Pineapple Game Engine

Adam Yaxley
Third Year Computing Systems (MEng)
University Of Warwick
Supervisor: Dr Nathan Griffiths

26th April 2012

Abstract

This project explores a solution that removes the bottleneck that is present in scripting language based game engines. Using a compiled language for game logic instead of an interpreted language has major advantages, such as a huge performance increase, and removing the possibility of decompilation. Two case studies of current scripting language based game engines are analysed and a platform agnostic game engine is developed in C++ that features the same power and simplicity that scripting languages are popular for.

KEYWORDS*:* Game Engine, Computer Systems, Computer Graphics, Cross-Platform, Low-Level Routines.

Contents

Chapter 1

Introduction

This chapter provides a general introduction to the dissertation, and what the project is about.

1.1 Dissertation Outline

This dissertation will analyse the problems associated with scripting languages in games engines in depth and will propose a new solution that offers the same simplicity and power that is offered by current scripting languages but with a faster performance from the use of a compiled language. The design and development of the solution, as well as the project management techniques involved will also be detailed.

1.2 Acknowledgements

First and foremost, I offer my sincerest gratitude to my supervisor, Dr Nathan Griffiths, who has supported me throughout the entire project. His patience and knowledge has helped me to push my project in the right directions.

I would like thank all of the brilliant minds on stackoverflow[1] for their detailed discussions on popular computer science problems, which have given me a sense of standards, efficient algorithm design, and enlightened me with other popular techniques that have made the implementation side of this project possible.

I owe my deepest gratitude to all of the contributors to the unofficial Nintendo DS software development kit, to name a few: Michael Noland, Jason Rogers, and Dave Murphy. For their continued support and dedication to the tools that made it possible to create the Nintendo DS version of this project. I would also sincerely like to thank Jasper Vijn and Mukunda Johnson for their excellent tools for respectively compiling graphics and sound files into Nintendo DS

[1] An on-line question and answer community discussion.

ROM's[2]. I am also grateful for the on-line reference material and tutorials for the Nintendo DS provided by Jaeden Amero and Liran Nuna that made it possible to overcome the fear I had for Nintendo DS development. None of the actual testing for the Nintendo DS would have been possible without the work of Martin Korth and his Nintendo DS emulator[3], to whom I am greatly appreciative.

I am greatly indebted to Clinton Alexander for his daily support and helpful Linux knowledge. Also to my dedicated team of quality assurance testers for their patience and brilliant ideas: Jason Buwanabala, Daniel Law, Jenny Xiang, Aidan Yardy, and Josh Yaxley.

To all the contributors at freesound[4] for the sound effects and music that have been used in the demonstrations, and also to Emily Coad from DeviantArt[5] for her amazing pineapple graphic that has been used as the logo for the Pineapple Game Engine.

Lastly, I would like to thank my parents who have given me there full support throughout all of my university studies.

1.3 Motivation

The motivation for this project comes from my desire to create my own video games, and also from my experience with GameMaker[6] in the past eight years. GameMaker has gained huge popularity and a vast user base since its initial release in 1999. The main reason for this is that GameMaker boasts an incredibly simple user interface. The only disadvantage that this game engine has was the fact that it uses an interpreted scripting language for much of its game logic. This was a limiting factor and a significant bottleneck for developers and gamers alike. I was inspired by the simplicity of GameMaker's interface, but annoyed at the performance hit from the use of an interpreted scripting language. This motivated me to design a system that was extremely simple to use for the developer, and also extremely fast.

1.4 Aims and Goals

The purpose of this project is to develop a native, platform agnostic software system to be used for building graphical applications including games and simulations. Based on this, the two main objectives are defined as:

1. Remove the bottleneck that is present in game engines that use an interpreted language for game logic.

[2] A complete Nintendo DS executable application.
[3] A piece of software that acts as a virtual machine and emulates the hardware of another machine.
[4] Open database of free sound effects in the public domain.
[5] An Internet based art exchange forum.
[6] An easy game creation tool developed by Prof Dr Mark H. Overmars, Utrecht University.

2. Define a game as an abstract concept, creating a completely platform agnostic solution.

To achieve these two primary objectives the project will be broken down into two main parts. These two parts will together make up the Pineapple Game Engine, which will feature as a cross-platform practical solution to the problem. Specifically, these goals are as follows:

- Build the Pineapple Game Engine.
  - Create the Pineapple Object Interaction Framework, which will provide the necessary base components to interface with the application.
    - Provide a system to create and destroy objects in the current scene.
    - Have multiple scenes and the ability to switch between them.
  - Create the various plug-ins that will demonstrate that the Pineapple Object Interaction Framework is a functional solution to the problem.
    - Handle input and timers.
    - Provide support for physics.
    - Build a graphics engine.
    - Enable sound playback.

1.5 Legal, Social, Ethical and Professional Issues

One of the target platforms for this project is the Nintendo DS. There is some controversy in developing homebrew[7] software for the Nintendo DS, although there is no law against it.[20] Explicit permission is required from Nintendo in order for a company to receive official developer rights and access to the official software development kit, however that is not necessary for the scope of this project.

The Pineapple Game Engine uses FMOD for its sound system on desktop platforms, which is a proprietary audio library created by Firelight Technologies. Of the multiple licenses available, the non-commerical license has been adopted for this project, which allows FMOD to be used freely in software not intended for commercial distribution.[8]

[7]Software that has been developed using unofficial development kits.

Chapter 2

Related Work

This chapter surveys previous work that has defined a game as an abstract concept, incorporating the use of platform agnostic systems.

2.1 Simple Directmedia Layer

Simple Directmedia Layer (SDL) is a cross-platform open source game programming library that is available for a wide range of platforms including Android, Windows, Linux, Mac OS X and the PlayStation Portable.[27] It features generic low-level functionality for interacting with graphics, input and sound devices and is mainly used to develop cross-platform computer games. Due to its platform agnostic nature it can also be used to port games from one platform to another, and was used to port the popular old school shooting game Doom to BeOS. There is an unofficial Nintendo DS version of SDL, but it is not listed on their front page, and judging from the Nintendo DS section of SDL's souce code, it is still in early development and not very well supported.[19]

2.2 Allegro

Also an open source cross-platform library, Allegro attempts to generalise the platform by presenting a hardware abstraction layer to the developer for graphics, sound and input devices. It is structured in a similar way to SDL, with its functionality split up between separate modules and currently supports Windows, Mac OS X, MS-DOS, Linux, FreeBSD and BeOS.[12, p.37]

2.3 Discussion

SDL and Allegro both feature a modular based system, where each module has specific implementations available for each platform, and the correct implementation is chosen during

compilation. This design pattern is frequently used in many other cross-platform game engines because it reduces the platform specific work done by the developer, saving a lot of time.

Both support game specific concepts such as timers, graphics, sound, and input. Both feature 2D sprite manipulation, with functionality for scaling and rotation transformations. Support for tiled background maps is also included in both SDL and Allegro. These two features, sprites and maps, when used simultaneously are a fundamental component of the graphical architecture that most 2D games would require, which is why these graphical concepts will also be supported by the Pineapple Game Engine.

Even though both SDL and Allegro include functionality for such a wide range of game specific concepts, neither have support for physics. Computational physics is an extremely complex area of computer science and computer games development. The exclusion of a suitable physics component forces developers to incorporate physics into their games manually. There exists a selection of free physics engines such as Box2D[1] and the Bullet Physics Library[2] that could be integrated, but this is extra unnecessary effort for the developer. The Pineapple Game Engine will feature a minimalistic physics engine for use in games to solve this problem.

In SDL, events are gathered from both the user and the platform, which are then subsequently pushed into an event queue. This event queue can then be traversed by the developer to access the individual events. Allegro has a similar message loop system, but also offers support for a variable callback rountine, where the developer will specify a custom function that the events will be dispatched to, which is one step closer to automaticity.[12, p.175] Both libraries are written in C, and when used in C++ projects, with object-orientated design methods in consideration, this event loop and callback rountines are often wrapped in virtual functions in abstract event handling interfaces. This interface based system is one of the features that the Pineapple Object Interaction Framework will include natively.

Althrough both support a wide range of platforms, there is little or no support for the Nintendo DS system. Is it because the Nintendo DS has multiple screens? Or is it because of the complex limitations in its hardware? Whatever the reason, development for the Nintendo DS is never strong. To be truly platform independent the Pineapple Game Engine will have full support for the Nintendo DS, as well as two desktop platforms.

[1] A free opensource 2D physics engine written in C++ by Erin Catto.
[2] An opensource 3D physics engine that features soft and rigid body simulation and collision detection.

Chapter 3

Analysis

In this section the problem with current game engines will be analysed, and formal requirements for the project will be realised.

3.1 Scripting Languages in Game Engines

This section will discuss the current issues with scripting languages in game engines.

3.1.1 Introduction

Developing video games in recent decades has become an increasingly difficult and heavily involved process. Triple-A games[1] typically require a large team of skilled individuals, and can take over a year to complete. This is in contrast to the early 1980's where a few hobbyist programmers could group together and develop a triple-A game within a few short months. The reason for this dramatic change in industry is due to the many advances in technology over the past few decades. With an abundance of technology at the disposal of computer games studios nowadays, there are a whole host of considerations and technical challenges. For example, a typical modern game might have the functionality to calculate three-dimensional shadows in real-time, while integrating the velocities of the various physical bodies in the game to determine if any of them have collided. Providing a simple interface to complex functionality such as this is the job of the game engine.

Modern game engines typically encapsulate a large array of features suitable for developing triple-A titles, while increasing productivity and reducing error on behalf the developers that use them. This is often accomplished through the use of a scripting language, which acts as the layer in between the developer and the raw components of the game engine. The game engine acts as a virtual machine, dynamically interpreting scripts during runtime, allowing for detailed debugging messages to be produced upon a runtime error. Using a scripting language offers

[1] Of an extremely high commercial quality.

further abstraction from the game engine, and gives games developers more time to concentrate on the core game logic, rather than the complex integration of many of the high end features of game engines today.

However, there are some problems with this approach. The first problem is that game logic can only be processed as fast as the scripting language can be interpreted. For most mainstream desktop and console games today, this isn't really an issue. When the target platform is an handheld system on the other hand, where every clock cycle counts, the interpretation will serve as a major bottleneck. For example, Lua[2] suffers an on average 22x slowdown compared to a solution written in a compiled language.[9]

The second problem with this method is that a file-system is usually assumed to exist on the target platform. However, this is not always the case, as can be seen with the Nintendo DS. This will greatly limit what platforms the games that are built using these type of game engines will run on.

Since the game logic is being stored as a script either on the consumers hard drive or embedded in the executable, it is possible to decompile these scripts and inspect their source, which may reveal confidential or copyright assets. This access of internal resources is usually not favourable by the creators, who may need to keep their algorithms and data secure. Not only will hackers be able to potentially decompile these scripts, but they may also have the ability to reconfigure them, make modifications and recompile them back into the original game, hence altering the gameplay in one way or another[3].

3.1.2 List of Advantages

- *Debugging* - Object metadata can be inspected during runtime, and detailed reports can be provided upon a runtime error.
- *Abstraction* - Hides the complex components of the game engine, allowing the developer to concentrate on game logic.

3.1.3 List of Disadvantages

- *Speed* - Often significantly slower than their counterpart written in a compiled language.
- *Decompilation* - The source code can usually be inspected against the will of the creator.
- *Recompilation* - Modifications can be made to the decompiled source, and then recompiled back into the application.

[2] A lightweight scripting language that is often used in game engines.

[3] This process is known as "modding", and is popular in many games including the Unreal Tournament and Half Life series.

3.2 Case Studies

This section will analyse two popular game engines in the form of two separate case studies.

3.2.1 Case Study I: GameMaker

This case study will focus primarily on the pros and cons of GameMaker's inbuilt scripting language.

History

First released as Amino in 1999 by Prof Dr Mark H. Overmars of Utrecht University, it was primarily developed for creating short 2D animation sequences. An inbuilt scripting language was present in Amino, although it was at a primitive level compared to the functionality available in today's GameMaker. As its user base grew, Animo was renamed to Game Maker[4], introducing more functionality specifically for creating games.[13] It was so successful that it gained the attention of United Kingdom based company YoYo Games. Mark Overmars publicly announced their partnership in 2007.[14]

Features of the Scripting Language

Developers use GameMaker to first create rooms, in which instances of objects will reside and interact with each other. Rooms can be switched, and instances can be created and destroyed.[25] Each objects behaviour is defined as executing a series of actions, when certain events occur. Events can be anything from user input to collisions with other objects.[23] Actions are imperative statements that can change its own state and the state of other instances. These actions can be either be taken from GameMaker's large action library or programmed in by the user in GameMaker Language, GameMaker's own scripting language.[11, p.225]

GameMaker Language, commonly abbreviated to GML, is structurally similar to C in many respects. It is an imperative, object-orientated, event-driven language that supports a wide range of programming constructs, notably:

- *Functions* - A host of functions are provided as core, and it is possible for the user to define there own as a "script". Applications can also be extended by loading in additional functionality from dynamically linked libraries (DLLs) at runtime.[11, p.230]
- *Variables* - For simplicity, only two types are supported: string literals, and real numbers.[11, p.228]

[4]Today it is referred to as GameMaker, without a space.

- *Scope Change* - Variables can either be local to an instance, or local to a script. Instance namespaces can be changed from one instance to another to access variables in another instance's scope.[11, p.239]
- *Memory Management* - Memory is automatically allocated for variables in the current scope, and deallocated when they go out of scope.[25]
- *Resources* - There are no pointers in GML, instead each referable entity has a unique identifier. This is applicable to all instances of objects, sprites, sounds, fonts, backgrounds etc.[30]
- *Cross-Platform* - As of writing, GameMaker supports Windows PC, Macintosh and HTML5 browser based applications.[5]

Since GML can only be used for actions, the events, and object definitions are part of the integrated development environment itself and can not be scripted.

Advantages

GML is incredibly easy to use, and being able to switch between instance namespaces allows one instance to modify the state of or even destroy another instance.[11, p.239] It is especially important for this functionality to be present in a game engine, since it allows the interactions between the the individual entities to be modelled in consistent way. For example, if two physical bodies were to collide, the current state of both of these bodies would directly affect the resulting state of each of them. So the first physical body would have to inspect the state of itself and the state of the other physical body to determine what forces are exerted upon each of them, and what its resulting velocity and other such physical attributes should be set to.

The automatic memory management feature in GML frees the developers from touching complicated garbage collection algorithms and worrying about memory leaks, letting them focus on more important matters such as the quality of the gameplay. In addition to this, GML operates in a pointerless environment, eluding the developers from the large array of defects that are normally encountered whilst using pointers, such as segmentation faults and invalid memory accesses. If any bugs are present in the code, GameMaker will issue detailed runtime error messages, giving the user the option to ignore them.[11, p.60]

Another major advantage with GML is that extra components can easily be loaded in with external DLL libraries.[25] Third party extensions such as physics and graphics libraries can be linked into the executable in this way, providing developers with a large number of customisation options.

Disadvantages

Although it is evident that GML is a good scripting language, it does have its drawbacks. One of these is the limited data structures that are available to the developer. There are only two

types, string literals and real valued numerals, and there is no way to define new types.[25] Although it does feature some simple data structures such as stacks and queues, these are not very well integrated into the language, and thus cannot be used efficiently.[25]

When GameMaker builds executable files, the entire GML source is included as text within the executable itself. The interpreter is also embedded so that the GML source can be interpreted at runtime. The overhead of the interpreter significantly affects the execution speed of applications created with GameMaker when compared to the speed of a native application written in a compiled language.[9] It is also possible to decompile the executable file and extract the GML source code from it. With each release of GameMaker, the executable format is cracked and a new decompiler is released a short time later. Along with the GML source, all of the other resources that are associated with the application get exposed, such as graphics and sound data.[10]

3.2.2 Case Study II: Unreal Engine

This case study will focus on the notable features of Unreal Engine's scripting language, features of scripting languages that have not already been analysed in section 3.2.1.

History

Initially released in 1998 by Epic Games, the Unreal Engine featured a large set of integrated features and game specific concepts as an all in one package.[29] The core performance critical libraries were written in C++, while developers had access to the engine through the engines own scripting language, UnrealScript.

UnrealScript went through several iterations of design, and multiple programming paradigms were explored before a final style was decided. Tim Sweeney, founder of Epic Games and designer of the Unreal engine, wanted to create a language that incorporated the "major concepts such as time, state, properties and networking that traditional programming languages don't address".[29] Sweeney first looked into using the Java virtual machine, but decided against it since the overhead of both the scripting language and Java was too much for a playable frame rate.[29] The Java virtual machine had inefficiencies in its garbage collector in the case of a large object graph, and the overhead of the task switching within the virtual machine resulted in it being too slow to be viable. A second early implementation was based on a Visual Basic variant, but was discarded in favour of a C++/Java inspired syntax, which has the advantage of being immediately familiar to the developers.[29]

Since making its debut with Unreal, several new versions of the language have been released, each offering a wider variety of features.

Features of the Scripting Language

The language is primarily used for scripting gameplay events, where the set up of the world and scenes are controlled within the Unreal Editor. The noteworthy features of UnrealScript within the Unreal Engine are as follows:

- *Extendible* - In addition to the wealth of functions provided, developers can also use external DLL libraries for extra functionality.[29]
- *Memory Management* - An automatic garbage collection system frees memory from any unused variables.[29]
- *Game Specific Concepts* - Major concepts of time, state, graphics, physics, artificial intelligence and networking are part of the language definition itself.[29]
- *Editor* - A real-time editing and play testing tool.[31]
- *Cross-Platform* - As of writing Unreal Engine supports Windows PC, Microsoft Xbox 360, Sony Playstation 3, Apple iOS, Android and Sony Playstation Vita.[32]

Advantages

The syntax of UnrealScript is very similar to that of C++ and Java, which makes it immediately familiar to new developers. Borrowing concepts from C++ such as structs[5] and automatic garbage collection from Java, developers can quickly become comfortable with the language and start building applications.[29] Incorporating graphics and physics is done automatically, and creating a playable character in a simple test world can be accomplished within a considerably short period of time. A lot of the level design can be done in real-time within the powerful Unreal Editor, making for an easy sandbox to play-test design quickly and efficiently.[31]

The UnrealEngine offers a vast range of features for games developers, and provides cross-platform support for many of today's seventh generation home consoles.[32]

Disadvantages

There is a performance hit with processing UnrealScript: "Where design tradeoffs had to be made in UnrealScript, I sacrificed execution speed for development simplicity and power."[29] Tim Sweeny has openly said that UnrealScript suffers from a 20x slowdown compared to C++.[29]

The UnrealScript game source files are compiled into bytecode, and stored with the Unreal virtual machine as a package in the filesystem, allowing open access to the source code inside. Rogue programmers can decompile and modify these bytecode files to make alterations to the game. For instance, someone could modify the game logic so that the player would be invincible,

[5]Structure consisting of data members only.

thus changing the way the game is meant to be played. It cannot be assumed that all developers who use the Unreal Engine will be happy that people can modify their game without their consent.[16]

3.2.3 Conclusion

Both GameMaker and the Unreal Engine have similar advantages and disadvantages. The various aspects of both game engines will be separated into two distinct categories. One Category for those features that are desirable to have in a scripting language, and another category for those features which are undesirable.

List of Desirable Features

1. Read and write access to other objects state variables.
2. Automatic memory management.
3. Inbuilt game specific concepts.
4. Extendable.
5. Platform agnostic.
6. Has a familiar syntax to C, C++ or Java.
7. Simple and powerful.

List of Undesirable Features

1. Slow performance compared to a compiled language.
2. Possible to decompile and recompile.
3. Limited set of data structures.

3.3 Requirements

In this section the desirable and undesirable features will be considered, and a formal set of requirements will be established that will adhere to the aims and goals covered in section 1.4. Where a particular requirement implies a desired feature, it will be marked with a DFx, where x is the item number in the list of desirable features above. Undesirable features will be marked in a similar fashion but with a UDFx. The requirements for the plug-ins are taken from the discussion on related game engines covered in section 2.3.

1. Pineapple Object Interaction Framework requirements:
    (a) Completely platform agnostic (DF5)
    (b) Game objects
        i. Inherit behaviour and traits from each other
        ii. Inspect and alter the state of other objects (DF1)
        iii. Make use of functionality provided in external modules (DF4)
        iv. Receive events from external modules (DF4)
    (c) Scene handling
    (d) Automatic garbage collection for dead objects (DF2)
2. Plug-ins (external modules) (DF3) requirements:
    (a) Implementations of each available for each supported platform (DF5)
    (b) Input
        i. Provide a standard format for user input
    (c) Timers
        i. Alarms can be set to trigger events in the future
    (d) Physics
        i. Model the physical bodies as axis-aligned bounding boxes
        ii. Define collisions once, and check for them automatically during each frame
    (e) Graphics
        i. Sprites
            A. Have a physical position
            B. X and Y scaling
            C. Horizontal and vertical flipping
            D. Can have multiple frames
        ii. Tiled Backgrounds
            A. Have a physical position
            B. Tiles are placed on the screen according to a tile map.
            C. Can have two at any one time: a foreground and a background
    (f) Sound
        i. Play, stop and loop audio data.

These requirements almost completely include the list of desirable features, and do not include any from the list of undesirable features. The only desirable features that are not covered in the requirements are DF7 and DF6, which are to be considered during the implementation phase. It is also important to consider the list of undesirable features during the design and implementation stages to ensure that a successful solution is delivered.

Chapter 4

Design

This chapter will provide a detailed understanding of the design phase of the project, and the conception of a platform independent solution.

4.1 Core Design Goals

Several key goals were kept in consideration throughout the entire design phase. These goals are directly inspired from the list of desirable features of scripting languages in section 3.2.3, and are as follows:

- Power & Simplicity
- Automaticity
- Flexibility
- Extendability
- Portability

4.2 Engine Architecture

This section details the design of each of the components that make up the Pineapple Game Engine.

4.2.1 Game Objects

The state of a game is entirely dependent on the input events that are received from either the player or the platform.[26] These events have the potential to trigger further events, which

may cause even further events to occur, resulting in a domino like effect where these somewhat simple inputs can cause complex behavioural interactions. This idea that games are entirely event-driven state machines is the core principle of the Pineapple Object Interaction Framework.

In a computer game, everything is an object, from the player to the weapons and bullets, to the walls and aliens, everything can be represented by an object.[24] Game objects are physical entities that exist in some shape or form in the game. Multiple *instances* of game objects can be created, destroyed and process their own game logic. They are completely event-driven, and so their state can only be altered on the occurrence of some event. In order for game objects to receive events from external sources such as player input, they must register with the corresponding event handlers. Other than that, all game objects will start off with three basic events, which are as follows:

- *Create* - Fired when the object is first created.
- *Step* - This event is called once per frame to update the object with its own logic.
- *Destroy* - When the object is destroyed and removed from the game, this event is sent.

The requirements state that game objects must also be able to inherit behaviour and traits from one another. A parent child relationship model has been formulated to allow child objects to inherit behaviour and traits from their parent objects, specifically:

- *One way* - This relationship is strictly from parent to child, parent objects do not inherit behaviour and traits from their children.
- *Events* - Child objects are registered for the same events that their parents are, and behave in the exact same way by default. However, this default behaviour can be overridden with new behaviour if necessary. Note that this does not alter the behaviour of the parent objects, but specifically for that child object. This design is inspired from popular object-orientated programming paradigms and abstraction mechanisms.
- *States* - The parent's various state variables are passed on to the child.

The requirements also state that game objects may inspect and alter the state of other game objects. To do this, one must first locate a particular instance of an object, and in the Pineapple Game Engine there are two ways to do this:

- *Searching* - Game objects have the functionality to traverse through the list of instances of any other object in the game.
- *Events* - Game objects may be passed by a reference to the particular instance of the object through certain types of events, such as a collision event, where the reference of the colliding instance may be passed.

Once an instance has been found, there are various things that can be done with it. The instance can be destroyed, its state variables can be inspected and modified, behavioural procedures can be invoked, and events can be sent.

To handle events that can only take place when two game objects are in a certain state, a further component is needed, which has been named a *trigger*. These triggers routinely check and compare all of the instances for two sets of game objects to see if a certain condition is fulfilled, and if it is then the appropriate action is taken. An example usage of triggers are for collisions between two objects, where the condition would be if their bounding volumes are overlapping, and the action would be to send a collision event to each instance, and let the instances decide what to do.

Defining common game elements using this event-driven framework is simply done. For an example, consider the game space invaders. A core element of this game is the players ability to shoot and destroy the aliens to clear the level. To define that system of behaviours using this event-driven model, it would typically be done with three different interacting objects: the player's ship object, a bullet object and an alien object. Each object would have their own events and behavioural routines. An example setup is shown in the table below:

Object	Event	Parameter	Behaviour	Execution Order
Ship	Input	Fire Button	Create a Bullet	1
Bullet	Create	-	Move Upwards	2
	Collision	Alien Instance	Destroy Self	3
Alien	Collision	Bullet Instance	Destroy Self	3

4.2.2 World

The world acts as an overseer for all of the game objects in existence, and manages the various scenes in a game. Scenes can be menu screens, levels, or other such screens that are present in video games. Once instances of game objects are created in the world, they will follow the rules of the world, and interact with all of the other game objects in the current scene in a closed box environment. Each scene can be represented by a game object which acts as a *scene object* by creating initial game objects in the scene and setting them in the correct state. There are a few functions that can be performed on the world, these are:

- *Change Scene* - To change the scene a new scene object is declared. The world will then destroy all existing objects in the current scene, and create a single instance of the chosen scene object.
- *End* - Ends the world, and quits the game.

Every game object has access to the world, and can perform any of the functions listed above.

4.2.3 Plug-Ins

In the requirements it states that game objects can make use of functionality and receive events from external modules. These modules come in the form of platform independent packages, which can be used together or individually. Each package can provide a set of events and objects that are related to some game specific concept. The design of these plug-ins is heavily influenced by the two case studies analysed in section 3.2, and the works surveyed in chapter 2.

Input

This module monitors state changes in the generic input peripherals of the player and the platform, consequently dispatching them as events in a standard format. There are three main types of event, all containing the associated key identifier as a parameter:

- *Key Press* - When a key is initially pressed down.
- *Key Down* - Sent every frame that the key remains held down.
- *Key Release* - When a key is released.

Timers

This module provides functionality for managing multiple timers. Timers can be started and stopped, and are measured in frames. Only one event is required:

- *Timeout* - When a timer counts down and reaches zero. The associated timer identifier is sent as a parameter to distinguish between multiple timers.

Physics

This module is responsible for modelling objects as physical bodies, providing core state variables such as position, acceleration, and velocity, with additional functionality for collision checking. One object is provided:

- *Collision* - Routinely checks whether two game objects bounding volumes overlap. All game objects are modelled as axis-aligned bounding boxes. Makes use of a trigger.

To compliment this, an event is also defined:

- *Collision* - Sent by a collision object when two game objects are found to be overlapping. The other instance involved in the collision is sent as a parameter to each object, as well as a response vector for the collision.

Graphics

This module provides a suite of components for the cross-platform rendering of 2D graphics. The objects that are included are as follows:

- *Texture* - A resource that represents a single image.
- *Sprite* - Created from a texture object. Sprites are automatically updated and are used to draw moving images on the screen. Sprites have a position on the screen, a rotation angle, horizontal and vertical flipping and scaling, and a visible flag.
- *Animated Sprite* - This is the same as a sprite, except that it is created from multiple texture objects, and so contains multiple frames. Provides an extra attribute, which is the current frame of animation.
- *Tile Set* - A resource that is made up of multiple tile graphics.
- *Tile* - Contains a tile graphic identifier, and horizontal and vertical flipping flags.
- *Tile Map* - A two dimensional array of tile objects, which represent how the graphics of a tile set object are to be laid out.
- *Map* - Is the application of a tile map object to a tile set object. Maps are automatically updated, and are used to render the background on the screen. Maps have a position, and a horizontal and vertical scroll speed. Only two of these objects can be used at any one time, one for the foreground, and one for the background.

Sound

This module is used for playing sound effects and background music. For the scope of this project, advanced sound effects like echo and reverb are not necessary, and so only the basic sound functions will be included. A single object is provided:

- *Sound* - Once loaded, the audio data can be played, looped and stopped.

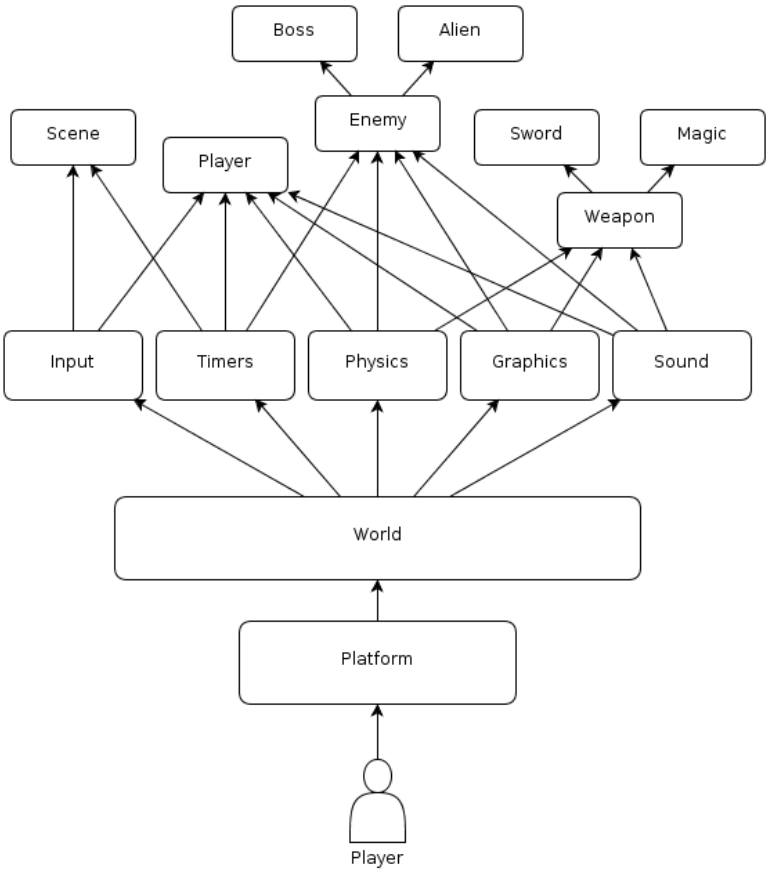

Figure 4.1: The architecture design of the Pineapple Game Engine.

4.3 System Structure

This section will detail the three stages of the system, from starting up to shutting down.

4.3.1 Start Phase

Before using the Pineapple Game Engine, it has to be set up in the environment. The stages to initialise the system are as follows:

1. *Create the Platform* - Firstly, the platform must be set up.
2. *Create Resources* - This is exclusively for the graphics and sound modules only, as texture, tile set and sound resources need to be first initialised to be used later in the game.
3. *Create the World* - The world is created and each of the modules are initialised.
4. *Create Game Objects* - These are the initial objects that will make up the starting scene for the game. Typically one *scene object* will be created which will set up the scene.

4.3.2 Processing Phase

Once everything has been set up, there is only a single thing that needs to be done. That is to run the main loop of the world object, which steps the world and all of the game objects one frame into the future. This fully automated system resembles one of the core design goals of the Pineapple Game Engine. What the world is actually doing in each execution of it's main loop in no particular order is as follows:

- *Step Game Objects* - Run each game objects step event to process the game logic.
- *Process Modules* - Each module may have its own task that needs to be completed each frame. For example, with the graphics module the sprites and maps need to be moved and rendered to the screen.
- *Process Triggers* - Execute the triggers set actions if any of the conditions are evaluated to be true.
- *Collect Garbage* - For any instances of game objects that have been destroyed in this frame, remove them from the world and free any resources that they were using.

Notice that great care has been taken in the design of this system to eliminate the need for any platform specific routines in the processing phase. The processing of the modules is not platform specific, as they are considered to be platform independent.

4.3.3 End Phase

When the game has finished, these are the stages that shut down the entire system:

1. *End the World* - Any game object has the ability to end the world at any point in the game, and when it does all of the game objects will be destroyed, and the modules will be shut down.
2. *Free Resources* - Any resources that were created at the beginning of the the game need to be released.
3. *Free the Platform* - Finally shut down the platform.

4.4 Platform Agnosticism

This section will detail the sections of the engine that are platform independent, and the sections which are not.

4.4.1 Platform Specific Phases

The whole engine is designed to extensively exclude any platform specific routines during the main processing phase. This will ensure that once the platform has been set up in the start phase, the Pineapple Game Engine will run in a fully deterministic manner. The world object runs the game using generic procedures, and the various integrated plug-ins are entirely platform independent, providing the game objects with a perfectly platform agnostic interface to the machine that the game is running on.

This leaves the start and end phases for the platform specific routines. During the start phase, different parameters are required to set up different types of platforms. A window size is needed to set up desktop platforms, but will not be necessary on handheld devices where the application fills the entire screen. Aside from setting up the platform, any platform specific resource objects that are used in the game are also initialised in the start phase. These resource objects need to be set up for the current *type* of platform. For example, a texture object might load in an image from the file-system on one platform, but read the data from the executable's data store on another. These subtle differences between each platform need to be addressed in the start phase before the game begins, and in the end phase once the game ends for a completely cross-platform solution to be realised.

4.4.2 Requirements

For a particular platform to be compatible with this system, there are some set requirements that need to met by that platform. The only components of this design that are interacting with the platform are the external plug-ins, which implies that the union of the set of requirements for each of those plug-ins is in fact the set of requirements in question for the entire Pineapple Game Engine. Listed, these are:

- *Input* - Some method retrieving input from the user.
- *Graphics* - At least one display device with support for two-dimensional graphics.
- *Sound* - Capable to output audio data to the user.

This implies that any platform that features the above functionality is a suitable candidate for the Pineapple Game Engine.

Chapter 5

Development

In this chapter the justification of the target platforms is explained and the methods used for the efficient implementation of the Pineapple Game Engine are presented.

5.1 Compiled Language Justification

The language that has been chosen to implement the game logic in the Pineapple Game Engine is C++, mainly for its execution speed and portability. C++ programs are compiled into machine code, alleviating the need for a virtual machine, and allowing applications to be developed for a wide range of different operating systems and processor architectures. This choice of language also satisfies DF6[1].

Most of the C++ compilers include a powerful compile-time optimisation feature, which can be set to different levels of optimisation upon compilation. Compilers will inspect the C++ source code, and over a series of iterations modify the code to increase its performance or decrease its memory consumption. The full set of optimisations is too large to discuss in this section, but the notable optimisation options are as follows:

- *Omitting the frame pointer* - On most machines functions will use up one of their registers for a frame pointer, even if they don't need one. Omitting the frame pointer relieves the need for instructions to save, restore and set up frame pointers. It will also make debugging impossible on some machines, so this is usually an option specifically for a finished application.[2]
- *Inlining functions* - Inlining a function causes the compiler to replace every instance of the function call with the function's body, removing the overhead of function calling. However, inlining large functions in several different places will increase the memory consumption of the application. The compiler uses a heuristic to determine which functions to inline to achieve optimal execution speed, whilst limiting memory consumption.[2]

[1] Has a familiar syntax to C, C++ or Java.

- *Devirtualisation* - A virtual function is an abstract function in an object, for which the implementation of the function may be different for each child object. This functionality can be overridden in several different layers of inheritance. The compiler usually stores a vtable for each instance of that object in memory, which points to each implementation for each virtual function, for that particular object. Each time a virtual function is called the vtable is used to look up the memory location of the correct implementation. Devirtualisation attempts to replace virtual functions with normal functions where appropriate to reduce these overheads.[2]

- *Reordering functions* - The compiler will reorder the functions in the object file to improve code locality.[2]

- *Reordering data structures* - The data members of a structure need to be aligned to particular offsets in memory. For example an four byte integer needs to be aligned to a four byte offset in memory. Padding is inserted in between some data members to ensure that each member is correctly aligned. The compiler chooses the optimum ordering of data members to reduce padding.[2]

- *Memory aligning* - Aligns the start of functions, labels, loops, and branches to a power of two boundary. This increases the speed at which the processor can jump to arbitrary instruction locations.[2]

- *Fast math* - Many optimisations can be made whilst performing mathematical functions. Some of these include turning off error checking for single instructions such as square root, and also assuming that the floating point values for both the arguments and the result of a function are valid and finite.[2]

C++ is proven technology, and the international standard for games development.[7] It is particularly powerful as an object-orientated language, featuring function and operator overloading, virtual functions and multiple inheritance. C++ boasts a fully supported set of modern concepts, including a powerful generics system. Since its conception in 1983, many unique code libraries have been built to solve many of the problems in computer science, making it an ideal language for this project.[28, p.7]

5.2 Target Platforms

This section will explain in detail the justification for the chosen platforms for this project.

5.2.1 Considerations

For each platform the different application development languages and software development kits must be compared. For a platform to be viable the software development kit must both be

available to unofficial developers, and the language used to build applications must be C++. A brief comparison of the different platforms available today is shown below.

Platform	Software Development Kit	Application Development Languages
Windows	Various	C/C++, Java...
Macintosh	Various	C/C++, Java...
Linux	Various	C/C++, Java...
Xbox 360	XNA	C#
Playstation 3	Authorised Developers Only	C/C++
iPhone	xCode	Objective C++
Android	Android SDK	Java, some C++
Nintendo 3DS	Authorised Developers Only	?
Nintendo Wii	DevkitPPC (unofficial)	C/C++, assembly
Playstation Vita	Authorised Developers Only	?
Nintendo DS	DevkitARM (unofficial)	C/C++, assembly
Playstation Portable	DevkitPSP (unofficial)	C/C++, assembly

Two desktop platforms will be chosen to show that this project is a platform agnostic among desktop environments, i.e. does not rely on components from a particular operating system. To demonstrate that the Pineapple Game Engine is platform agnostic across all platforms, one more platform will be chosen that is completely unique to the rest of them in terms of its user interface and hardware components.

5.2.2 Microsoft Windows

Today, the single most popular platform for game development is Microsoft Windows, being the main gaming system for 49% of gamers aged 16-49.[15] Windows development is well supported with hundreds of pages of references and tutorials available on the Microsoft Developers Network. Microsoft Visual C++ Express Edition will be used to develop the solution for the Windows platform, since it features a large array debugging features specifically for Windows that will speed up the implementation and testing phases.

5.2.3 Linux

Linux based systems are on the uprise, and since the operating system is built in a completely different way to that of Microsoft Windows, it will show that this project is cross-platform in terms of desktop environments. For Linux, the GNU[2] compiler collection will be used for the compilation process. There are some extremely powerful Linux specific development tools that are to be used for debugging and profiling, which have been used in the validation stages of the Pineapple Game Engine.

[2]A massive open source suite of software developed by thousands of collaborators.

5.2.4 Nintendo DS

The third and final platform for this project is the Nintendo DS. Most people believe that the DS stands for "Dual Screen", when in fact it initially stood for "Developer's System".[22] The Nintendo DS was the first handheld console to incorporate so many features into one device, most notably: two screens, one with touch sensitivity, and a microphone. Nintendo believed that "it gives game creators brand new tools which will lead to more innovative games for the world's players."[22] This is certainly true, considering the huge success of the system and variety of games that are developed for it.

The official development kit (NitroSDK) is only available to authorised Nintendo developers, but there does exist an unofficial development kit in the DevkitPro[3] tool-chain, DevkitARM[4]. Included in the DevkitARM package is libnds, which is a library for the development of Nintendo DS applications. There is only a handful of reference material and tutorials publicly available for Nintendo DS development, which means that the Nintendo DS version of the Pineapple Game Engine will be a significant challenge. Refer to section 7.2 on page 47 for a detailed review of the technical challenges involved in Nintendo DS development.

5.3 Implementation

This section will detail the implementation of the different components that make up the Pineapple Game Engine, focusing on the Pineapple Object Interaction Framework. For reference purposes, a complimentary UML diagram of the Pineapple Game Engine is attached in appendix A.

5.3.1 Pineapple Object Interaction Framework

This section will give a detailed description of the core components that make up the Pineapple Object Interaction Framework.

5.3.1.1 Objects and Instances

Instance Base

The first component of the Pineapple Object Interaction Framework is the *InstanceBase*. The *InstanceBase* class represents the base traits, behaviour and events for every game object, where traits are represented by variables, behaviour by functions, and events by virtual functions. *InstanceBase* is the top most superclass for game objects, allowing every game object to be upcasted[5] to an *InstanceBase* object. This functionality is useful for executing common procedures

[3] A series of unofficial software development kits for various video games consoles.
[4] The specific tool-chain for ARM based processors.
[5] Converting a derived class pointer to a base class one.

for every game object, such as traversal or destruction. This is similar to Java, where every class is an "Object".[1]

InstanceBase objects have the three basic events: *onCreate*, *onStep* and *onDestroy*, which were covered in the design chapter. Each event is implemented as a virtual function with an empty function body by default. This means that every game object isn't required to provide behaviour for these events, but rather leaving it as a choice for the developer.

InstanceBase objects have two functions that destroy them, one named *destroy* which also sends an *onDestroy* event, and one named *kill* which does not. The latter is provided for cases where instances need to be quickly cleaned out of memory without invoking any behavioural rountines specific to the game logic, such as during a change in scene. Even though the *onDestroy* event is not guaranteed to be called, the *destructor* of the class is, which is used to release any resources that were being used by the object. This destructor is a in fact a *virtual destructor*, which means that when a pointer to base is deleted[6], the destructor in dervied is actually invoked. This ensures that any memory that the derived object was using in addition to the base object is also released, and is a behaviour recommended for partially abstract classes by Bjarne Stroustrup himself.[28, p.324]

When *destroy* or *kill* is invoked the instance is not deleted straight away, instead the instance is flagged as "dead", and is cleaned up with any other "dead" instances during garbage collection. This is done to avoid potential invalid memory accesses during game logic. For example consider the situation where object A and object B collide, and both recieve an *onCollision*[7] event with a pointer to the other involved in the collision. Now if object A's onCollision event is called first and object A is destroyed as a result and the memory is freed, when object B's onCollision event is called the pointer to the instance of object A it recieved is now invalidated, and any access to this pointer will cause a segmentation fault. Deleting all of the "dead" instances at the same time, outside of game logic, avoids this potential error and provides a safe sandboxed environment for games to run in.

Instance List

The next most important component to discuss is the *InstanceList*, which can store an arbitray number of game objects. The *InstanceList* class provides methods for storing and removing game objects, as well as traversing though them. It acts as a templated container, which can store any type of game object, and can also be upcasted to an *InstanceListBase* object which treats the game objects contained within as type *InstanceBase*. The InstanceListBase version is used throughout the Pineapple Object Interaction Framework as a generic container for game objects.

But what data structure should *InstanceList* use to store game objects? The time complexities of the basic operations of two suitable candidates are compared in the table below:

[6]The memory it was using is freed by the platform.

[7]Inherited from a the physics plug-in, explained in detail later on.

Data Structure	Insert	Remove	Traverse
Vector	$O(n)$	$O(n)$	$O(n)$
Doubly Linked List	$O(1)$	$O(1)$	$O(n)$

The vector data structure has the advantage of having its elements strored in a contiguous memory block, which means that when its elements are accessed, several consecutive elements are also loaded into cache lines at the same time, providing a higher locality of reference. This will greatly reduce the number of cache misses during traversal, compared to that of a linked list, where all of the elements are stored at arbitrary locations in memory. However, it is extremely important to have fast insertion and removal operations, especially during the execution of complex scenes where game objects are being created and destroyed multiple times per second. The fast implementation of these operations for a linked list outweighs the disadvantage of cache misses during traversal, and that is why a doubly linked list was chosen for the internal representation of the *InstanceList* class.

To adhere to good software engineering principles, the insertion, removal and traversal operations on an *InstanceList* object are generalised through the use of iterators. This allows the underlying representation of *InstanceList* to be changed without breaking other parts of the system. The iterator class for *InstanceList* is called *InstanceIterator*, and *InstanceIteratorBase* is used for iterating over *InstanceBase* objects in an *InstanceListBase*.

Instance Manager

The *InstanceManager* class stores pointers to each game object's own *InstanceList*. These *InstanceList*'s are stored in an ordered linked list, where each *InstanceList* is ordered according to it's priority. The priority of the *InstanceList* can be changed during runtime, automatically reordering the linked list. The main use of this class is to provide a complete snapshot of all of the instances that exist in the game at that current moment. This is known as the *master instance table*, which can be used to traverse through every existing instance in the game.

The *InstanceManager* class is also used for processing each instances *onStep* event, garbage collection of instances, and deleting any remaining instances at shutdown.

Instance Static

The *Instance* class is a templated class which is completely made up of static data members and functions. There is no way to create an object of *Instance*, nor is there any need to. It takes a game object class as its template parameter and acts as a static store of information and functionality related to that particular game object. It is used for the following:

- *Creating* - All instances of game objects are created using the *Instance* class.
- *Storing* - The *Instance* class is where the *InstanceList* for the game object is stored.

- *Registering* - Ensuring the instance lists appear in the *InstanceManger*.

The *Instance* class operates in a very special way, exploiting implicit template instantiation to automate the whole set up for each game object without the developer even lifting a finger. This is discussed in detail in section 7.1.1 in the Technical Challenges chapter.

5.3.1.2 The World

Task Manager

The *TaskManager* is a static class that is responsible for managing the tasks that need to be carried out during the start, process and end phases. Each task is represented as a function with a priority, and so each list of tasks in the *TaskManager* is stored as an ordered linked list of *function pointers*, ordered by their priority. Functions can be added as tasks to any of the three lists if firstly they do not take any parameters, and secondly do not have a return value.

There are three objects defined that make the process of adding tasks automatic. Each object has only one possible constructor, which takes two arguments: a pointer to a function that matches the requirements of a task, and a priority as an integer. These objects have been named *AutoInitTask*, *AutoProcessTask* and *AutoShutdownTask*, where each object adds a task to a different list upon construction. For each class that requires a particular task to be executed, one of these three objects is included in the class declaration as a static data member with a function pointer and a chosen priority as parameters. Once the static member is initialised, the task will be added to the *TaskManager*. In C++, the initialisation of static data members is carried out before the program has even hit the main function, which means that the *TaskManager* is ready to be used as soon as the main procedure is executed.[2]

Triggers

The functionality of triggers is given by the *Trigger* and *TriggerSelf* classes, both of which are derived from the *TriggerBase* class. The *Trigger* object is constructed by providing two different *InstanceList* objects, and the *TriggerSelf* object is constructed by providing only one *InstanceList* object. These triggers are actually interfaces, containing a single virtual function that must be implemented, which takes two instances as arguments. When the *Trigger* object is processed, the function is applied to each possible two instance combination, one from each *InstanceList* object. The *TriggerSelf* object is processed in a similar way, but the function is applied to every possible combination of two instances from a single *InstanceList* object. The use case scenarios of triggers are explained in section 4.2.1.

Platform

The *Platform* interface class represents a generic platform, the implementation of which is different for each platform. Each implementation of the *Platform* interface may have different

parameters for its construction as discussed in section 4.4.1. However once constructed, the current platform can be accessed by other objects in a standard way, without the worry of writing platform specific code. The interface defines the following virtual functions:

- *pollEvents* - The platform checks and stores all events that have occurred since this function was last called.
- *getEvents* - The player and platform events are retrieved as a linked list of *Event* objects.
- *queryDevice* - Access to various devices such as the mouse or touch screen.
- *setFullScreen* - Switch between windowed mode and full screen mode on desktop environments.
- *swapBuffers* - Some systems that use double buffers require that the back buffer be swapped with the front buffer once per frame.
- *waitForNextFrame* - A function that simply waits until the current frame has finished, delivering game play at a maximum of 60 frames per second.

This functionality is implemented by virtual functions, and any functionality that is unavailable is simply left with an empty function body. This is done so as to not break games that use different features on different devices.

But how does the system determine which platform it is currently running on? During compilation, a preprocessor flag that represents the current platform is defined in the compilers options. This is called a *preprocessor define*, and can be used to determine which sections of the source code to include in the current compilation unit. Using this functionality, the correct platform program code and libraries are always included automatically.

World

The *World* object is at the top level of the application, and is responsible for processing each and every frame. Only one *World* object is created, and upon construction it takes a pointer to a *Platform* object, which is stored for later use. When the world is created the tasks that have accumulated in the initialisation task list in the *TaskManager* are executed, and a static *World* pointer in the *InstanceBase* class is set to the value of this object, allowing every game object access to the functions provided by *World*.

Once created, to step the game one frame into the future, the *mainLoop* function is called. The *mainLoop* returns the value *true* if the *World* has not yet ended, and *false* otherwise. This allows the user to process the entire world, from start to finish with a single line of code:

```
while ( world.mainLoop() );
```

To end the *World* and finish the game the *end* function can be invoked. When the *World* object is deconstructed, the tasks in the shutdown task list in the *TaskManager* are executed.

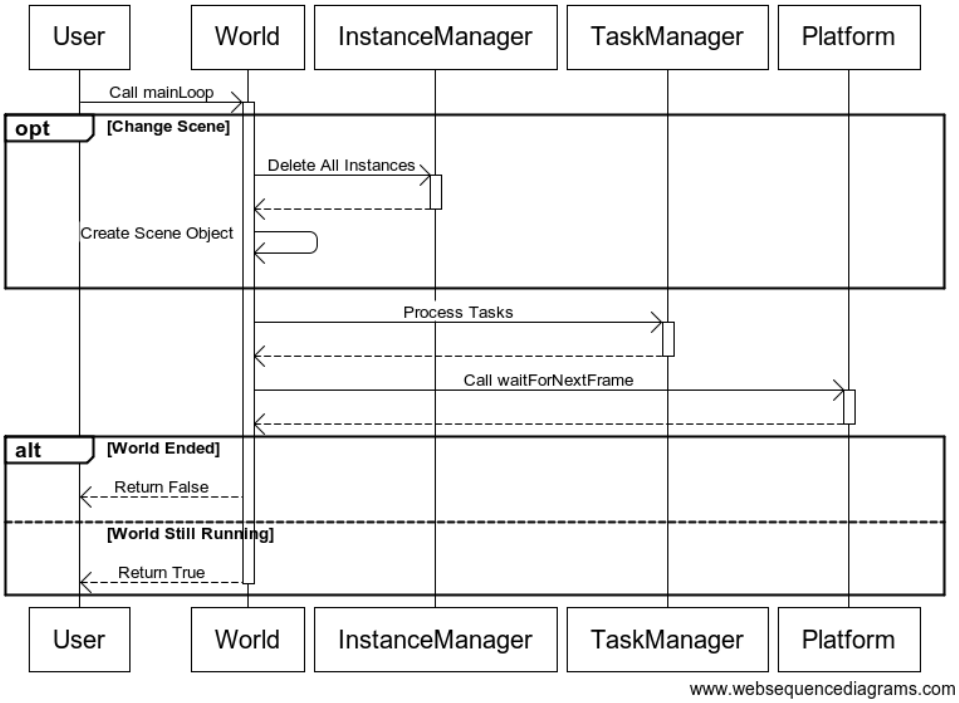

Figure 5.1: The main loop of the World

The program below shows the stages involved in setting up the world and the platform, and creating the first object. However, once this "setup" code has been written, it never needs to be changed, since the initially created object can be used to drive the game from then on.

```
#include <pineapple/pineapple.h>

struct Scene : public InstanceBase { /* Empty object */ };

Platform *initPlatform()
{
    // Choose the platform to create based on preprocessor defines
    #ifdef        PA_LINUX
        return new LinuxPlatform(640, 480, "Pineapple");
    #elif defined PA_WINDOWS
        return new WindowsPlatform(640, 480, "Pineapple");
    #elif defined PA_NDS
        return new NdsPlatform();
    #endif
}

int main()
{
    Platform *platform = initPlatform();          // Initialise the platform
    World world(platform);                        // Create the world
    Instance<Scene>::create();                    // Create scene object
    delete platform;                              // Shutdown the platform
    return 0;                                     // Exit
}
```

Figure 5.2: The general setup code for any game.

```
struct Scene : public InstanceBase
{
    void onCreate() {
        PA_PRINTF("Hello World\n");
    }
};
```

Figure 5.3: Hello World

5.3.1.3 External Module Support

The final component of the Pineapple Object Interaction Framework is used to provide an interface to additional functionality for game objects through the use of external modules such as input timers, physics, graphics and sound. These plug-ins must be able to provide additional events and procedures to the ones provided in *InstanceBase*. Two separate designs were considered for the implementation of this component, one using inheritance and the other using composition. Once an external module has been implemented with this component, the developer should be able to utilise the functionality inside in a simple and powerful way. Minimising the work done by the developer, while providing a high execution speed is a key goal in the implementation.

The general structure of each design's usage can be seen in appendix B. Using the composition

structure, external modules are attached to the game object by composition, and the functionality for the modules can be called through the modules corresponding object. Events are *hooked* in to the game object by providing a *callback* function pointer for each event. However, in the inheritance structure, events from modules are *inherited* as *virtual functions* into the game object. Other functions and state variables are also inherited in the same way. This inheritance structure uses less logical lines of code than the composition one to achieve the same functionality, making it the ideal choice.

Handler

The *Handler* class is the implemented interface for the inheritance design discussed. It is an extension of the *InstanceBase* class, and can automatically process event handling and behaviour for external modules, satisfying DF4 of scripting languages. Each module includes a *process* routine that defines the necessary steps to process and handle events for a single instance that uses it. For example in the process function for a physics module, the forces on the physical body will be resolved and the position of the physical body will be updated. Each *Handler* class is templated with the module class that derives from it, giving each module a copy of the functionality, but which is specific to that module.

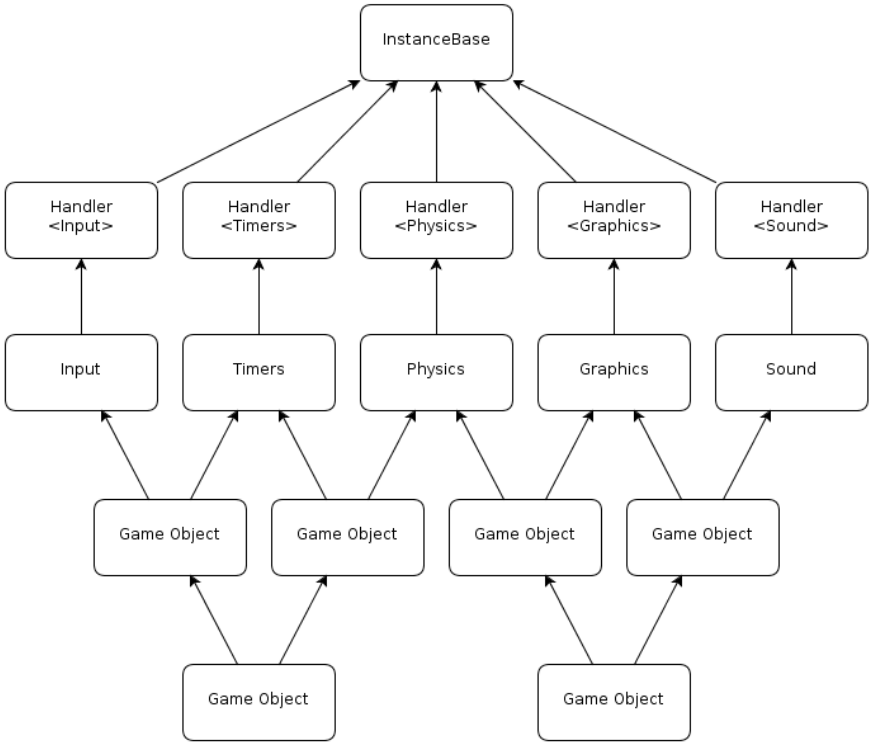

Figure 5.4: The class hierarchy of the game objects and external modules in the Pineapple Object Interaction Framework.

To process the instances the templated *Handler* class traverses all of the exisiting instances in the master instance table from the *InstanceManager*, and runs each *process* rountine for each instance for that templated module. This is done automatically using the *TaskManager* for each module on every instance, thus correctly updating all game objects.

The master instance table consists of instances of game objects in their generic *InstanceBase* form. Whether an instance is using a particular module or not can be easily tested through the use of a dynamic cast[8]. The handler also features a set of *static virtual functions*[9] that can be used for pre and post processing tasks that are required by the module.

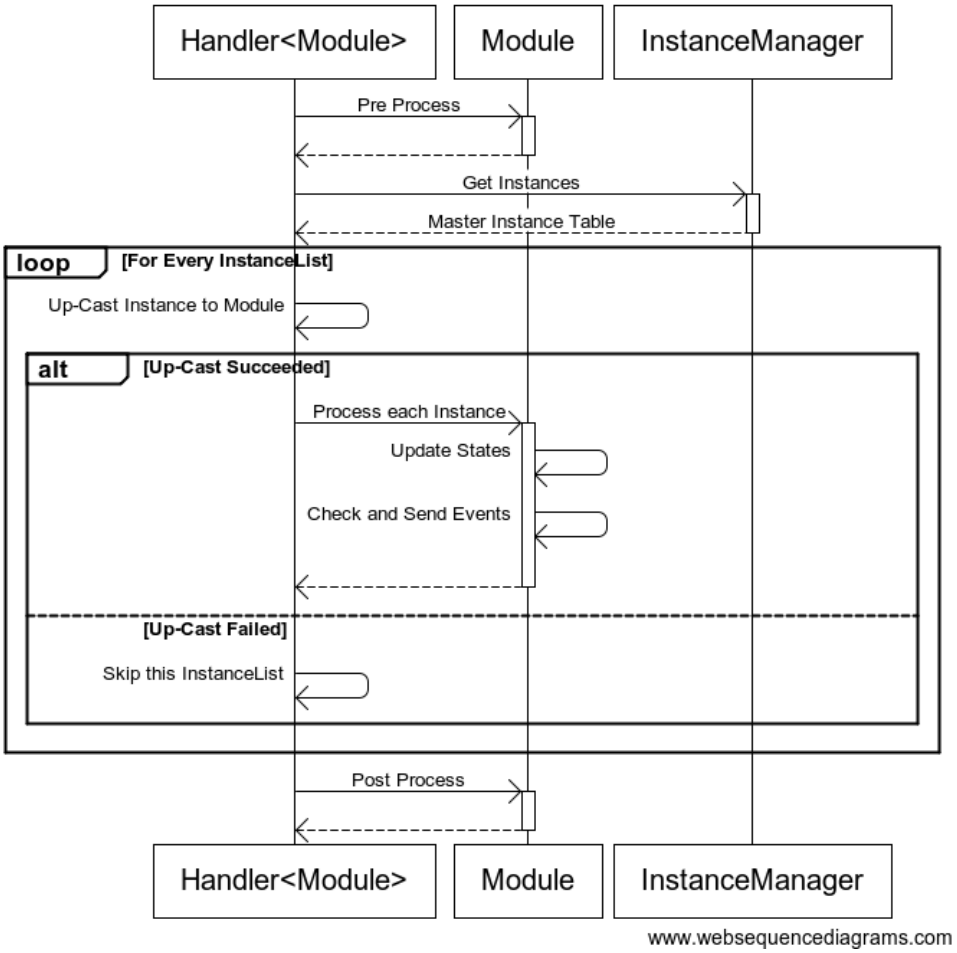

Figure 5.5: A Handler's process routine for a particular Module

Justification for Diamond Inheritance

The so called "dreaded diamond of death" is a problem with class hierarchy where an object is derived from several other objects which all derive from the same base object.[21] This can

[8] A down-cast from a base class to a derived class, which fails if the derived class does not derive from the base class.

[9] These are not part of the C++ language definition, but this paradigm can be simulated through the use of templates.

be seen in figure 5.4 with game objects at the derived end, and *InstanceBase* at the top of the diamond. It is generally avoided because the derived object will have multiple copies of the base object in its definition, one for each possible inheritance path, resulting in ambiguous behaviour when a function in the base class is invoked, since the function will appear mutiple times in the class definition. In C++ this problem is easily solved through the use of *virtual inheritance*, which ensures that only one copy of the base class is present in the classes at the bottom of the diamond.

However, this model imposes the restriction that external modules must not override any functions or implement any virtual methods in *InstanceBase*. In practice this is unnecessary anyway, since the *Handler* class provides more than enough functionality for manipulating instances.

5.3.2 Plug-Ins

In this section the implementation specific details for the various plug-ins are briefly summarised.

Input

Most of the work for this module is already handled by the *Platform* object. All the input module does is dispatch these input events from the platform to the game objects in the form of *onKeyPress*, *onKeyDown* and *onKeyRelease* events with the key in question as a parameter.

Timers

A fixed number of timers are available in this module, each having a unique identifier. Each timer can be started, and stopped, and when a timer counts down to zero, the *onTimeout* event is sent with the timer identifier as a parameter.

```
struct Scene : public InputHandler, public TimersHandler
{
    void onKeyPress(unsigned int key) {
        if (key == PA_ESC)
            startTimer(0, 60);
    }

    void onTimeout(unsigned int id) {
        world->end();
    }
};
```

Figure 5.6: Combining functionality from two modules

Physics

Each physical body has a vector data structure for its *position*, *velocity* and *acceleration*. These vectors use floating point calculations on desktop environments, and fixed point operations on the Nintendo DS system, since the Nintendo DS has no floating point processing unit. The *Collision* object is implemented using the *Trigger* objects provided in the core framework. Two objects collide if their axis-aligned bounding boxes (AABBs) overlap, which sends an *onCollision* event if they do overlap with a pointer to the other instance and the corresponding response vector as parameters. AABB collision detection was chosen for its simplistic implementation, the psuedo code for which is attached in appendix C.

Graphics

The graphics module consists of a set of abstract classes and two different implementations, one in OpenGL for desktop computers, which was chosen for its cross-platform nature, and one which is specifically written for the Nintendo DS. There are two main subsystems in the graphics module, the *sprite engine*, and the *tile engine*. The sprite engine is responsible for rendering animated images to the screen, while the tile engine draws a series tiles on the screen in the form of a foreground and a background.

Sprite objects are created from *Texture* derived resource objects, and *Map* objects are created from *TileSet* derived resource objects. The same way the correct Platform object is chosen using preprocessor defines, these resource objects are created in a platform specific way during the start phase, after the *Platform* object has been initialised. On Windows and Linux *FileTexture* objects are created by specifying a file name, while on the Nintendo DS, *NdsTexture* objects are created from graphics memory addresses. Both of these objects implement the *Texture* interface, and both have a *createSprite* method which returns a pointer to an abstract *Sprite* object, which also has a different underlying implementation for each platform. *Map* objects are created in a similar way except a concrete *TileMap* object is passed as a parameter to the *createMap*.

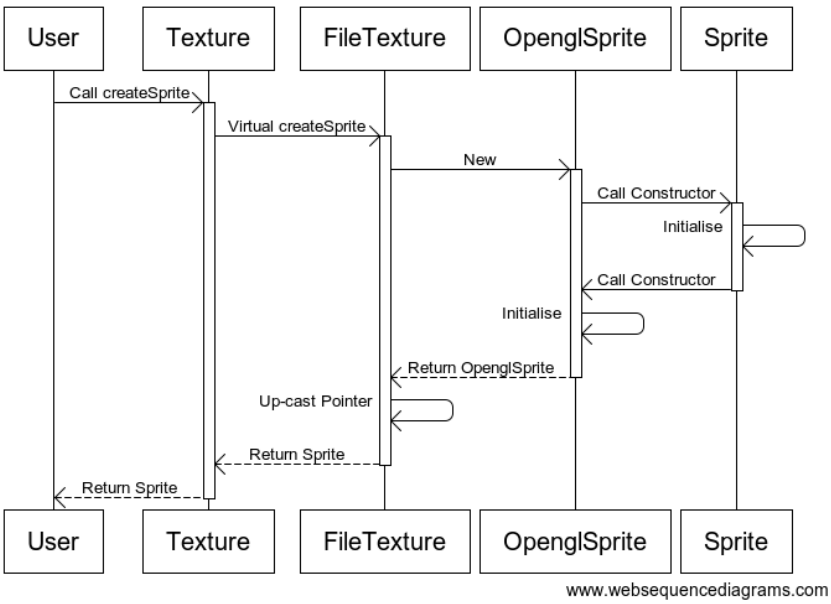

Figure 5.7: Abstract sprite creation on Windows and Linux platforms

Both *createSprite* and *createMap* from the *Texture* and *TileSet* interfaces respectively feature an optional parameter *screen*, which specifies which physical screen to create that object on. This parameter is only used on devices with multiple screens, such as the Nintendo DS.

The Nintendo DS implementation of the graphics engine is discussed in great depth in section 7.2.

Sound

Sound objects are created in a similar way to graphics objects, except there is no *createSound* function. *Sound* is simply an abstract class that is either implemented using FMOD[10] on Windows and Linux, or maxmod[11] on the Nintendo DS.

5.4 Putting it all Together

The following shows how a game object is defined by utilising functionality from different modules, and making use of resource objects for sprites and sounds.

[10] A proprietary audio library created by Firelight Technologies.
[11] An audio system built for the Gameboy Advance and Nintendo DS by Mukunda Johnson.

```
struct Player : public TimersHandler, public InputHandler, public Physics2DHandler
{
    bool canShoot;
    Sprite *sprite;

    Player() : canShoot(true){
        sprite = resource::playerTex->createSprite();
    }

    ~Player() { delete sprite; }

    void onCreate() {
        sprite->rotation = 270;
        size.cart(28, 28);
    }

    void onKeyDown(unsigned int id) {
        Vector increment;

        switch (id) {
        case PA_UP:
            increment.polar(0.1f, PA_DEGTORAD(sprite->rotation));
            velocity += increment;
            break;
        case PA_LEFT:
            sprite->rotation -= 4;
            break;
        case PA_RIGHT:
            sprite->rotation += 4;
            break;
        case PA_SPACE:
            if (canShoot) {
                resource::fireSound->play();
                Bullet* bullet = Instance<Bullet>::create();
                bullet->position = position;
                bullet->velocity.polar(7.0f, PA_DEGTORAD(sprite->rotation));
                canShoot = false;
                startTimer(0, 10);
            }
        }
    }

    void onMove() {
        sprite->position = Vect2<int>((int)position.getX(), (int)position.getY());
    }

    void onTimeout(unsigned int) {
        canShoot = true;
    }

    void onCollision(Rock*, Vector&) {
        destroy();
    }

    void onDestroy() {
        world->end();
    }
};
```

Figure 5.8: The Player object in the game Asteroids

Chapter 6

Project Management

This section will detail the various aspects of project management that were utilised throughout this project.

6.1 Software Development

This section will discuss the software engineering concepts that were used throughout this project's software life cycle.

6.1.1 Software Development Model

The software development model chosen was the V-model, where the solution is tested against the requirements after each coding phase. This model was chosen primarily for its extensive verification and validation phases, and with a project such as this one, constantly verifying that the software works, and validating that the software is in line with the requirements is key to developing a functional solution.

6.1.2 Software Engineering Principles

To take complete advantage of C++, an object-orientated programming paradigm was adopted, where the solution consisted as a series of separate objects. The Pineapple Object Interaction Framework is designed and implemented as a completely extendible piece of software, providing non-intrusive ways for new functionality and platforms to be plugged in. To ensure compatibility and enforce strict requirements for external plug-ins, interfaces and partially abstract classes are used extensively. Polymorphism concepts are used throughout to generalise functionality and to process objects in a uniform way.

6.1.3 Source Code Management

The codebase for this project is quite large, and so major revisions need to be recorded in a standard format. There are a number of source code versioning tools that are freely available that would do the job. From these, git was chosen for its incredibly simple interface between repositories. Since the project source tree would be present in several different locations at any one time, git[1] was more than capable.

There was a repository set up on an offsite privately owned server, that was to be used as a primary back up store, and serve as a intermediate link for commiting changes between the repositories on the separate platforms. As well as the server repository, a repository exists on a Windows system, a Linux system and a USB stick.

Whenever a revision is made the files in the repository are committed and pushed to the repository on the private server. With every major update to the codebase, the files from the repository are copied to a USB stick for extra security. Comments are recorded with each update so that the history of the project can be tracked to make sure that the project is on schedule.

6.2 Verification

This section details how the software was verified to fit with the requirements of the end user.

6.2.1 Requirements Analysis

In chapter 3 the requirements were directly derived from the features of two popular game engines. The design of each was carefully broken down to determine which parts worked well, and which could be improved upon, and from this and additional analysis in chapter 2, the requirements for a solution were defined that also matched the project's aims and goals in section 1.4.

6.2.2 Verification of Design

The system was designed in such a way that completely abstracted out the concept of a video game, and broke it down into a fully event driven system consisting of *game objects* and the *world*. The design patterns that were used incorporated limitless bounds on extendibility, allowing the use of external plug-ins to add a variety of *game specific concepts* to the existing system, and the ability to port to *any platform* that matched the platform requirements determined in section 4.4.2.

[1] A distributed revision control and source code management system.

6.3 Validation

This section details how the software was validated to be working correctly, and free from defects and failures.

6.3.1 Unit Testing

The entire system is built up in a very modular way, where each module represents a single concept by itself. As each module was developed a test program was built for each module to validate that the individual module was functioning correctly. For example, the *LinkedList* class was tested by applying every possible combination of operations in every possible order, the output of which was human validated.

6.3.2 Integration Testing

When a complete package of modules was completed, they were integrated into the system, and tested together. Since the system is built in a generic way, with a set of interfaces that link different parts of the system together. Some of the defects with the integration stage could be detected at compile time, when the implementation of the abstract classes was not correct. The actual communication and usage of the interfaces were checked to see if the packages were working well with each other and matching the design specifications in chapter 4. There were a total of nine separate packages in this project, the Pineapple Object Interaction Framework was one package, each of the plug-ins was another package, and each platform was a package. Minimal coupling between the various packages made the integration testing a clean and simple process, and each of the plug-ins were implemented completely independent of each other.

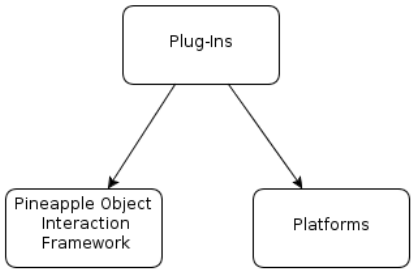

Figure 6.1: Integration dependency graph

However, validating that the packages can be integrated correctly is not enough, there are various other defects that are difficult to detect that need to be tested for. One such defect is a memory leak, where memory allocated by the system is not freed when it is not needed anymore, and is a common cause of software aging. For this, valgrind[2] was used to run the system in a sandboxed

[2]A GPL licensed programming tool for memory debugging, memory leak detection and profiling.

environment and to show any memory leaks upon exit of the program. Valgrind was also used to detect invalid read or write operations to memory, and for debugging segmentation faults.

To test the Nintendo DS developers usually have access to a Nintendo DS development kit, which includes a powerful hardware debugger that can sandbox the software in a similar way to valgrind. Since this hardware is unavailable for unofficial developers, a software debugger for analysing memory usage and finding memory leaks was developed and used to debug the Nintendo DS version.

6.3.3 Platform Testing

A key element to the design of this project was to build a completely platform agnostic solution, meaning that not only should it run on different platforms, but it should run in the *exact same* way. This was achieved by compiling the same game for different platforms and checking for any discrepencies between them.

6.3.4 System Testing

As this project grew, and more and more plug-ins were built, several small games were developed using the Pineapple Game Engine for demonstrative purposes. These games tested that the event driven model in the Pineapple Object Interaction Framework and the different plug-ins can work together to create small scale games, and any features that were needed but not present were analysed and refitted into the design to adhere to the user requirements. At the end of development a rather large project was undertaken to test if the Pineapple Game Engine can be used in real world applications, which is discussed in further detail in section 8.1.

6.4 Criticisms of Project Management

Due to the nature of this project, the designs were often reworked in the early stages of the project. It was only after a the design was iteratively improved on, that the separation of the Pineapple Object Interaction Framework and the platform agnostic component, as well as the integration of external modules was truly achieved.

Chapter 7

Technical Challenges

7.1 Pineapple Object Interaction Framework

7.1.1 Automation: Exploiting Implicit Template Instantiation

In C++ templates are used to create generic classes and functions that can be used with variable types of objects. This pattern is used to create generic containers and is prominently used in C++'s Standard Template Library (STL). For example, in STL there is a vector object which can store arbitrary types of objects such as *integers* and *strings* alike, just by specifying *int* or *string* as a template parameter.

At compile time, the actual templated class or function body is copied and adjusted accordingly for each possible template parameter. The way the compiler knows which templated versions to build is through *explicit* and *implicit* template instantiation. Explicit template instantiation involves the programmer explicitly defining that a class or function uses certain templated parameters. Implicit template instantiation is where the compiler will observe the program code and create each templated version on demand, as it is implicitly used. The latter method is often used as it involves less work for the programmer, and utilising this method is what makes the *Instance* static class completely automatic.

As a quick reminder, the *Instance* static class acts as a static store of information for game objects and is responsible for the following:

- *Creating* - All instances of game objects are created using the *Instance* class.
- *Storing* - The *Instance* class is where the *InstanceList* for the game object is stored.
- *Registering* - Ensuring the instance lists appear in the *InstanceManger*.

For each game object that the *Instance* static class is templated with, an *InstanceList* object needs to be created, and then this *InstanceList* object needs to be added to the *InstanceManager*'s master instance table. The only question is, how will the *Instance* static class know about

the existing game objects, without explicitly defining each game object as a template parameter to *Instance* using explicit template instatiation?

Looking back at how implicit template instantiation works, each game object needs to be *used* as a template parameter with *Instance* for a new templated version to be created. To make sure that this happens with every game object, the decision was that game objects can only be created using the *Instance* static class with that game object as a parameter. Once the compiler notices that the game object has been used as a template parameter with the *Instance* static class, it will create a parametrised version specific to that game object. Then all of the static variables which are *used* in this new version will be instantiated, including the *InstanceList* object and an *InstanceManager* object. The *InstanceManager* object simply adds the *InstanceList* object to the master instance table in its constructor. The *InstanceManager* object was not actually initially used in the class, it was only included so that it can be constructed. This meant that in order to actually instantiate the *InstanceManager* object, a single mention of the object had to be included in every function of the *Instance* static class, which read:

```
(void)instanceManager;
```

The void cast is necessary only to eliminate warning messages from the compiler that the statement is not doing anything, when in fact it is doing everything!

This also has the added benefit that the *Instance* static class only stores information on game objects which have been created.

7.2 Nintendo DS

This section will discuss the various technical challenges associated with the development of the Nintendo DS version of the Pineapple Game Engine.

7.2.1 Hardware Specification

This section will detail the technical properties of the Nintendo DS system.

- *Two TFT LCD Screens* - The Nintendo DS features two screens both of the same resolution: 256 pixels by192 pixels, and bit depth: 18-bit (262,144 colours). The bottom screen is also touch pressure sensitive.
- *4MB Main RAM* - This is the main RAM for the Nintendo DS, and has to hold the entire executable, as well as any memory during runtime that the application is using. When the final demonstration game for the Pineapple Game Engine was first compiled for the Nintendo DS, it failed because the executable would not fit into 4MB. However, this was largely due to the resources such as sound and graphics data that were embedded into the executable. To solve this problem the quality of some of the sound effects and images was lowered.

- *656kB VRAM* - This is where all of the graphics data, including the textures, sprite and backgrounds associated with the application are stored. To give the Nintendo DS version of the Pineapple Game Engine the same graphics capabilities as the desktop version special memory managing algorithms were developed, which will be covered in depth in section 7.2.3.

- *67.028 MHz ARM946E-S* - This is the Nintendo DS's main processor, where most of the application will be executed. Compared to a 2.5GHz processor on a desktop computer, this processor only runs at about $1/40^{th}$ of that speed, requiring highly optimised code. Performance was considered throughout the entire implementation stage, and so the Nintendo DS version was realised at the full 60 frames per second.

- *33.514 MHz ARM7TDMI* - This is the second processor present on the Nintendo DS, and is primarily used for performing background tasks such as handling user input, streaming audio data and providing access to the system clock.

- *No Floating Point Unit* - There is no support for floating point operations on the Nintendo DS. There is software emulation available for floating point operations, but this is far to slow to be viable. Fixed point integer operations have to be used instead. This was the case for the physics engine, where all floating point vector operations were replaced with fixed point counterparts.

7.2.2 Graphics Capabilities

The Nintendo DS features two separate graphics cores, the *main* core, and the *sub* core. The main core can be used for both 2D and 3D graphics, wheras the sub core can only be used for 2D graphics. Each graphics core can render to either screen, and they can each be set in different modes of operation. For the various modes of operation that are available for each core see appendix D. It is important to note that 3D graphics cannot be delivered to each screen simultaneuosly at the full 60 frames per second, since there is only one 3D core. However, it is possible to render 3D graphics to both screens at the same time at *half the frame rate*, since the 3D core has to render to one screen first, then render to the other screen second. This is not usually done in practice because of this restriction.

For the scope of this project, only 2D graphics needs to be considered. When the main core is in a 2D mode it behaves in exactly the same way as the sub core however, there is a number of restrictions with the 2D graphics capabilities. Each core in a 2D mode has the following limitations:

- *128 Sprites* - Only 128 sprites can be displayed on the screen at any one time. This is different to how many can be stored in memory at any one time. 128 sprites refers specifically to the Object Attribute Memory (OAM) in the system. Each sprite has a position, the option of being flipped horizontally or vertically, a visible flag, and various

other attributes that tell the Nintendo DS what to draw and where to draw it on the screen.

- *32 Affine Matrices* - Each matrix represents an affine transformation that can be used to rotate, scale, and shear sprites. Multiple sprites can share the same matrix, but multiple matrices can not transform the same sprite.
- *16 Sprite Palettes* - Each palette consists of only 16 16-bit colours, and can be shared between several sprites. To save on sprite graphics memory, it is common practice to swap the palette of a sprite to make it appear a different colour. An example of this is shown in figure 7.1.
- *16 Extended Sprite Palettes* - These palettes are used in a similar fashion to regular sprite palettes, but can hold up to 256 16-bit colours.

Figure 7.1: A classic example of palette swapping in the Mortal Combat series.

It is not desirable to have these limitations applied to both the top screen and the bottom screen, so it was decided for the main core to be set in a 3D mode, and to blit 2D sprites to the selected screen using 3D primitives. This allows one screen to far surpass the set of limitations above.

7.2.3 Organising Video Memory

7.2.3.1 Bank Mapping

The Nintendo DS only has 656kB of dedicated video memory, which has to hold all of the textures, colour palettes, fonts and background graphics for the application. The VRAM is split up into nine consecutive blocks, labelled VRAM banks A - I. The first four banks are the largest, each with a capacity of 128kB, which are mainly used for 3D textures, background and

sprite graphics. The last five banks are usually used for colour palette data. The various VRAM banks can be set to different modes of operation depending on what needs to be achieved. For example, in a 3D adventure game the optimal memory bank mapping might be to use VRAM banks A - D for 3D textures for the main engine, and for the sub engine: map VRAM banks H and I for the background and sprites respectively.

For the Pineapple Game Engine all but one bank is utilised, the resulting mapping is as follows:

- Main Core (Mode 0, 3D)
  - Bank A (128kB) → Textures
  - Bank E (64kB) → Texture Palette
  - Bank B (128kB) → Background
  - Bank G (16kB) → Background Ext Palette
- Sub Core (Mode 0)
  - Bank D (128kB) → Sprites and Palettes
  - Bank I (16kB) → Sprites Ext Palette
  - Bank C (128kB) → Background
  - Bank H (32kB) → Background Ext Palette
- Bank F (16kB) → Unused

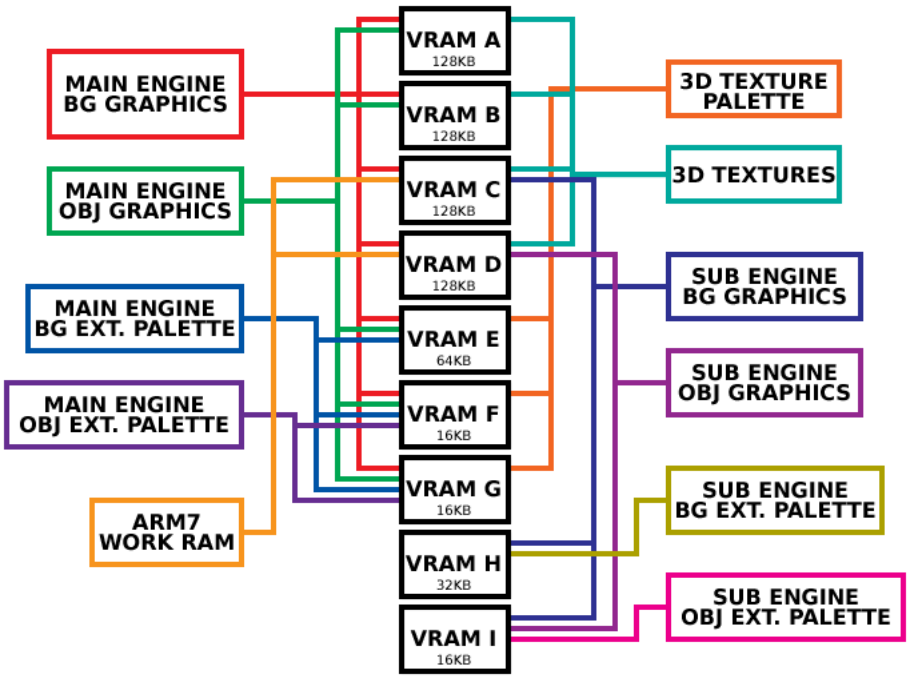

Figure 7.2: A diagram showing the different VRAM mappings possible for the Nintendo DS.

7.2.3.2 Managing Resources

A lot of the resources in the Nintendo DS such as sprite palettes and sprite entries have a fixed upper bound on the number that can exist at any one time. These limitations have already been discussed in section 7.2.2, and this section presents a solution to efficiently manage these resources in memory.

To use the limit of 32 affine matrices as an example, each matrix has a corresponding identifier that ranges from 0 to 31. If no two matrices can have the same identifier at any one time, how does one correctly choose an identifier that is not currently in use? A naïve solution would be to perform a linear search on the whole list of available identifiers to find an unused one. On a system such as the Nintendo DS, where every clock cycle counts, every possible optimisation must be made to achieve a smooth framerate.

The method that has been used in the Pineapple Game Engine has constant time complexity for both finding an available identifier, and for freeing an identifier when it is no longer in use. The algorithm is presented in the form of an efficient data structure, named an "Index Buffer". This data structure is structurally similar to a *circular buffer*. The buffer is initially filled with all available identifiers, so for the affine matrices, it is filled with values from 0 through to 31. Two pointers are used to point to the start value and the end value of the buffer. When an identifier is requested, if the buffer is not empty, the value at the start of the buffer is popped out, and the start pointer is incremented by one. When an identifier is released back into the buffer, the end pointer is incremented by one, and the value of the identifier is saved to that location. Provided the integrity of the identifiers is not compromised during transit, then this makes for an extremely efficient solution.

This data structure is not called an identifier buffer because that would imply that its usage is limited to resources that have a fixed number of identifiers. The index buffer can also be used to refer to locations in memory, by using a memory offset with a memory multiplier, one can easily manage a set number of pages in a contiguous block of memory.

The index buffer structure is used to manage:

- Sprite Engine
  - Entries in the OAM
  - Palettes
  - Ext Palettes
  - Affine Matrices
- Map Engine
  - Tile Memory

7.2.4 Sprite Engine

This section will detail the implementation of the sprite engine in the Nintendo DS version of the Pineapple Game Engine.

7.2.4.1 Assigning Graphics Cores

To ensure that the Nintendo DS version behaves in the same was as the Windows and Linux versions, not only must the resources of the DS must be used wisely, but a system for correctly managing both of the screens must be defined.

The top screen of the Nintendo DS is the one which is usually used for the projection of the main game, whilst the bottom screen is often used for displaying components that are secondary to the game such as a heads up display or an in game map. For this reason, the Nintendo DS's most powerful graphics core, the main core, was assigned to the top screen, while the sub core was mapped to the bottom screen.

7.2.4.2 Texture Loading

The two graphics cores of the Nintendo DS accept data in two different ways. The main core uses bitmap graphics for rendering textures in three-dimensional space, whilst the sub core takes in tiled graphics to draw sprites in a series of blocks. The actual per pixel data is the same, the only difference is that the order in which the pixels are arranged in memory. For bitmap data the pixels are ordered in horizontal rows starting with the top left corner of the image, and finishing at the bottom right corner. Tiled data is stored as a series of contiguous bitmaps 8 pixels by 8 pixels in size, and these "tiles" are arranged in memory row by row starting from the top left tile and finishing with the bottom right tile.

The screen that the sprite is created on will determine the type of formatting that is required by the sprite graphic. It is impractical to store both formats of every texture in the executable as the main RAM of the Nintendo DS is extremely limited. This led to the idea of embedding a single format in the executable, and generating the other format at runtime when required. Since the main graphics core is estimated to be used the most, the texture data of every sprite is compiled into the executable in bitmap format, and when the texture is needed on the bottom screen, the tiled data is algorithmically calculated from the bitmap data.

As well as having two separate texture formats, there is another texture attribute that cannot be ignored. For both bitmap and tiled graphics data, two different bit-depths exist, measured in bits per pixel (BPP). These bit-depths are 4BPP (16 colours) and 8BPP (256 colours). Graphics data in 8BPP format takes up twice the space in memory as graphics data in 4BPP format. A significantly higher quality can be seen from 8BPP sprites compared to 4BPP sprites, so it is important to choose the correct format for each sprite. This choice of bit-depth is left to the developer at the application level, when creating an *NdsTexture* resource.

7.2.4.3 Main Screen Sprite

The Nintendo DS's main graphics core is a similar state machine to the model OpenGL uses, making the blitting of 2D sprites somewhat of a simple task. Sprites are drawn to the screen using 3D primitives to render 2D images in a similiar fashion to that of the Windows and Linux platforms. Rendering sprites to the main screen was not exactly a technical challenge but is mentioned for the completeness of this section.

7.2.4.4 Sub Screen Sprite

Sprites on the bottom screen are read from the OAM once per frame, it is vital that the OAM is kept up to date with the correct information. Each frame, each sprite in the Pineapple Game Engine is processed and checked. Using the index buffer, OAM entries and affine matrices can be distributed correctly to every sprite that requires them. All that was needed was to decide if each sprite needed them. For the OAM entry (to actually display it on the screen), a simple boundary check is executed each frame to determine if the sprite is in the current viewport. For the affine matrices, if the scale or rotation properties were not the default, then an matrix would be acquired. The default values being (1, 1) for the scale factor and zero for the rotation.

7.2.4.5 Dual Screen Sprite

Main screen and sub screen sprites work well, but what if the developer wanted to have a sprite that could be thrown from screen to screen? It was this thought that provoked a *dual screen* sprite type. Dual screen sprites are simply made up of a main screen sprite and a sub screen sprite. The dual screen sprite manages the attributes of both sprites to provide, what appears to be, a single sprite seamlessly travelling from screen to screen.

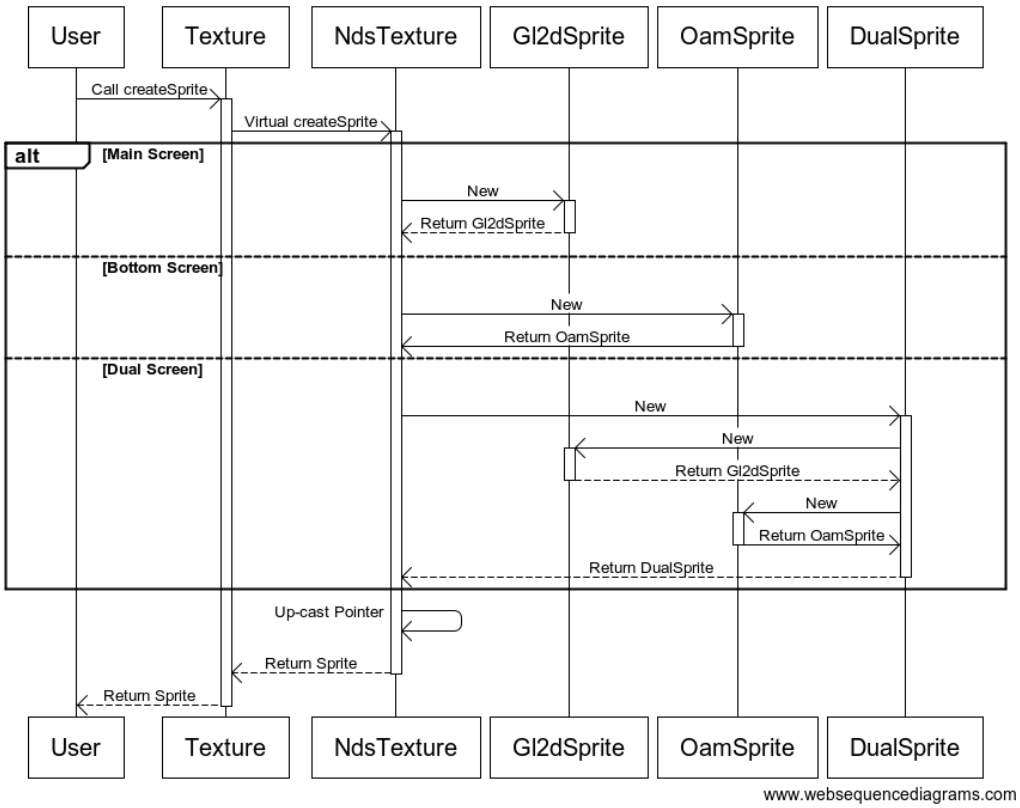

Figure 7.3: Abstract sprite creation on the Nintendo DS

7.2.5 Map Engine

This section will detail the implementation of the map engine in the Nintendo DS version of the Pineapple Game Engine.

7.2.5.1 Hardware Backgrounds

Each graphics core features four hardware backgrounds that are controlled in the same way on each core. When the main core is in a 3D mode, then it must use its first background as a 3D framebuffer. The background control engine in the Nintendo DS is structurally similar to that of the Pineapple Game Engine's map engine, its just *a lot smaller*. It consists of tiled[1] background graphics formatted in either 4BPP or 8BPP, and hardware maps consisting of tile entries that dictate where tile graphics are to be placed on the screen.

7.2.5.2 Tile Entries

Each tile entry is 16 bits in size, has a tile index, and horizontal and vertical flipping flags, just like the tile entries in the Pineapple Game Engine. The only difference is that in the Nintendo

[1]This is the same 8 pixel by 8 pixel tiled graphics that is used by sprites.

DS 4 bits are used for selecting a colour palette, which are only used in the 4BPP format. In the 8BPP format these bits are unused, since in the 8BPP format only one colour palette is used per background. The 8BPP format will be used in the map engine to achieve the best image quality, as the 4BPP format is inadequate for backgrounds with many colours.

To compare the format of the Nintendo DS tile and the Pineapple Game Engine tile:

	16 bits															
Tile Entry	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Nintendo DS	Palette				VF	HF	Tile Index (0 - 1023)									
Pineapple Game Engine	HF	VF	Tile Index (0 - 16383)													

7.2.5.3 Tile Bases and Map Bases

In the Pineapple Game Engine, the main graphics core uses VRAM bank B for background memory, and VRAM bank G for its colour palette, while the sub graphics core uses VRAM banks C and H. Both cores behave in exactly the same way with respect to background rendering.

Background memory is divided up into overlapping sections of tile and map memory, which is only referencable through *tile bases* and *map bases* respectively. Each 2kB map base has 32 by 32 tile entries, and each 16kB tile base can hold up to 256 tiles in a 8BPP format. The map bases can only range from 0 to 31, which means only the first 64kB of the background memory can be used for map data, but tile bases span the whole 128kB.

Memory Offset (kB)	Map Base (0 - 31)	Tile Base (0 - 7)
0	0	0
2	1	
4	2	
6	3	
8	4	
10	5	
12	6	
14	7	
16	8	1
18	9	
...	...	...

Figure 7.4: VRAM Bank B (128kB) in Background Memory mode.

Correctly placing tile graphics and map data alongside each other is absolutely critical. To decide how much memory to allocate for each, a few calculations were carried out.

Firstly, the total amount of tile memory that will be needed was calculated. If each 8BPP tile takes up 64B, and each background in a game can have up to 2^{14} (16384) unique tiles, and each screen must be able to have up to two backgrounds at any one time, then the maximum amount

of tile memory that would use is *2MB per screen*[2]. Secondly, the total amount of map memory necessary cannot be actually calculated, since background maps are allowed to be infinitely big in the Pineapple Game Engine.

So to adhere to the requirements of this project then that 2MB of tile data and an indefinite amount of map data must be streamed in real-time into the Nintendo DS's memory bank of 128kB at a playable frame-rate, for each screen.

7.2.5.4 Squeezing Memory

Is it really possible to stream so much data into the Nintendo DS, and still have it running smoothly?

As mentioned earlier, the screen of the Nintendo DS is 256 pixels by 192 pixels in size, which is equivelant to 32 tiles by 24 tiles. However, if partially visible tiles are being counted, a maximum of 33 by 25 tiles can be seen at any one time. To display a full screen of unique tiles will require 825 tiles to be loaded into memory at once for each background.

The chosen map base(s) and tile base(s) must have sufficient storage space to be able to display a single *region* of game's foreground and background at any one time. As the background is moved relative to the game's viewport, the correct tile graphics and map data must be dynamically loaded in to properly render it.

Two map bases are needed for each background since each map base is 32 by 32 tiles, which is only 1 tile too small on the horizontal to display 33 by 25 tiles. With 256 tiles per tile base, 3.22 tile bases[3] are required out of the 8 that are available. The overlapping memory model in figure 7.4 can be taken advantage of. As previously mentioned, the map bases span the same region of memory as the first four tile bases, and so 3.22 tile bases can fit into 27 map bases. Placing the first background at tile base 0 leaves map bases 27 to 31 available for map data, of which we only need two map bases per background, which means we need four of these. The second background can be placed at tile base 4 with no worries. That leaves one map base to spare, and some more tile graphics memory at the end of tile base 7, since the second background won't use all of the remaining four tile bases. This was used for a debug console for the entire application which only needed one map base, and 4kB of tile graphics memory for the 128 ASCII characters.

[2] 2 backgrounds * 64 bytes * 2^{14} tiles.

[3] 825 tiles divided by 256 tiles per tile base to three significant figures.

Memory Offset (kB)	Map Base	Tile Base	Used For
0	0	0	Foreground Tile Graphics (54kB)
2	1		
...	...	...	...
52	26		
54	27		Foreground Map (4kB)
56	28		
58	29		Background Map (4kB)
60	30		
62	31		Debug Console Map (2kB)
64	-	4	Background Tile Graphics (54kB)
66	-		
...	...	...	...
110	-		
112	-	7	
114	-		
116	-		
118	-		Debug Console Tile Graphics (2kB)
120	-		
122	-		Unused (6kB)
124	-		
126	-		

Figure 7.5: The resulting map and tile base allocation for storing a foreground, background and a debug console.

7.2.5.5 The Algorithm

The algorithm works by removing tiles that are no longer in the current viewport, but were in the previous one, and removing them. It then looks at the tiles that are in the current viewport, but were not in the previous one, and placing them into map memory, loading the corresponding tile graphics into tile memory if necessary. It stores the tiles that are currently loaded in a cache, which is implemented as a hash map. It runs with a time complexity of $O(nlogm)$ where n is the number of tiles that need to be swapped out, and m is the number of tiles currently in the cache. A visualisation of the algorithm running is shown in appendix E.

Algorithm 1 updateBackground(P, V)

Require:
P = set of all tile positions in previous view
V = set of all tile positions in current view

```
for all p⃗ ∈ (P \ V) do
  β ← hardwareTiles[p⃗]
  count[β] ← count[β] − 1
  if count[β] = 0 then
    α ← memory[β]
    cache.erase(α)
    indexBuffer.release(β)
  end if
end for
for all p⃗ ∈ (V \ P) do
  α ← applicationTiles[p⃗]
  if cache.find(α) then
    β ← cache.get(α)
  else
    β ← indexBuffer.acquire()
    cache.insert(α, β)
    hardwareTileMem[β] ← applicationTileMem[α]
    memory[β] ← α
  end if
  count[β] ← count[β] + 1
  hardwareTiles[p⃗] ← β
end for
```

Chapter 8

Conclusion

This section concludes the work carried out during the project.

8.1 Demonstration

A large scale game was developed to demonstrate that the functionality included in the Pineapple Game Engine was suitable for creating games, screenshots of which can be viewed in appendix F. The game consisted of two unique levels, that were formulated to test two individual genres of 2D games, an overhead shooter, and a platformer.

The game starts with a cutscene, where a lone spaceship is ambushed by a group of alien spaceships, consequently forcing the player to intervene and command the spaceship to eradicate the enemy ships. After a couple of waves of enemy ships have been defeated, the player is introduced to the queen ship, which is a carefully scripted boss battle. Once the player thinks that the boss ship has been defeated, the boss fires out one last laser beam which causes the players ship to hurtle towards the earth and subsequently crash land.

Once the player has crash landed, the scene is changed to that of a platformer, where the player, who is shown to be a fighter droid robot, has survived the crash and now has the opportunity to explore the vast landscapes in this section. The robot faces multiple semi-intelligent dragon enemies before facing the dragon boss, which is also another carefully scripting boss battle.

Throughout the game, the following game specific concepts are demonstrated:

- Animations
- Behaviour
- Artificial Intelligence
- States

- Interactions
- Attributes
- Inheritance

The game was designed, developed and tested in an extremely short time (less than two weeks), and was fully featured with animated graphics and sound effects. The game was first written with the desktop platforms in mind, and when it was ported to the Nintendo DS, it had to be scaled down to fit the smaller screens. This involved scaling the textures down to half the size, and altering other specific attributes that were related to the size of the viewport of the game, such as the dragons visual perception distance. The quality of some of the sound resources also had to be lowered in order for the executable to fit in the 4MB of main RAM that is offered by the Nintendo DS. These kind of modifications are inevitible when porting a game from a more capable platform to a less capable one. However, it would be beneficial if there was some way that the whole game could be automatically scaled appropriately to adhere to the requirements of the platform it is compiled for.

There was one feature that wasn't present in the Pineapple Game Engine that was needed in the platformer level. That was collision masks for large world scenes. This feature had to be programmed in by myself during development, however by doing so shows that the game engine is easily extendible.

8.2 Discussion

A working solution to the problem has been developed and demonstrated, showing that it was possible to achieve the goals and objectives set out in the in the requirements. I have shown that it is possible with careful design and efficient implementation, and I have also made the system completely extensible and compliant for third party developers.

However, the sheer size of this project meant that there was not enough time to completely optimise the solution. Even though the solution as it stands is optimised to a certain degree, there have been a few well known optimisation techniques that have been overlooked. A notable one to mention is the pooling[1] of objects to reduce the number of allocations and deallocations per second.

Since this game engine is minimal, fast and platform independent it is most suitable for the design stage of multi-platform software projects, where ideas can be turned into something concrete on the screen and prototyped in a matter of hours. However, with commercial plug-ins, this game engine can be deployed and used in many mainstream projects. It can also be used as a tool for porting video games from one platform to another, in similar way to SDL and Allegro.

[1]Creating a fixed amount of objects during start up, then reusing these throughout the execution of the program.

8.3 Relevance to Computer Science

The main goal of this project was to find a more efficient method to current scripting language based systems. Using modern high level programming features as well as designing efficient data structures and algorithms was key to finding the solution. This project has introduced new system design techniques and modern programming paradigms that have the potential to be used to create more dynamic software solutions.

8.4 Summary of Contributions

This dissertation introduces the idea of using modern concepts in C++ programming to develop event driven games in a completely platform agnostic environment. The Pineapple Object Interaction Framework itself can be used as a cross-platform base for many applications outside the field of video games, including simulations and graphical applications. The automatic processing of object specific functionality using implicit template instantiation as described in section 7.1.1 is also noteworthy.

This is also the first project, to my personal knowledge, that has fully bridged the gap between the Nintendo DS platform, and desktop environments, allowing an abstract game to be defined, and ported between platforms without any problems. In fact, the only issue with porting from PC to Nintendo DS is that the hardware limitations on the Nintendo DS need to be taken into consideration, such as the graphics memory available, although this project has shown that it is possible to squeeze large bitmap images into the Nintendo DS in section 7.2.5.

8.5 Further Work

Implementations for a wide range of platforms could be built, and since this project is written in C++, there are a large number of platforms that could be supported, including PlayStation 3, Android, and iPhone. There is also room for an integrated development environment to be created around the game engine to make the process of setting up projects, compilation, and managing resources easier.

To add further functionality to the game engine, additional plug-ins could be developed that cover more game specific concepts. These plug-ins could add more types of events to the system, and provide the developer with a wider set of objects to utilise. Such plug-ins include:

- Artificial Intelligence
- Navigation
- Internet Multi-player

- Text Rendering
- 3D Graphics
- More Sophisticated Physics
- Threads

Concerning the platform specific stages of games in the Pineapple Game Engine, manually loading in the correct resources during startup is somewhat of a chore, and could be replaced by auto-generating such a file. A way to auto-scale a game to fit the current platform is also a good idea for this project, as it would relieve the developer from having to pursue in such tasks manually.

Bibliography

[1] Oracle. 2010. URL: http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html (Retrieved on 18/04/2012).

[2] *A GNU Manual*. Free Software Foundation Inc. 2010.

[3] Jaeden Amero. *Introduction to Nintendo DS Programming*. 2008. URL: http://static.patater.com/files/projects/manual/manual.html (Retrieved on 19/04/2012).

[4] OpenGL Architecture Review Board et al. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*. 6th. Addison-Wesley Professional, 2007. ISBN: 0321481003, 9780321481009.

[5] *Compare GameMaker versions*. YoYo Games. 2012. URL: http://www.yoyogames.com/make (Retrieved on 23/04/2012).

[6] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[7] Michael Dawson. *Beginning C++ Game Programming*. Ed. by Mitzi Koontz. Premier Press, 2004, p. xi.

[8] *FMOD Licenses*. Firelight Technologies. 2012. URL: http://www.fmod.org/index.php/sales (Retrieved on 21/04/2012).

[9] Brent Fulgham. *Computer Language Benchmarks Game*. 2004. URL: http://shootout.alioth.debian.org/ (Retrieved on 08/04/2012).

[10] Philip Gamble. *GameMaker 8.1 Decompiler Released*. Game Maker Blog. 2011. URL: http://gamemakerblog.com/2011/06/14/gamemaker-8-1-decompiler-released/ (Retrieved on 10/04/2012).

[11] Jacob Habgood and Mark Overmars. *The Game Makers Apprentice: Game Development For Beginners*. Ed. by Chris Mills. Pearson, 2006.

[12] Jonathan S. Harbour. *Game Programming All In One*. Ed. by Jenny Davidson. Second. Thomson Course Technology, 2007.

[13] *History of Game Maker, 1999*. YoYo Games. URL: http://wiki.yoyogames.com/index.php/Game_Maker_History#1999 (Retrieved on 10/04/2012).

[14] *History of Game Maker, 2007*. YoYo Games. URL: http://wiki.yoyogames.com/index.php/Game_Maker_History#2007 (Retrieved on 10/04/2012).

[15] ISFE. *Video Gamers in Europe*. 2010.

[16] Jeff Wilson Jason Busby Zak Parrish. "Introduction to Unreal Technology". In: *InformIT* (2009).

[17] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006. ISBN: 0-321-29535-8.

[18] Martin Korth. *GBATEK - Gameboy Advance / Nintendo DS - Technical Info*. Tech. rep. 2007.

[19] Sam Lantinga. *SDL version 1.2.15 (stable)*. 2012. URL: http://www.libsdl.org/download-1.2.php (Retrieved on 24/04/2012).

[20] *Legal Information (Copyrights, Emulators, ROMs, etc.)* Nintendo. 2012. URL: http://www.nintendo.com/corp/legal.jsp (Retrieved on 09/04/2012).

[21] Robert C. Martin. "Java and C++: A critical comparison". Mar. 1997.

[22] *Nintendo DS Frequently Asked Questions*. Nintendo. URL: http://www.nintendo.com/consumer/systems/ds/faq.jsp#ds (Retrieved on 12/04/2012).

[23] Mark H. Overmars. *GAME DESIGN IN EDUCATION*. Tech. rep. 2004.

[24] Mark H. Overmars. "Teaching Computer Science through Game Design". In: *Computer* 37 (2004), pp. 81–83.

[25] Mark Overmars. *Designing Games with Game Maker*. 7.0. YoYo Games.

[26] Theo Pavlidis. "Computer Game Basics (Draft)". 2006.

[27] *Simple Directmedia Layer Official Website*. URL: http://www.libsdl.org/ (Retrieved on 23/04/2012).

[28] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2011.

[29] Tim Sweeney. *Unreal Language Reference*. Epic Games. 1998.

[30] *The difference between instance ids and indexes*. YoYo Games. 2011. URL: http://wiki.yoyogames.com/index.php/The_difference_between_instance_ids_and_indexes (Retrieved on 23/04/2012).

[31] *The Unreal Editor*. Epic Games. 2012. URL: http://www.unrealengine.com/features/editor/ (Retrieved on 23/04/2012).

[32] *Unreal Engine: Platforms*. Epic Games. 2012. URL: http://www.unrealengine.com/platforms (Retrieved on 23/04/2012).

Appendix A

UML Diagram

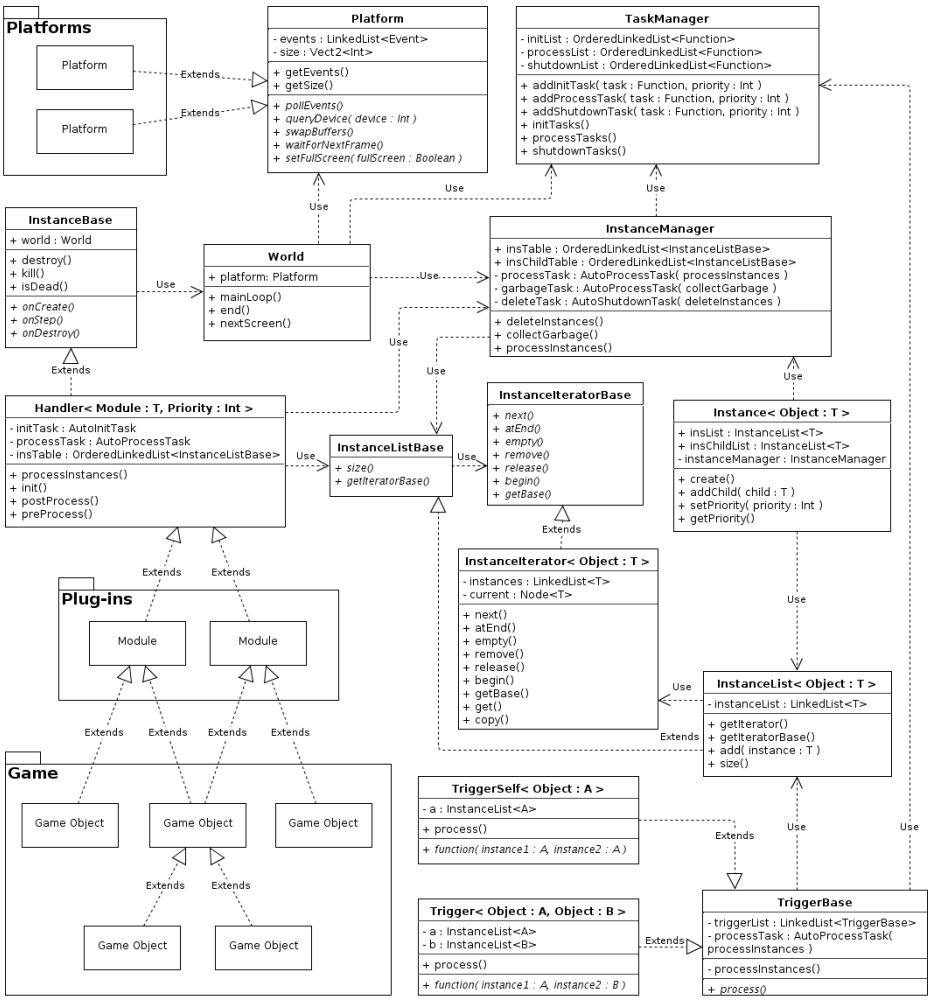

Figure A.1: A UML digram of the Pineapple Object Interaction Framework

Appendix B

External Module Design

Composition

```
class Object {
    Module1 module1;
    [Module2 module2; ...]
    Object() {
        module1.eventHook<Object>(&eventFromModule1);
        [module2.eventHook<Object>(&eventFromModule2); ...]
    }
    void function() {
        module1.aFunction();
        [module2.aFunction(); ...]
    }
    void eventFromModule1() {
        // actions
    }
    ...
};
```

Inheritance

```
class Object : public Module1 [,public Module2 ...] {
    void function() {
        aFunctionFromModule1();
        [aFunctionFromModule2(); ...]
    }
    void eventFromModule1() {
        // actions
    }
    ...
};
```

Appendix C

AABB Collision Algorithm

Algorithm 2 overlapping(A, B, V)

Require:
A = first AABB
B = second AABB
V = a vector to will store the response

$left \leftarrow B.x2 - A.x1$
$right \leftarrow A.x2 - B.x1$
$bottom \leftarrow B.y2 - A.y1$
$top \leftarrow A.y2 - B.y1$
if $left > 0 \wedge right > 0 \wedge bottom > 0 \wedge top > 0$ **then**
  $shortest \leftarrow left$
  $response \leftarrow (-left, 0)$
  if $right < shortest$ **then**
    $shortest \leftarrow right$
    $V \leftarrow (right, 0)$
  end if
  if $bottom < shortest$ **then**
    $shortest \leftarrow bottom$
    $V \leftarrow (0, -bottom)$
  end if
  if $top < shortest$ **then**
    $V \leftarrow (0, top)$
  end if
  return $true$
else
  return $false$
end if

Appendix D

Nintendo DS Video Modes

The various modes of operation each graphics core of the Nintendo DS can be set into.

Main Core				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM Bitmap Display			
Sub Core				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

Background Type	Scale	Rotate
Text	No	No
Rotation	No	Yes
Extended	Yes	Yes

Appendix E

Nintendo DS Map Engine

A visualisation of the algorithm that is used to drive the map engine for the Nintendo DS version of the Pineapple Game Engine. From left to right, the Nintendo DS map memory (2 * 32 * 32 tiles), the tile memory (832 tiles), and the user's screen (256 by 192 pixels). The red box in map memory indicates where the Nintendo DS is currently rendering the screen background data from.

Appendix F

Demonstration Screenshots

The same game running on Linux and Windows on the left, and on the Nintendo DS on the right.