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111 – 122, 1981.

[15] Farhan Ullah and Shun'ichi Kaneko. Using orientation codes for rotation-invariant template matching. *Pattern Recognition*, 37(2):201 – 209, 2004.

[16] C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid methods in image processing, 1984.

[17] Dr. Gary Rost Bradski and Adrian Kaehler. *Learning opencv, 1st edition*. O'Reilly Media, Inc., first edition, 2008.

[18] M. Piccardi. Background subtraction techniques: a review. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 4, pages 3099 – 3104 vol.4, 2004.

[19] Kyungnam Kim, Thanarat H. Chalidabhongse, David Harwood, and Larry Davis. Background modeling and subtraction by codebook construction. In *In International Conference on Image Processing*, pages 3061–3064, 2004.

[20] A. Colombari, A. Fusiello, and V. Murino. Video objects segmentation by robust background modeling. In *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, 2007.

[21] Connected components labeling. `http://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm`.

[22] Satoshi Suzuki and KeiichiA be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32 – 46, 1985.

[23] H. Freeman and L.S. Davis. A corner-finding algorithm for chain-coded curves. *IEEE Transactions on Computers*, 26:297–303, 1977.

[24] C. H. Teh and R. T. Chin. On the detection of dominant points on digital curves. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11:859–872, August 1989.

[25] John Albert Horst and Isabel Beichl. A simple algorithm for efficient piecewise linear approximation of space curves. *Image Processing, International Conference on*, 2:744, 1997.

[26] B. Schiele and J.L. Crowley. Object recognition using multidimensional receptive field histograms, 1996.

[27] Y. Rubner, C. Tomasi, and L.J. Guibas. The earth mover's distance as a metric for image retrieval, 2000.

[28] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8:679–698, November 1986.

[29] M. Hu. Visual Pattern Recognition By Moment Invariants, 1962.

[30] Professor Guang-Zhong Yang. Imperial College 4th Year Computer Vision notes. `http://ubimon.doc.ic.ac.uk/gzy/m371.html`.

[31] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 2001.

[32] H.A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(1):23 –38, January 1998.

[33] H. Greenspan, S. Belongie, R. Goodman, P. Perona, S. Rakshit, and C.H. Anderson. Overcomplete steerable pyramid filters and rotation invariance. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, June 1994.

[34] E. Osuna, R. Freund, and F. Girosit. Training support vector machines: an application to face detection. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 130 –136, June 1997.

[35] Yoav Freund and Robert Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In Paul Vitnyi, editor, *Computational Learning Theory*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin / Heidelberg, 1995.

[36] Peter Bartlett, Yoav Freund, Wee Sun Lee, and Robert E. Schapire. Boosting the margin: a new explanation for the effectiveness of voting methods. 1998.

[37] Constantine P. Papageorgiou, Michael Oren, and Tomaso Poggio. A general framework for object detection. *Computer Vision, IEEE International Conference on*, 0:555, 1998.

[38] Dan Roth, Ming hsuan Yang, and Narendra Ahuja. A snow-based face detector, 2000.

[39] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *ARTIFICAL INTELLIGENCE*, 17:185–203, 1981.

[40] Jean yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. *Intel Corporation, Microprocessor Research Labs*, 2000.

[41] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[42] D. J. C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambrdige University Press, 2003. `http://www.inference.phy.cam.ac.uk/mackay/itprnn/book.html`.

[43] Sing Bing Kang. Semi-automatic methods for recovering radial distortion parameters from a single image. In *Research Labs, Technical Reports Series CRL 97/3*, 1997.

[44] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.