Ergophobia

The Silence Of Night

Masters Thesis

Matt Osbond

Ben Chandler

Hasan Atieh

Ali Derweesh

N.C.C.A Bournemouth University

September 9, 2007

Abstract

With a shared desire to create a functional video game, four students decided to do so for their MSc Term 4 Project. They were given the opportunity to create such a piece in a professional working environment through entry into an internation video games competiton called 'Dare to be Digital'. Upon clearing the first stage, the team went to work for the 10 weeks in an industry development studio. The result is 'The Silence of Night', a third person ninja game. The game had strong media coverage for the duration of the competition and won appraisal for it's use of interesting technologies and it's artistic style.

Contents

List of Figures

List of Tables

Chapter 1

Introduction.

1.1 Document Overview

A successful video game must have an almalgamation of talent, organisation and creativity. The team was made up of a mixture of individuals who each posess these traits, and therefore a final product was realised. This document is an outline of the production process, from an overview of the engine used to description of the core technologies that were developed. Each member of the team has written their own specialist chapters, as well as providing input to the remainder of the document.

1.2 Introduction

The video games industry has been enjoying a consistent rise in popularity in recent years ([1]), and this is reflected by the number of successful student games projects. Dare to be Digital was started in 2000 to give support to these students by providing not only the means but also the motivation. Until 2007 it was a competition only open to Scottish students, but this has now been changed to allow students from England and Ireland to participate.

Team 'Ergophobia' consists of four MSc Computer Animation students and one BA Computer Animation and Visualisation Student. After successfully getting through to the stage where development began, they moved to Electronic Arts' studio in Guildford for 10 weeks in order to create their game.

'The Silence of Night' is a third person game based in feudal Japan, in which the player has to reach a target while avoiding detection by enemeies through the use of the stealth.

The game will use two new technologies: a unique sound visualisation system and intereactive geometry cutting.

Various game engine were considered for use in the project, such as Ogre, Renderware and the more simplistic OpenGL. After much research the team decided on the usage of Instinct, an engine developed by Instinct Technologies. This was technically a beta release, as the engine itself is not available to purchase at the time of writing.

The remainder of the following two chapters is, for the most part, taken from the Instinct Studio documentation. [5][6]

1.3 Architecture overview

The Instinct architecture aims to provide the following features:

- Stable framework for rapidly evolving game software
- Highly integrated tools and runtime
- Efficient use of hardware resources
- Multi-platform support (Win32/64 PC and Next-Gen consoles)

1.3.1 Modules.

The Instinct API is composed of a number of code Modules. Each Module is a set of code that provides a distinct set of services. Modules may depend on other Modules and may be platform dependent. The engine can be extended by the addition of new Modules at compile time or runtime. Modules developed by the Instinct team are prefixed with the letters "ie". Some examples: ieCore, ieGraphics, iePhysics.

1.3.2 API Layers

Instinct is organized into a hierarchy of layers.

1.3.3 Core Layer

This layer provides the base functionality for all Instinct code. It provides the following services, all of which are typically implemented in the ieCore Module:

Table 1.1: Core Layer Services

Service	Description
Memory management	Optimized alternatives to standard new and delete pooling structures
File and resource management	Binary & text reading and writing XML, CSV and other parsing functions
Module management	Loading and unloading Instinct Modules
Logging and error handling	
High resolution timers	
Code profiling	Timings and counters for function calls, Memory usage
Scripting	Command parsing and execution
System component management	
Entity management	Construction, configuration and destruction of entities Entity event management

1.3.4 System Component Layer

System components are C++ objects that typically provide interface-based access to hardware or operating system functions such as those provided by DirectX or Windows. An instance of a system component can be given a unique name. Such an object is known as a Component Instance. Common component instances include:

- File Manager
- Graphics Device
- Sound Channel Manager
- Input
- Command Mapper
- Diagnostics

System components can be scripted and component instances may also be configured using the system configuration file.

1.3.5 Entity Layer

Entities are data-driven objects that are composed of smaller objects called entity components. Entities are scriptable objects that are used to define the game world and may be edited using Instinct Studio. Common examples include light, sound, camera and player entities.

Users can specify the composition of an entity using entity templates. These entity templates act as blueprints from which entities may be created. An entity can only exist within an entity manager object. Every entity must have a unique name within its entity manager. Instinct allows multiple entity managers to exist at once but a typical game runs with a single entity manager.

1.3.6 Application Layer

Instinct applications are the programs that make use of the Instinct API, such Instinct Studio, 3D Studio Max Exporters and Instinct games. Note: Modules may provide functionality ranging across multiple layers.

1.4 Entities and Entity Components

The game world is modelled in Instinct as a set of objects called Entities. Each Entity has a unique name and exists within the context of an Entity Manager. Instinct can support multiple active Entity Managers but typically a game only requires one.

Instinct Entities are entirely composed of objects called Entity Components. Entity Components can be reused and combined in order to define many different types of Entities. For example, a crackling torch entity might be defined using light, sound and mesh entity components.

Entities to be used for the purpose of level construction are located within what the developers call a scene file. Instinct breaks up the file into two sections; primary entities and standard entities. The purpose of the primary entities is to provide the 'scaffolding' of the level, while the standard ones are the 'bricks and mortar'. For example the physics simulation entity (typically called 'priPhysics') is a primary entity, while all of the objects that are simulated are standard entities.

1.4.1 Entity Communication

Instinct provides a number of mechanisms for entity communication: C++ Interfaces direct access to virtual C++ methods using standard interface pointers. Scripting Interfaces via properties and commands (see below). Event Objects sending and receiving events.

1.4.2 Scripting

Instinct provides a framework to allow C++ developers to expose scriptable properties and commands for entity components with a minimum runtime overhead. Entities can be manipulated from the command line or script files using an object-oriented syntax: Copy Code

Listing 1 Script Example

```
Player.Health.MaxHealth 100
Sound.Manager.StopAllSounds
Enemy12.Health.TakeDamage 2 10
```

Instinct provides a number of in-built property types including boolean, integer, floating point, string, vector and quaternion. Developers can also create their own property types and register them with Instinct.

1.4.3 Entity Templates

Users can define the structure of games entities using entity templates. Each entity template contains a list of entity components along with the default property values for entities created using that template. Entity Templates can inherit structure and default property values from other templates. For example, the follow template definition describes the Chair Entity and Lampshade entities making use of a Base Entity Template called SimpleObject:Â

Copy Code

Listing 2 Entity Template Example

```
EntityTemplate
{
    _name = "SimpleObject"
    // Entity components
    _components = "RigidBody,Model,WorldPosition?
}

EntityTemplate
{
    _name = "Chair"
    // Parent templates
    _parents = “SimpleObject”
    // Default property values for this template
    RigidBody.physicsFile = “test/chair.psx”
    Model.meshFile = “test/chair.mesh”
}

EntityTemplate
{
    _name = "Lampshade"
    // Parent templates
    _parents = “SimpleObject”
    // Extra Entity components not in my parents
    _components = "Sound, Light”
    // Default property values for this template
    Sound.resource = “test/lampshade.wav”
    Light.type = “box”
    Light.extents = (2,2,1)
}
```

This data-driven approach for entity creation allows for rapid prototyping and allows uses to create their own entity types without having to program in C++.

1.4.4 Serialization

Instinct provides a framework for automatic entity loading and saving through entity component properties. Developers of entity components may also implement their own custom loading and saving routines if desired.

Instinct Studio Integration Entities are automatically editable inside Instinct Studio via exposed properties and commands. No extra code is necessary.

Chapter 2

Directory Structure.

Here is a brief description of the various file folders used in Instinct Studio.

2.1 Bin

The bin folder contains all compiled executables and DLLs, including compiled client code. The folder is further subdivided by platform & compiler. At the beginning of the project, there were two Win32 compilers supported: Microsoft Visual C++ 2003 (bin/x86_vc7) and Microsoft Visual C++ 2005 (bin/x86_vc8). This has changed over the course of the project and now only the latter has support from the developers.

When using the debug configuration in Visual Studio, the files will be compiled to the x86_vc8_debug folders. Similarly, using the retail configuration will compile the files to the x86_vc8_retail folder and the release configuration compiles to the x86_vc8 folder.

2.2 Env

This folder contains a variety of data that Instinct Studio uses to operate.

2.2.1 Env/Profiles

This folder contains information about each users configuration for Instinct Studio, such as window layout, user interface customizations and user preferences.

2.2.2 Env/Text

This folder contains xml files used to describe colour syntax highlighting for the different types of text files used in Instinct Studio.

2.2.3 Env/Screenshots

This folder is used to save out in-game screenshots. This can be done using a console command, typically bound to a shortcut key.

2.2.4 Env/Config

This folder stores the config files used to launch studio and the runtime, these can be overridden if required. These config files are used if no other config is specified. For example, when you run bin/x86_vc8/Studio.exe directly it will automatically use the config in env/config/InstinctStudio.cfg

2.3 Projects

A project describes the structure of the game, which is mostly a list of the packages that the game uses and some configuration info. According to the Instinct documentation, it is recommended that any projects created for the game be stored in this folder.

2.4 Packages

Packages are the mechanism used to organize assets in Instinct Studio. Examples of such asset can be scenes, templates, textures, models, audio files, etc. It is usual to arrange these assets in sub folders within a single package folder.

Three packages are provided by Instinct as standard: Base Reference SDK

2.4.1 Packages/Base

The Base Package contains essential content required to run Instinct Studio.

This folder contains the public includes and compiled libs for Instinct Studio so that one can link with and extend the functionality provided. Solution and project files for Microsoft Visual Studio 2003/2005 are available in the build folder.

2.4.2 Packages/Reference

The Reference Package demonstrates the suggested use of game functionality provided by Instinct Studio such as models, physics, etc. As new functionality is added to Instinct Studio, the Reference package is updated to demonstrate each new feature. This means that the assets contained here are liable to change as new versions of Instinct Studio are released.

According to the documentation provided with Instinct Studio, it is recommended to create a separate package for the game and store it in the "packages" folder as this is the only location where Instinct Studio looks for them. Also, the code written for the user defined components and applications should be stored in the users package folder.

2.4.3 Packages/Tools

This folder contains some useful tools which can use in conjunction with Instinct Studio, including the 3D Studio Max Exporter Plugin and NormalBumpMap-Merger tools.

2.5 Log

This folder contains log files generated by Instinct Studio. The logs contain a step by step list of commands executed and any errors or warnings that are generated. This can be useful when trying to diagnose problems with the game. Instinct allows users to output to the log by using the LogString() function. The contents of the log file can also be seen in the console window when running Instinct Studio.

2.6 Docs

This folder contains the documentation provided with Instinct Studio. This consists of the User Guide which provides help for content creators, and the Programming Manual which provides information for game programmers.

2.7 The Development Framework

In accordance with the development practices suggested by Instinct, a separate package was created for each team member and another for the game resulting in six packages. These are MO_Package, AD_Package, BC_Package, SH_Package, HA_Package in addition to the Game_Package where the first two capitals of the package name represents the initials of the owner of the specified package. Also a project called "Game_Project" was created to bundle together the six packages and the rest of the standard packages provided by instinct.

All the packages were kept in a shared folder where each team member had the ability to upload his own package to that folder and download the other packages including the game package. The Game_Package which contained the Game_Scene was updated by the level and environment designer.

2.7.1 The Team Members and Their Tasks

The roles of the team members were clearly defined with minor overlapping. The rest of this thesis will follow a similar approach where each chapter is written by a team member and represents his work on the game.

Each member had specific responsibilities within the team:

- Matt Osbond:

  Team Lead / Producer

  Environment Design and Modelling

  Sound Design

- Ben Chandler:

  Lead Programmer

  Graphics and Post-Processing

  Input And Character Controller

  Animation Blending

  Shader Design

- Hasan Atieh:

  A.I. Programming

  Physics Implementation

- Ali Derweesh:

  Real-Time Cutting Mechanism

  Concept Research

- Sebastien Huart:

  Character Design and Modelling

  Animation Cycles

  Concept and Graphics

Chapter 3

Project Management, Level Design, Environment

By Matt Osbond

3.1 Project Management

3.1.1 Outlining Production

The preparatory elements of the project pipeline were initially discussed in great detail as a team. Having previously created a game in term 2, we were already aware of the production process and all too familiar with the possible pitfalls of game development. It was important that these elements were taken into consideration when outlining the initial production schedule. The outcome was an overview of the entire production process that took into account the following:

- Two weeks of diluted workflow at the beginning of the project to account for overlap of projects. The main task for these weeks was for each member to get used to the game engine. We all had key roles that demanded us to have a good working knowledge of the Instinct engine, and these two weeks were used to traverse the learning curve.
- One week at the end of the project to allow for tweaking and polishing of assets and code.
- Week by week breakdown of tasks on an individual basis. This allowed everyone to see at a glance what the other members of the team were supposed to be doing.

The final point is possibly the most important, as a key to success in team-based project management is communication. The ability for each member to view the tasks of others was vital, as the inter-member dependencies were great within this project. For instance, cutting could not be tested until the correct geometries were created, or the character controller could not be started until animation cycles were produced. These dependencies were taken into account during creation of this initial production outline.

The final initial Schedule of Production can be found in Appendix A

3.1.2 Management Methods

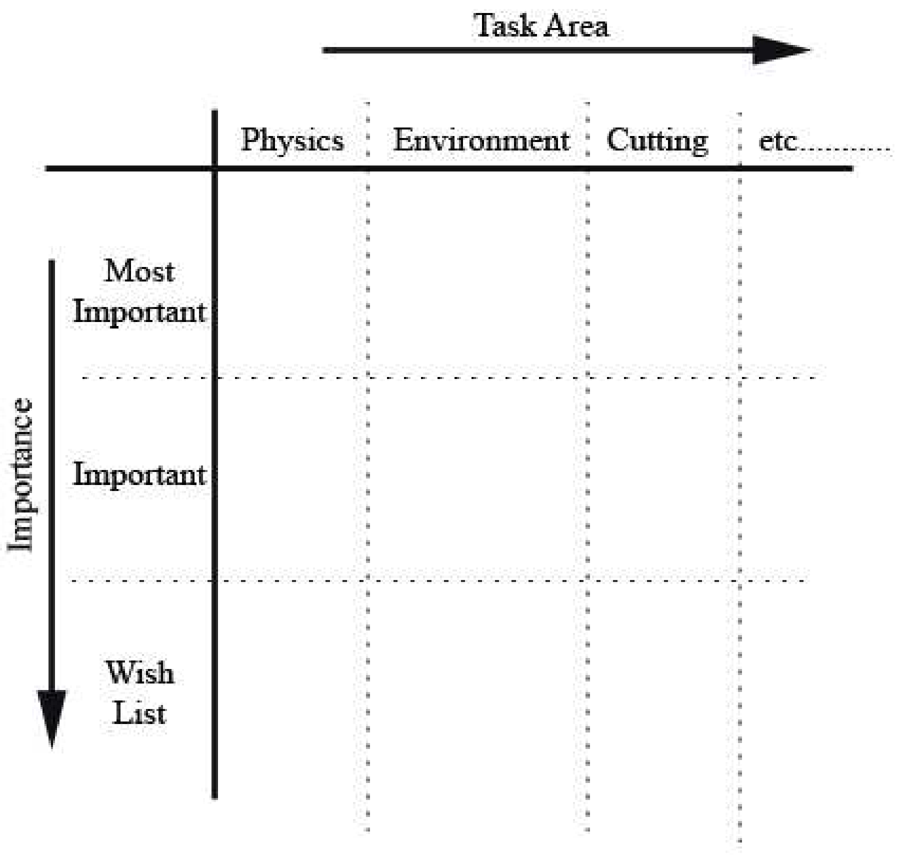
Outlining the production is the first step, from then it is imperative that the team stays on top of the production. There are many methods of project management available to use, some were investigated as follows:

- Microsoft Project 2007 . This was the first option explored as it came highly recommended. It appeared to be a very capable and fluid program (being able to interact with other pieces of software, enabling facilities such as automatically emailing members who were falling behind) and included features such as Gantt charts, milestones and other important elements of project management. However, it seemed vast and upon further investigation appeared to have a learning curve that rendered it useless for such a short project.

- Zoho Projects (projects.zoho.com). A web-based project management tool that was similar in functionality to Microsoft Project, but with a far more intuitive interface and, being web-based, had the added ability to be accessed from anywhere by every member of the team. This system would have been perfect if our timeframe was longer (so it made setting it all up worthwhile) and if the team were dispersed across multiple locations. However, with all of us within reaching distance of each other and a window of 8 weeks in which to operate, it again seemed surplus to requirements.

- Post-it Notes. This system was adopted after observing how the professionals within the game industry operate. Post-it notes would be covering all available wall space in an effort to write down every conceivable task the team had to do. After using it only a couple of days the benefit was already noticeable. The fluidity of production provided by this method was allowing the team to re-prioritise as each member saw fit. The basic principle is that each seperate task would be written on a post-it note, and affixed to a board. This board was laid out as follows:

Figure 3.1: Scheduling



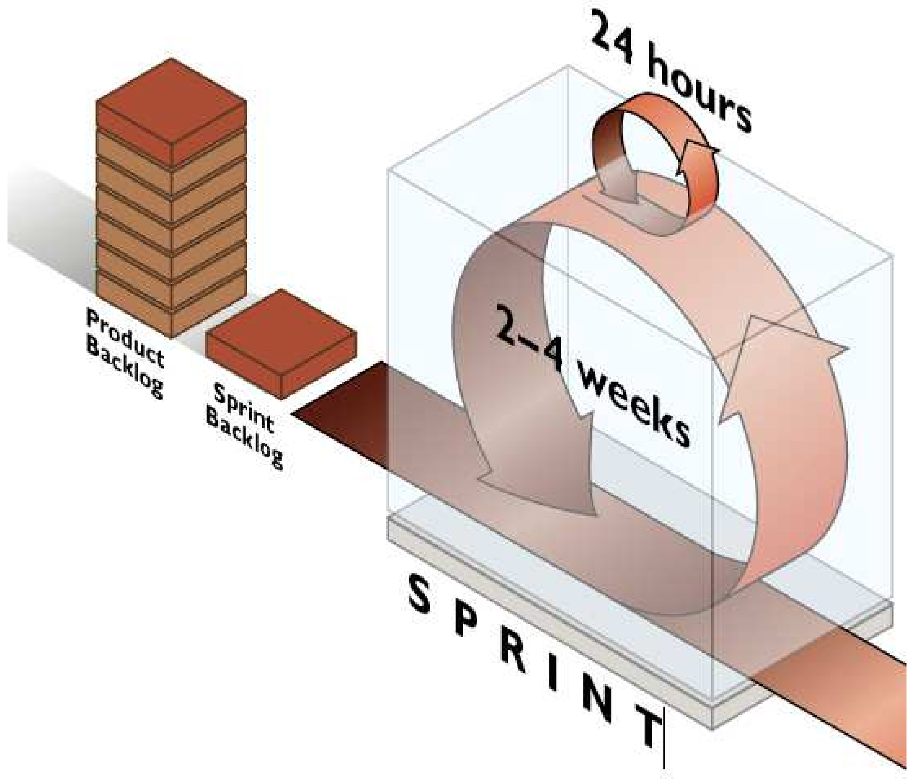
(a) Production Planning Using Post-it Notes

See Appendix A for scheduling and a photo of the post-it note board in action.

3.1.3 Scrum vs Rigid Planning

Maintaining control over the project throughout the duration of the process is just as important as the initial scheduling. Within our team we adopted a method of management known as 'scrum' [3]. This is a vast system, intended to deal with larger projects, so our use was toned down to accommodate our more modest production. The idea is that the production units divide themselves into teams of a handful of people, who meet every morning to discuss (and possibly alter) what needs to be done. It operates by each small team operating under their own command in short periods of time known as 'sprints'. Each team has a Scrum Leader, who attends meetings with other Scrum Leaders. This forms a hierarchy of meetings, and allows for every member to be updated with overall progress with only the leader attending more than one meeting. One key to scrum is that during a 'sprint', their task cannot be changed by outside influence (except of course in exceptional circumstances). The only deviation from their task should come from within the team.

Figure 3.2: Scheduling [3]



(a) Scrum Time Flow Organisation

In the meetings, each member asks themselves three key questions:

- What have you done since yesterday? (accomplishments)
- What are you planning to do by tomorrow? (to be accomplished)
- Do you have any problems preventing you from accomplishing your goal? (risks)

This enables each scrum team to analyse what they've done, plan tasks for the near future and foresee any obstacles that may occur. Each unit within scrum has the ability to complete their tasks with the highest degree of success, due to the fact they operate as a small team.

Our team did not strictly adopt this method, we did however use a few elements from it. The primary element was to acknowledge the importance of daily meetings. Using a '10 o'clock daily' we were able to have an overview of all processes going on that day, and the usage of the three step system enabled every member to remain focused.

The fluidity that scrum provides is definitely beneficial, but it should not be used as a replacement for a scheduled production process. By using an amalgamation of the methods, the team remained focused right until the end.

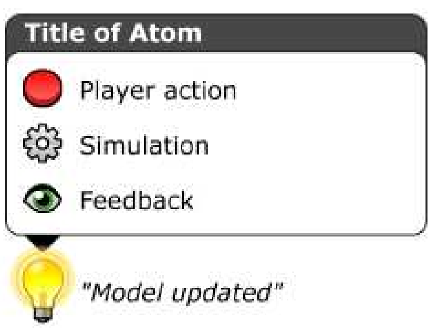
3.2 Level Design

3.2.1 Player Education

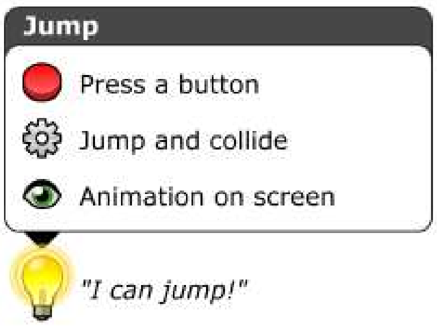
One of the key factors to bear in mind when designing a level (and more specifically the first level of a game) is the inclusion of a system whereby the player is taught various gameplay elements in a certain order. These consist primarily of controls, environment interaction and interface.

The key to a successful tuition interface is progressive learning, something that the Mario series of games accomplished to perfection. Essentially it involves avoiding teaching the player too much at once (such as displaying all controls on the loading screen, a method used in many demo version of games), instead adopting a step-by-step procedure. For instance, in the example below the player has first come to an obstacle and learns how to jump. Then they learn how to jump over a pit, but are not punished for failing. Finally they are made to jump over a pit and will die if theyre unsuccessful. This process is far more intuitive for a player and allows the experience of playing the game to be more enjoyable.

Figure 3.3: Player Learning [2]



(a) Description of Learning Atoms



(b) Learning Outcomes For Each Atom

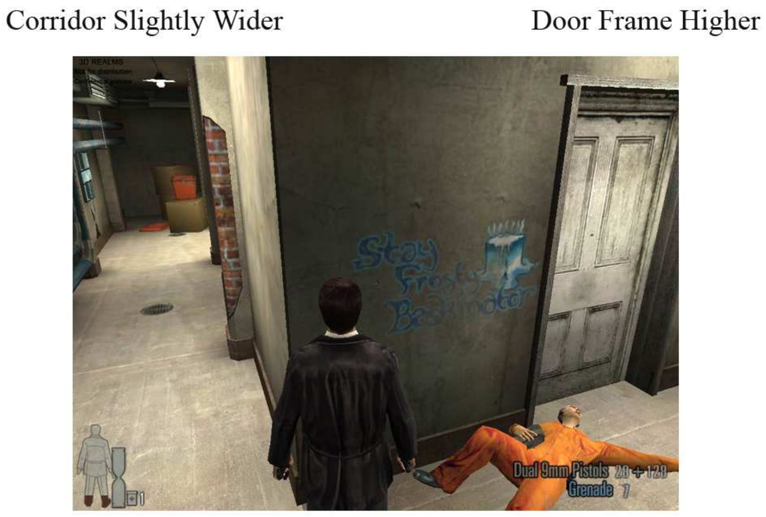
This method was adopted in our game by presenting the player with different aspects of gameplay periodically throughout the first half of the game experience, as outlined below:

Situation	Learning Outcomes
Start of game	Interaction: Movement controls
Environment elements	Interaction: Some objects are cuttable
First asset emitting pulse	Interaction: Pulse = important
Darkness	Interaction: Pulse used for navigation
Guard	Interaction: Combat controls

3.2.2 Scale of Assets / Spatial Awareness

It is important to understand from square one of level design that the scale of the world in relation to the character cannot be equal to that of real life. Creating buildings that have doorways and ceilings to scale will, in the majority of cases, immediately create a feeling of claustrophobia. It is important to bear this in mind when designing the environment, and only trial and error in the beginning stages will get this scale perfect. The screenshot below is from Max Payne (2001 Rockstar Games src). It shows that only a slight upscaling of environment size is needed for a successful effect.

Figure 3.4: Research: Max Payne [4]

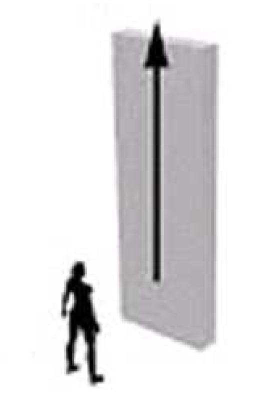


(a) In this example, it is clear to see the scale of the environment is slightly larger than that of the characters

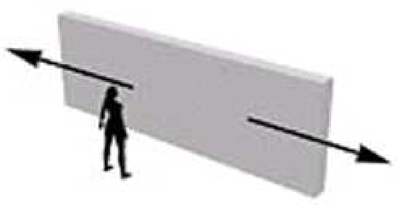
Successfully immersing the player in the digital world is the result of an amalgamation of various elements, from interaction to sound. However, once the player is comfortable within the environment, the level designer can create the assets within the world to invoke a psychological feeling or particular movement upon the player. The entire world can be manipulated to essentially force the player to act as the level designer wishes them to at a certain point.

The texturing and lighting can be altered to create a specific mood, but these will be explored later in the following chapter. The focus for now will be on the actual shape and size of geometry assets in the game.

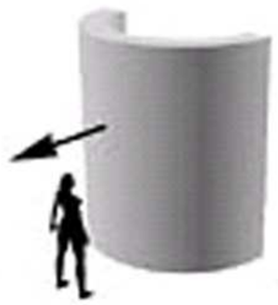
Figure 3.5: Pschological Features of the Environment [6]



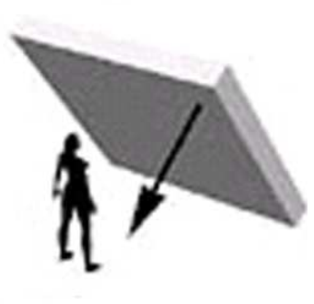
(a) Creates an illusion of grandeur but makes walls appear weak.



(b) Makes walls and objects appear structurally strong.



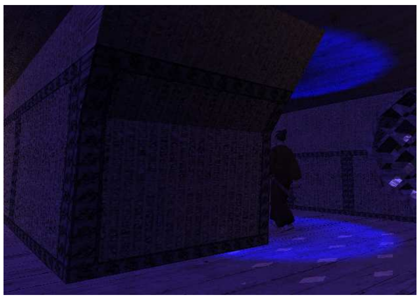
(c) Can make the player more cautious, as well as sometimes making them turn around.

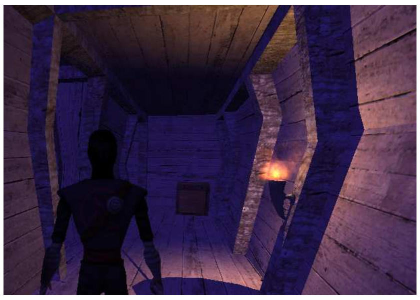


(d) Invokes claustrophobic feeling by decreasing effective floorspace.

These are just few of the examples whereby the assets in the world can be manipulated to invoke various emotive feelings upon the player. These were adopted to some degree within the game, as demonstrated below.

Figure 3.6: Environment Features


(a) Sloping Walls: Various walls within the game are sloped to both invoke a claustrophobic feeling and give the impression the walls are more structurally sound.

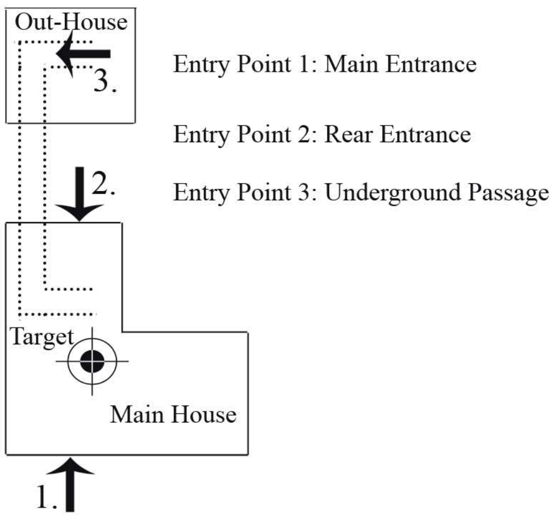

(b) Confined Spaces: The curved walls of the basement area give the illusion of bringing the ceiling closer to the player.

3.2.3 Multiple Routes

One important aspect of gameplay nowadays is giving as much control as possible to the player. A large part of the responsibility of ensuring this occurs falls upon the level designer. Games have for years included multiple routes of gameplay, in both environment and storyline, and the demand for this is becoming heavier in more recent years as gamers expect more from the developers.

With 'The Silence of Night' being a fairly short game, there was no necessity for multiple storylines. However, including multiple 'physical' routes of gameplay within the level was important in terms of longetivity. Below is the basic layout of the core elements of the level, with the 3 entry points defined.

Figure 3.7: Multiple Routes of Entry


Out-House
3.
Entry Point 1: Main Entrance
Entry Point 2: Rear Entrance
2.
Entry Point 3: Underground Passage
Target
Main House
1.

(a) The level was designed to allow for multiple entry points into the target house. Presenting the player with more than one option maintains their interest.

Each entrance has it's advantages and disadvantages, as outlined:

	Advantages	Disadvantages
Entry Point 1	Large doorway, easy to see guards	The guards can also see you easily
Entry Point 2	Guard in kitchen has back turned to rear entrance	No real disadvantage
Entry Point 3	Pickups inside basement	No real disadvantage

3.2.4 Rewarding Experience

Maintaining a player's interest in a game is the next challenge the designers come up against. It is vital that the player's desire to continue playing the game is not quashed too early in the game. This can be caused by such things as the inclusion of a difficult first level, an unintuitive control system or interface or an unrewarding experience.

Difficulty settings are mainly created through trial and error with the tweaking of settings, and the control system is a result of feedback coupled with good ergonomics. However, rewarding the player from the outset within the game is not only a superb method of maintaining interest, but also a very simple one to implement.

Games have used this method for years in order to engage more people in a shorter space of time. A good genre for emphasising this point is that of racing. In games such as Gran Turismo (S.C.E.E. 1997), the first race a player encounters is always going to be simple. But by adapting the AI to make the speed and handling of the competitor vehicles remain around the player's abilities, it becomes almost impossible to lose. Therefore, within minutes of picking up the game, the player is presented with an award, usually in this case a shiny new vehicle to use in the next race.

This methodology was adopted within our game in a few different flavours.

Firstly, the player begins with a low-damage weapon. Dotted around the environment are an assortment of more powerful weapons, along with actual weapon amplifier power-ups. These enable the player to gain stronger in their attacks from an early stage. It would have even been possible to place these items in without the functionality being there, and most players wouldn't notice the lack of difference, instead enjoying the 'placebo' effect of more powerful weapons.

Secondly, there are pick-ups located in secretive locations around the level. These come in a few incarnations: (Weapon Amp is listed again as the following list is exhaustive).

Pick Up	Description
Weapon Amplifier	Amplifies damage that current weapon inflicts upon guards
Health Pack	Adds 25% of total health to the player's health status
Artefact	Object of value that the player can 'steal'

With the game having a strong focus on stealth, the techniques used by the player to infiltrate the house have an effect on the outcome. Scoring is based on the way players 'deal' with the guards, with the following details being recorded by the scoring mechanism:

- If the player completely avoids detection by a guard
- If the player kills the guard
- If they do, was the player's presence acknowledged by the guard prior to the killing?

- Did the player not kill a guard, but the guard still saw them
- Or did the guard only hear the player?

Lastly, at the end of the game, the tallies are collated and displayed to the player. The player is then presented with a score, made up of a combination of these results. A stealthy mission, whereby no guard was alerted, will get you the most points, as well as a suitable reward. Likewise, you will receive an award if every guard was killed and every guard noticed you.

Rewarding the player in this fashion, be it either positively or negatively, is something that has been very successful, most notably in the game series 'Worms'. Awards such as 'Biggest Coward' or 'Most Useless' can be just as entertaining to receive as 'Most Dangerous' or 'Best Player'.

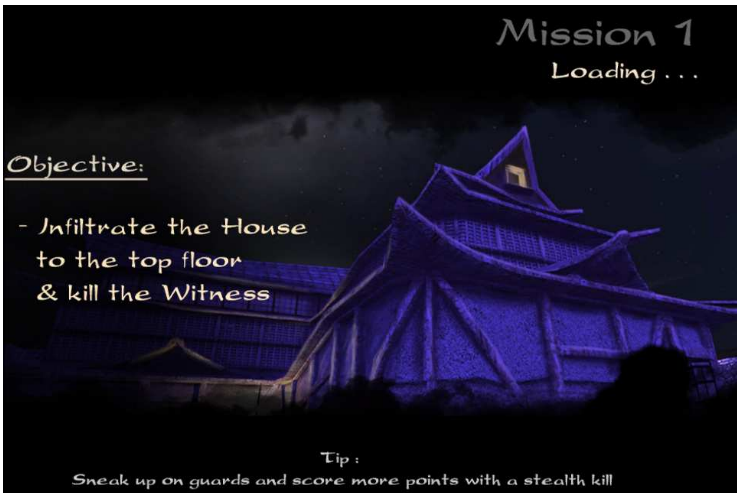
The award process is described in detail in the design document (Chapter 9 - Appendices).

3.2.5 Objectives

The importance of giving a clear cut objective to the player in terms what they have to do cannot be underestimated. There are of course some games that do not always display this information, instead allowing the player to seek out an objective and then follow it up (such as the 'Grand Theft Auto' series of games). However, this process still ends up with the player being presented with an objective.

A game is essentially an interactive story, and therefore must have a path down which the player can traverse. With 'The Silence of Night' being a single mission prototype, the player is presented with the objective during the loading screen. This displays not only a text-based objective, but also a visual clue as to the physical location of the target.

Figure 3.8: The Objective



(a) The loading screen is a good place to have the objective as it distracts the player from the loading time.

By using this method, it not only gives the player a clear objective from the moment they pick up the controller, but it also takes the attention away from the time the game takes to load. This method is used by many games, and proves very successful.

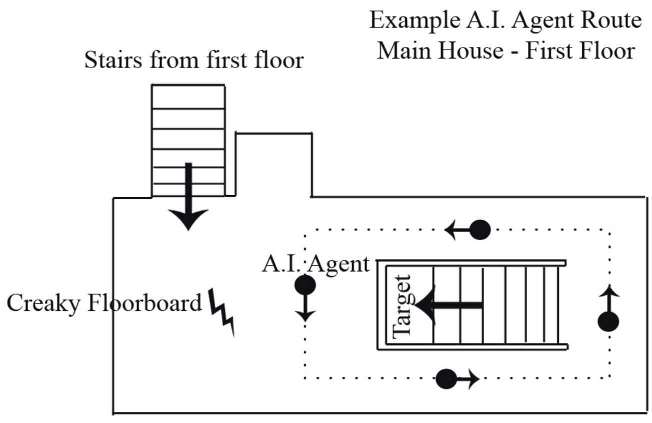
The original intention was to have a strong narrative within the game, with a Japanese language voice-over being played while the English subtitles were displayed on the loading screen. This narrative, however proved to be far more time consuming to implement than thought, so therefore was omitted from the production at an early stage in the process.

3.2.6 AI Agent Routes

Originally, the AI was planned to simply engage the player when they got too close, but the system took on a far more complex design and therefore allowed for a more comprehensive implementation within the game. The level design was semi-symbiotic with the other strands of production, none more so that the AI. Elements of the world were altered during the course of production to allow the new features of the AI to be demonstrated.

There were three main types of routes used by the AI in the game, static, circular and oscillating. A good example of the implementation of more than one AI feature within the environment is the first floor of the main house.

Figure 3.9: A.I. Agent Routes



(a) The level design was developed in coordination with other areas of production; here the first floor was adjusted to account for the more intelligent A.I. system.

In this diagram, the player enters the floor from the stairs in the top left. The route of gameplay is up the second set of stairs, the entrance to which is on the far right of the diagram. An AI agent is on a circular route, patrolling around the central column in which the stairs lie.

With the AI having the ability to acknowledge audio, a creaking floorboard was placed in the direct path between the player's entrance and their target, with the intention of creating a noise that the guard would pick up on.

By placing guards on defined routes between the player and their target, the gameplay is altered as the player has to use stealth (the game's focus point) to avoid detection. Other guards within the level are on either oscillating or static routes, also sometimes placed in areas of strategic importance.

3.3 Environment

3.3.1 Inspirations

The first step to creating an interesting environment for a video game is to analyse the styles that can be used for the game in question. It is up to the creative team to come up with the visual styles, but in such a small team it is important that the styles were given the OK by all team members.

The initial stage was to research styles in existing media that could be adopted, either in their entirety, an adaptation or by simply using a certain style or method. After researching games, film, television and 2D artwork, one game came up again and again with an artistic style that the environment designer felt could be successfully transferred to our game.

Fable 2 (Lionhead Studios) is due for release in 2008, but screenshots and artwork have been released in order whet the appetite of gamers. The setting has a medieval styling, which is akin to the setting of 'The Silence of Night'. By amalgamating the styling of Fable 2 with an oriental feel, the environment geometries take on their own styling.

Figure 3.10: Fable 2 Screenshots

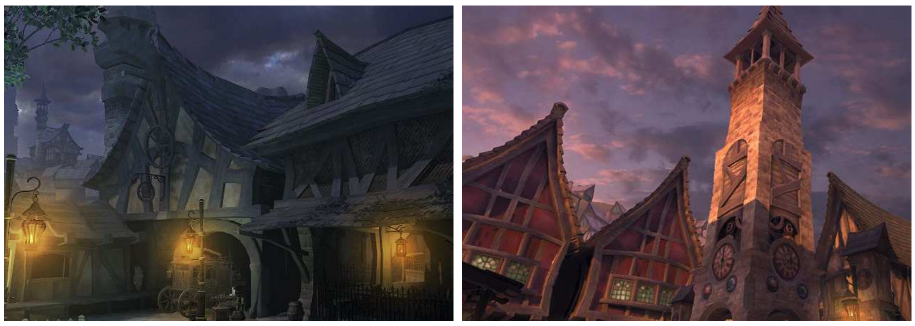
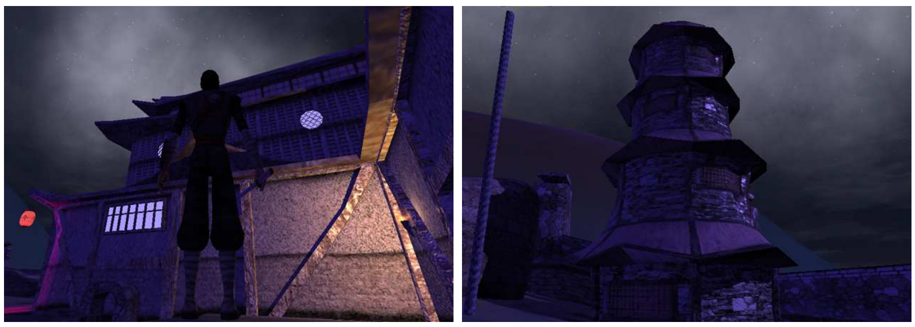

(a) Fable 2 night scene (b) Note the absence of any straight edges

Figure 3.11: In Game Screenshots

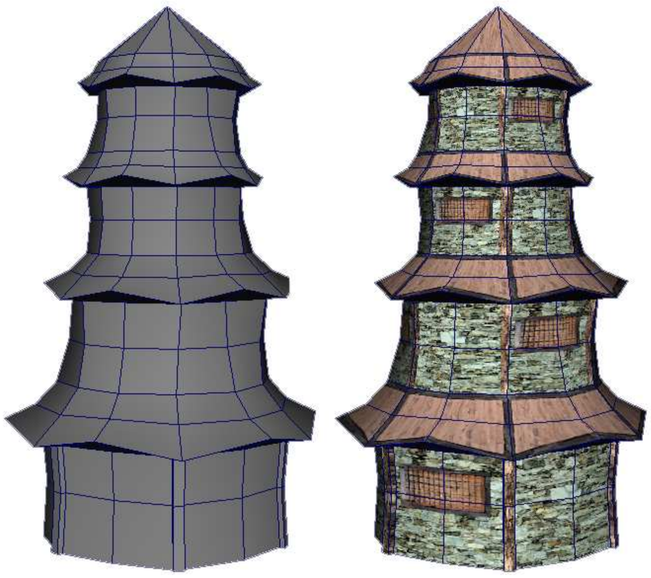

(a) Colours mixing in the night scene of the game. (b) Straight edges were avoided at every opportunity.

3.3.2 Polygonal Modelling

Despite the hardware we were supplied being top of the range with nVidia GeForce 7600 graphics, it was still important to maintain optimisation as a priority during production, as we did not have enough of a timeframe to optimise as a final stage. One of the key steps in increasing the frame rate of a real-time 3D simulation is the lowering of the polygon count.

To keep the poly count low the assets of the world were designed with minimum high detail areas (some areas required high detail and so were optimised using level of detail - see section 3.3.7).

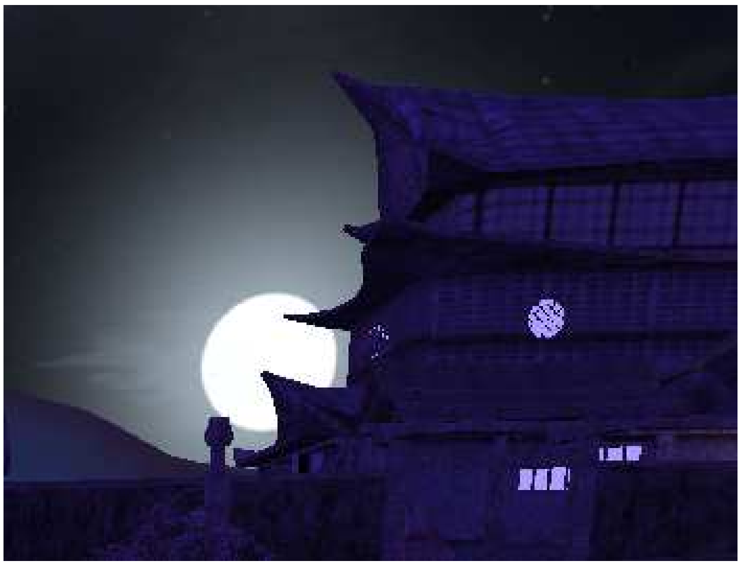
Figure 3.12: Low Polygon Models



(a) The models were as low on polygon count as possible in order to maintain the level of optimisation required.

The example above shows an asset within the game that, apart from the peak of the roof, contains no noticeable straight edges. This was a key concept that was adhered to for the majority of the external geometries. It was important for the artistic styling of the piece to ensure that the look of all the world assets maintained the intended styling even when viewed as a silhouette. This is an old artistic trick that is just as applicable in 3D creation as it is in conventional art.

Figure 3.13: Silhouette Styling

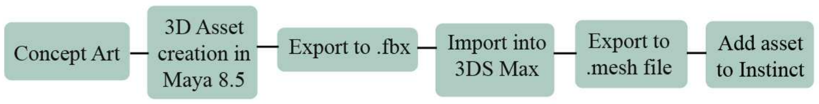


(a) With the styling being clearly visible as a silhouette, the textures and shading have little work to do in order to enhance the effect.

3.3.3 Geometry Pipeline

The geometry assets started out life as a 2D sketch, and were translated into 3D within Maya 8.5. However, the exporter supplied with Instinct only worked from within 3D Studio Max. This meant the pipeline was somewhat cumbersome:

Figure 3.14: Geometry Pipeline



(a) The pipeline for the geometry creation was not as streamlined as it could have been.

I conducted some research into the possibility of creating a .mesh exporter for Maya. Originally it seemed that this task would not be too daunting, as the environment required only static geometries, omitting the need for the exporting of bones, animations or skin weights.

The intention was to either create a MEL plugin that could export directly from the program, or create a small utility that parsed an exported .obj file. Seeing as the .obj parser would skip one step in the pipeline, whereas a MEL plugin would skip two, I opted to focus on a creating a small tool to add to the shelf within Maya.

After research into the complex file structure of a .mesh file, it was clear that the file type was optimised for grpahics. Therefore I concluded that the time it would take to create a tool capable of successfully exporting both the geometry and the UV's into such an awkward file system was not worth the sacrifice of time for such a small project. It turned out the supplied exporter for 3DS Max wasn't always successful at creating a functional file, which only reinforced my thoughts about the file type's complexities.

In hindsight, I feel that the rapid veto of the plugin was a mistake, as it was rare that each geometry asset was 100% correct on the first attempt at insertion into the engine. This resulted in literally hundreds of meshes going through the pipeline (the final tally for the number of meshes in the scene stood at around 170) and as such numerous hours were spent in the process of doing so.

3.3.4 Texturing and Stylisation

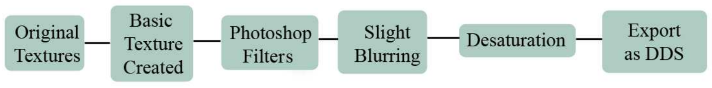
With the stylisation of the geometries already decided, the next step in creating the desired artistic style is the manipulating of the textures in order to achieve an illustrative look. Textures used were photographs, so these had to be significantly altered in order to create the target aesthetic. The process of taking the texture through from original source to final .dds was the result of weeks of trial and error with the aesthetic. The use of photographs as textures within the game was resulting in a horrendously unmatched feel, and therefore experiments were carried out to amend this effect.

The first step was to use a handful of Photoshop filters to instantly stylise the textures. The result was a custom Photoshop macro that encompassed elements of the artistic filters 'Poster Edges' and 'Watercolour', then slightly blurring the result. This took the edge off the realism of the photographs, while at the same time remaining slightly more realistic than cartoony. Feedback from various sources confirmed that this was a pleasing effect.

When the game scene reached a point of near completion in the final few weeks, the lighting was at a state that was almost the finished article. The vivid colours of the textures were fighting against the blues of the moonlight and the oranges of the fire torches creating an effect that was not only distracting but difficult on the eye. This had to be changed, and research suggested that removing some of the colour from the textures would resolve the issue. Tests were carried out on some textures by desaturating them to about 60-70% of their original intensities. It worked well, so another Photoshop automated script was created to go through the folder and apply a preset desaturation macro to every diffuse texture.

The resulting texture pipeline is shown below:

Figure 3.15: Texture Pipeline

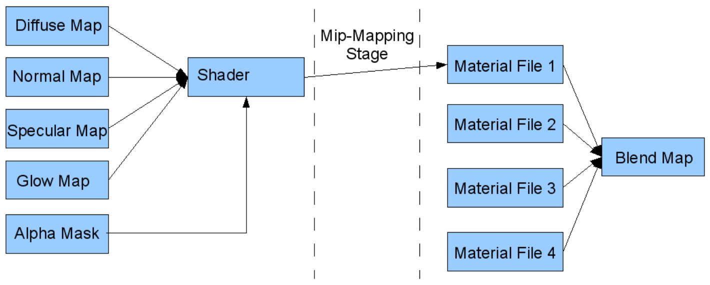


(a) The flow of textures, from the source files to the final output.

3.3.5 Materials

The Instinct engine supports a variety of graphics technologies, including the ability to use a variety of maps for the materials. As well as the standard diffuse, it also supports normal maps, specular maps, alpha mask maps, glow maps, mip-maps and blend maps. These can all be combined to create a single material, resulting in a very impressive effect. Although unused within the game, it is technically plausible to use every type of map in one complex mapping process, as outlined below:

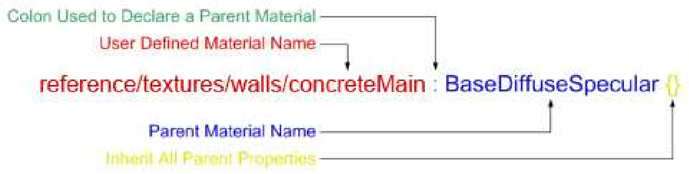
Figure 3.16: Example Material Usage



(a) A fictional scenario whereby every type of texture map is used.

The materials within the engine are created once, and can therefore be used on multiple surfaces without draining too many resources. A material file is created by first specifying the name and type of material, and then describing what types of files are used to create it.

Figure 3.17: Material File Description [6]


Colon Used to Declare a Parent Material
User Defined Material Name
reference/textures/walls/concreteMain : BaseDiffuseSpecular {}
Parent Material Name
Inherit All Parent Properties

(a) Description of the first stage of defining a material file, taken from the Instinct Studio documentation.

The textures used to create the material are defined within the main body of the material code. The example below is a full material template for the red lanterns in the game. Note the difference between the first line in the example below compared to the one in the above diagram. The following example has an alpha pass.

Listing 3 Material Example

```
MO_Package/materials/lanternRed : BaseDiffuseSpecularAlphaTest
{
  flags
  {
  sortType = litAlphaTestGlow
  }
    {
      diffuse = MO_Package/textures/lanternRed
      normal = MO_Package/textures/lanternRedNRM
      specular = MO_Package/textures/lanternRedSPEC
      map = MO_Package/textures/lanternRedFX
    }
  Pass SFX : BaseTextureColor {}
}
```

The 'flags' section of the code contains pieces of information that the engine requires to know before the material is created. In this case 'sortType = litAlphaTestGlow' defines the sequence in which the material needs to process the textures (or maps) in order to get the desired result. 'lit' is the name of the default material properties (diffuse, normal and specular). This is followed by alpha and finally the glow map.

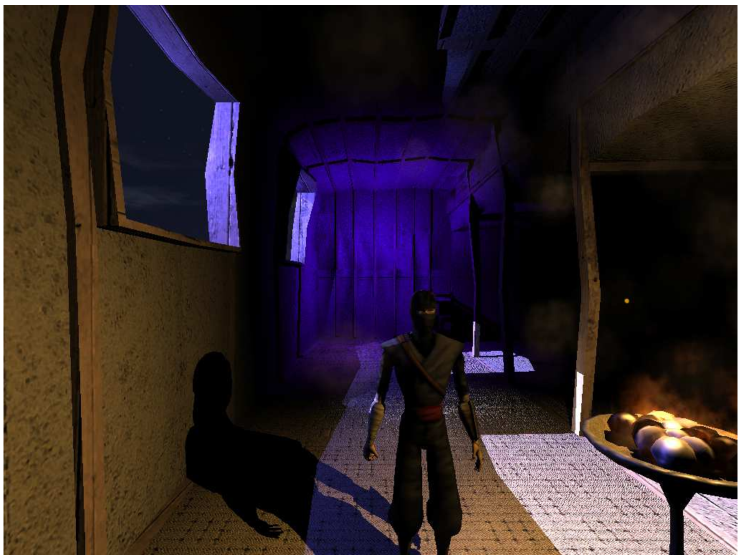
Within 'textureAliases' lies the paths of the various texture files required for the material. Aside from the usual three, the above example includes 'map = MO_Package/textures/lanternRedFX'. 'map' refers to a texture that is used as an FX map, in this case a glow map. It defines which areas of the lantern need to glow in a post-process.

3.3.6 Lighting and Atmosphere

Creating an atmosphere in any medium is the process of invoking a particular mood or feeling upon the viewer / player. One of the most influential methods of achieving this is to manipulate the lighting. Harsh shadows and dark colour give a completely different feeling to soft shadows and subtle tints of colour within the light.

The mix of blue moonlight and orange flaming torches worked well in tests and so was used throughout the game (see figure 3.3.1), as well as within the menu scene file.

Figure 3.18: Lighting Mixtures in the Menu



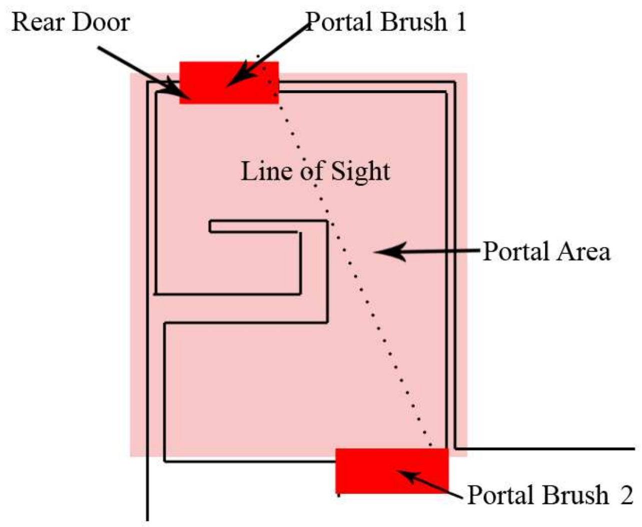
(a) A mix of blues and oranges work well in the menu scene file.

3.3.7 Visibility Management

One method of optimising a scene is to insert portals and break up all the geometries into areas. This then uses a method of visibility culling that kills every asset within a certain area if that particular area leaves the viewing angle of the character. By default, Instinct supports visibility culling, backface culling and lighting culling (if the character cannot see any area of a light's shadow volume then the light is turned off). However adopting portals breaks up the world into more manageable areas and completely deactivates all assets within these areas.

Below is an example of how portals were initially used within the Instinct engine.

Figure 3.19: Example of Portals



(a) Portals, although not implemented within the game, would have allowed for a cleaner culling of thw world assets.

However, as Instinct was still technically a beta release, the engine developers were unfortunately unable to get portals functioning successfully in time. This was not much of a drawback for the team, given that the world we had designed was rather compact. There was still a need for optimisation though, so the next step was to add L.O.D. to the more complex geometries.

Level of detail (or L.O.D.) is natively supported by Instinct, and so was implemented to a great degree, and in varying strengths. Complex geometries that could still be viewed from a distance (such as the windows, trees, rockeries etc) were given a level of detail that enabled the player to acknowledge no change in physical appearance when the change occurred. This is was to ensure fluidity in gameplay as well as for aesthetic reasons. However, complex geometries that were only visible close up (such as the basement support structures, the victim, the sandbag ramp etc) were given a more drastic level of detailing. In the case of the basement supports, the geometry was created as one large entity of around 1100 polygons. The second level of detail was a 4 faced shape that covered the same area as the supports. This change is drastic but enabled the use of more effects elsewhere in the scene.

3.4 Sound

3.4.1 Ambience

The second important area to consider when creating atmosphere is the audio. This was considered in great length, with the result being a score made up of three different looping audio tracks:

- Track 1 - Ambient: To be played through the entire game at a constant volume.
- Track 2 - Tension: Will kick in when the A.I. Agents acknowledge presence of player.
- Track 3 - Fight: This is played when the player engages in combat.

All three tracks are exactly 45 seconds long and are designed to be played over the top of each other. This enables the music manager to simply alter the volumes of the second two tracks to account for the current situation of the player.

Given the lack of serous knowledge of orchestrating a score, the three tracks were outsourced.

3.4.2 Subtle Touches

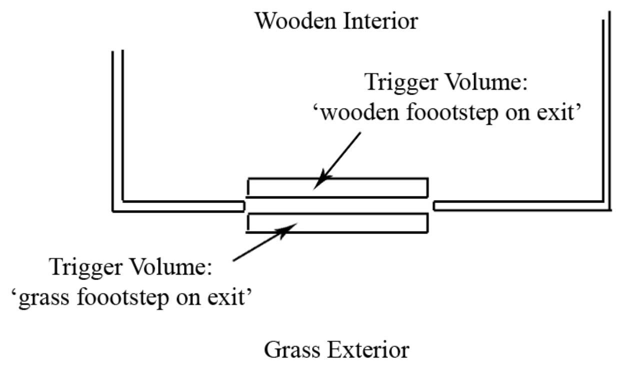
To give the player full immersion within the world, it was important to give as much audio input as possible. This included the creation of various types of footsteps for use on gravel, wooden floors and water. Environmental sounds are also important, and these were implemented in the form of trickling water, fires crackling and the occasional animal noise. These all combined to create a greater feeling of depth to the environment.

3.4.3 Triggers

In order to fully manipulate the audio within the environment, triggers were used that controlled what sounds were used for various functions. These trigger volumes surrounded areas that required a change in footstep sound, such as in the doorway, as illustrated below:

This enables the footsteps of the character to be altered based upon the player's physical location within the scene.

Figure 3.20: Trigger Volumes



(a) Careful placement of the trigger volumes enabled the player to interact with the environment in a more immersive way.

Chapter 4

Interactive Cutting

By Ali Derweesh

4.1 Introduction

The aim was to try to create a real-time geometry splitting system suitable for use in games. Games are continually advancing in realism and sophistication, and there is a continual search for new game ideas and mechanics. While geometry cutting systems exist in real-time applications, these are mainly surgery simulations; games would have a very different set of requirements.

In a surgery simulation what is important is accuracy of small cuts, usually in soft-bodies. Performance is not critical. Some games may have similar requirements, but it is more common to need to perform large cuts on rigid bodies. Either way, performance is far more important. This system is intended to quickly deal with large cuts, typically a single cut would represent the path of a moving blade over one game action.

In the game the system would be called on to cut both simple and complex models and integrate with the physics engine. It would also give a better idea of true performance in use.

An OpenGL visualisation was used with the actual cutting system for development and demonstration purposes. The system is simple, possessing only basic controls and no physics or texturing. However, polygon colour is set using vertex UV coordinates, and normals are viewable.

4.2 Design

The most important design consideration was performance. In surgical simulations typically only a few triangles at most are cut per frame, and frame-rate stutters occurring if longer cuts are performed can be forgiven. Schemes relying on the position of the cutting object can be used that avoid the need to check all triangles [16]. In a game, an entire model can be cut across in a single operation, and any triangle may be cut. Slowdown is also less forgivable.

Additionally, cutting soft bodies is actually simpler in some ways than cutting rigid bodies, as physics simulations are already being applied per vertex. This means that simply changing the vertices and edges of the geometry is sufficient to separate the resultant pieces of the object, it does not have to be divided into new objects. However, when a rigid body is cut extra processes must be performed to separate the new pieces created.

While initially the system was developed to use a semi-circle as the cutting plane, during development the decision was made to use an infinite plane to perform the actual cut. This was much simpler to implement, an important consideration as the timescale of the project was limited. Resolving cut triangles and triangulating new surfaces especially would have required much longer to implement if a finite cutting plane had been used. It was decided that using an infinite cutting plane would also work better for gameplay reasons. Working out whether an object should be cut could be done externally, then an object is simply dissected. This makes it simpler for the player to cut through anything than needs to be cut. In the game we made, slicing things cleanly in half proved to be very satisfying. Furthermore, using an infinite plane made some operations faster, thus improving performance.

4.3 Model Requirements

Using the native geometry data format of the game engine created some limitations on the current system. Presently the system only supports one triangle mesh per object. The game engine uses one material per mesh: a material uses a texture map, a specular map and a normal map. This means that the final system only works with one mesh and one material per cuttable object. A portion of the main texture is therefore set aside for use in the new surfaces created during a cut.

Since the game engine relies on objects being solid, rather than flat sheets, the current cutting system was also designed with this assumption in mind. However it would not be hard to modify in order to accept non-solid meshes.

It is even possible to automatically work out whether the object is solid or has holes, based on whether there are any edges that only connect to one triangle.

Most of these issues can be overcome through further development or the use of a more appropriate data structure. Some additional data is needed for special cases, for example objects that are attached to the environment need to have a vertex specified as a fixed point. This is handled in-game.

4.4 Data Structures

The data format in use by the engine had the correct structure and much of the necessary data. While the parent Mesh class stored several mesh surfaces, the main data structures consist of:

- Mesh Surface - Arrays of triangles, mesh vertices and edges
- Triangle - Contains a face normal, the indices of three vertices and three edges
- Mesh vertex - Contains vertex coordinate, normal and UV
- Edge - Contains two indices to triangles and two vertex coordinates or indices to two mesh vertices
- Plane - The cutting plane. The basic data stored is a point on the plane and a normal to the plane

The game engine also stored vertex tangents and binormals, UVs were stored outside of mesh vertices but corresponded to them and so did not need separate indices. Triangles did not store face normals or k values, and edges did not store indices to the original triangles, so these had to be stored separately.

More advanced cutting shapes would inherit the plane and store extra data. For example a circle would store the radius. A semi-circle can store an additional vector to define which half of the circle cuts. Using two additional vectors or a vector and an angle allows the use of a sector of variable size.

Storing edges allows some optimisations as an edge is shared by two triangles, halving the time needed in some steps, and also allows the entire mesh to be connected. This is important for splitting the object into multiple new objects.

One problem with storing excess data to that of the engine defaults was that it had to be stored on a per-object basis rather than a per-mesh basis. As all of this data was calculated, it could have been calculated at the time of a cut rather than stored. However to improve cut performance it was decided to calculate the data at load time and store it. This could easily be changed depending on the requirements of a system.

4.5 Program Flow

The first step is to find the intersections of the triangle mesh and the cutting plane. After each individual triangle is checked, the results are compared to the set of possible scenarios. Depending on the scenario, new edges and vertices are created and the old triangle is replaced by new ones. During this stage the edges along the cut surface are saved for use in the new surface triangulation phase. After this has been done to the entire mesh, a sorting step takes place in which the triangles are sorted into connected meshes representing the pieces of the object created by the cut. Finally the new surfaces can be triangulated. This takes place on a per-object basis. The correct edges along the cut from those saved earlier are copied, sorted, and then organised into loops, each loop represents a polygonal new surface that needs to be triangulated. New vertices with appropriate UVs and normals are created for the new surface. The polygon is subdivided into smaller polygons in successive steps until triangles are found.

4.5.1 Intersection Testing

With an infinite plane only simple plane-edge intersections can occur. When using finite planes, the shape dictates the specifics of the intersections calculations, but there are essentially two types of intersections: those between the edges of the triangle and the cutting plane, and those between the edges of the cutting shape and the triangle plane. For example, if the cutting shape is a polygon then the two calculations are almost identical. A circle would require different calculations for finding the intersection of the circle perimeter and the triangle plane. Some further checks may be necessary when the cutting plane is parallel to a triangle, such as line-line intersection calculations. Using inheritance, we can program different cutting shapes and use whichever is appropriate.

For the purposes of this project, initially a semicircle was used. This was stored as a plane, a radius and a directional vector representing the acceptable half of the circle. While all the intersection calculations were fully programmed, the decision was made to use an infinite plane and the semicircle was dropped. Using an infinite plane completely removed the need to use any calculations when the plane is parallel to the triangle. Only plane-edge calculations were required.

One optimisation that was done at this step was to store the results of checks on triangle edges. Since each edge is shared by two triangles, this halves the number of calculations required during testing. Fortunately the data structures used by the game engine supported this optimisation.

It is also necessary to initially transform the cutting plane in order to move it into the object coordinate system. This is more complicated when a finite cutting shape is used and non-uniform scaling may occur. However the engine did not support scaling and an infinite plan was being used, so it was not an issue for this project. Simply transforming the plane origin and normal into the object coordinate system was sufficient.

4.5.2 Triangle Resolution

For an infinite plane, there are only 5 possibilities, of which only 4 need to be resolved. For a finite plane, the situation is more complex but it comes down to 11 main cases that need to be resolved (see tables 4.1 and 4.2). Each case can be resolved separately, with new triangles, vertices and edges being saved to a list.

When the cutting plane is exactly along a triangle, it is a slightly different case that must be treated in a different way. In fact if the cutting plane is finite there are many different ways in which the plane can cut across the triangle. Fortunately each scenario does not have to be considered separately (though it may be more efficient to do so). If the cutting plane is infinite there is only one scenario and it is easy to solve. When this occurs the triangle face normal is used to determine which side of the cut the face is on, so if a mesh has an inward-facing triangle it will react incorrectly to a cut along that triangle.

It is during this stage that the separation of parts of the mesh above and below the cut first begins. This is done by duplicating edges along the cut. Storage conventions are used to separate those above and below the cut. Triangles below the cut always save references to the first edge, and triangles above always save references to the second edge (therefore the first edge is considered below the cut and the second is considered above the cut). This is important as connectivity is found using these references - if a triangle references the wrong edge the object will not separate after the cut. The actual separation takes place during a later sorting step.

When dealing with soft bodies and a finite cutting plane, certain extra considerations may need to be made. Because the vertices move independently, depending on how the system is being integrated, it may be necessary to have two copies of every vertex on the cutting plane. This is because an object can be cut without being split into new parts with a finite plane. With a rigid body a vertex on the plane does not move separately, so only one vertex is needed, that is referenced by both sides. In contrast, in a soft body the coordinates and normals will need to change separately as the points are pulled apart.

The problem with this is that if the edge of the plane intersects the edge of a polygon, it should have only one vertex. This means that cases that can be combined for a single solution with a rigid body or an infinite plane must be dealt with differently in order to have only one point (see table 4.3).

The points must be separate for a soft body as the coordinates will change. However, if the cutting plane's edge intersects the triangle edge it must be treated as a separate case. If we treat it as the same case, then we get incorrect results as the adjacent triangle should have only one vertex. Therefore some extra considerations are needed for soft bodies with finite planes.

4.5.3 New Object Analysis

After all the triangles have been analysed and the new triangles created, the new parts can be separated. Since each triangle stores references to edges and each edge stores references to triangles, the connections can be followed like a tree, with each edge, triangle and vertex found being added to lists of components for one resultant part. The fact that the edges along the cut were duplicated for the triangles above and below the cut means that no triangles below the cut are connected to any above, so the components of only one part will be added to a list. This is repeated until all the triangles have been placed into a number of lists.

Since the arrays of vertices and edges are changed during this process, the indices stored in the triangles and edges have to be changed. This is done during the sorting process. A vector of new indices is created for vertices, and another for edges. A flag value signals that the vertex or edge has not been added to the new lists, any other value represents the position of the vertex or edge in the new list, i.e. the new index. As the connections are being followed through the triangles and edges, every vertex or edge encountered is added to the new lists if the flag value is found. Otherwise the new index overwrites the old one.

The edge data structure did not store the connections to the triangles, so this information had to be stored separately. The connectivity is easily found by running through the array of triangles and adding the index of each triangle to each edge that it references.

4.5.4 New Surface Triangulation

After the resulting parts have been found, the new surfaces can be triangulated. A simplified Delaunay triangulator [18] is used. The current system is imperfect, and further work is required, however it successfully fills most of the hole most of the time.

Edges along the cutting plane are copied to a list, then they can be sorted into connected "rings". New vertices must be created, with appropriate normals and UV coordinates. In order to find the UV coordinates, first the maximum and minimum values of the coordinates on the cut surface are found. This is used to find a scaling factor. Using the smallest coordinate values as the origin point, each vertex coordinate is converted to a 2D coordinate along the plane. This is scaled by the scaling factor into the desired range. Currently the range is 0 to 0.03, as this region of the bottom left corner of the object texture was set aside for the internal texture.

Each ring is triangulated separately. If the ring has three edges it is already a triangle and is saved. Otherwise an edge is created from the first vertex in the ring to the third (see table 4.4). This edge is tested in two ways. First the angles of the new edge with the first and last edges are tested to see if it is inside or outside the ring. Then it is tested for intersections with all the other edges. If it fails either test the edge is discarded and a new edge is built from the first vertex to the next in the loop. If the edge passes, it is added to the list of edges. The process is repeated recursively on both new rings. If no acceptable edge is found the process is repeated from the next vertex in the loop, as sometimes an edge cannot be found from one vertex but can from another. A directional convention is used to ensure that the edges are facing the correct direction; therefore the resulting triangles are created facing in the correct direction.

A countdown is used to avoid an infinite loop. While theoretically in a perfect system an infinite loop should not occur as it should be possible to triangulate any polygon with these steps, a bug in this system meant a countdown was necessary. The system uses a single list of edges, and pointers to the first and last edge of a ring, for efficiency. Currently there are some situations that are not handled correctly, such as concentric rings.

Each ring is a polygon that needs to be triangulated to form a new surface. The triangulator tries to create new edges between vertices.This edge goes on the wrong side of the first edge, and therefore is rejected.This edge intersects another edge and therefore is rejected.When an edge is found to be acceptable, the polygon is subdivided into two new polygons and the function is called recursively on each one. This edge is added to the objects list of edges.In fact a new edge is added twice to the list that represents the polygon as it is part of both polygons. This is a separate list from the object edge list. While only one list is used, iterators mark the first and last edge of each polygon.When a triangle is found it can be added to the object triangle list.In this polygon, the original first point cannot form an edge to any other points. Therefore the list is rotated but removing the first edge and adding it at the end of the segment of the list. There is now a new first vertex, and the function is called recursively again.Eventually the entire polygon is triangulated.

The main complication that can arise is if two rings are in fact inside each other (see table 4.5). This would be a very useful situation to be able to handle, as it would allow hollow objects such as boxes or hollow bamboo to be cut.Unfortunately the obvious way of checking whether any rings are concentric is very inefficient as it involves checking a vertex from one each ring against all the edges of all other rings. Once concentricity is found it could be solved by connecting two vertices from the two rings.In essence this would work like a single polygon of unusual shape, and could be solved in the normal way.However this would not be enough, as it is possible to have many concentric rings. The possibility of such situations makes analysis of concentric rings difficult and slow, which is why it was not considered in this project. Any concentric rings that arise would simple each be triangulated as a simple polygon.

A finite cutting plane, however, introduces a new complication (see table 4.6). It is possible to have incomplete loops. This is a situation that is not considered by typical Delaunay triangulators.This means that the program has to join them together to create full loops. In theory this is possible by joining the end points together to create a loop.However there is no guarantee that there will be exactly two loops, there may even be an odd number of loops.Using the face normal of the triangle would be necessary to resolve this issue, but even then the solution is complex. Since it is possible for an object to be cut but not split, a triangulator for a finite cutting plane would need to be re-written to triangulate the same face twice for one object.

4.6 Game Integration

It was necessary to build the system around the native geometry data formats used in the game engine. The engine allowed user made entity components, two of which were used to integrate the cutting system. One was added to any item that could be cut; another represented the blade and was added to the player controlled character. The first referenced the geometry mesh and stored all the additional data needed to perform a cut. The second was basically a wrapper for the cutting plane. Additional data was stored in both classes for gameplay reasons.

The game engine was not designed to allow geometry to be modified, therefore it proved necessary to save the new meshes to file and load them again. This caused performance issues, as the final game would pause for a brief moment (on the system we were using it was approximately a split second) when a complex mesh was cut. Interestingly, in an OpenGL development environment the system would hesitate for a significantly longer period of time, suggesting that the game engines memory management system was catching the file and reading back from memory, speeding up the process.

The in-game mesh wrapper class took care of loading the files and creating new objects. The type of new object could be changed, allowing objects with different physics or game properties to be created. For example, when cutting a shoot of bamboo, one piece would remain stuck to the floor and immovable while the other would fall to the ground as a dynamic physics object. The dynamic part was also classed as a swappable weapon, meaning the user could swap it with his current weapon.

This was done using entity templates. An entity template could be written, describing the components and default properties of an entity. This template could then be specified for with the parts created by the cut. Geometry and physics data would be set based on the results of the cut. Presently the mass is divided evenly across the new pieces. Given more development time a more advanced system could be written to use an estimate of the object size to assign mass more realistically.

As mentioned in the previous example, different parts could be given different properties. However, this was limited. An object could have a vertex specified as a fixed point. Which ever resulting piece or pieces of the object contained this point would be set a fixed physics objects, meaning that they didn't move but did interact with other moving objects. The remaining pieces would use the specified entity template.

A particle system can be created when an object is cut. The particle system must be created as an entity template. After a specified period of time it is deleted. One system is created per piece after the cut, in the center of the cut surface and parented to the piece to follow it as it moves. This was used for blood when enemy characters were cut, and wood chips and sawdust when wooden objects were cut. Other possible effects include sparks for metal objects. Wood chips and sawdust only appeared for an instant, knocked off by the sword, while blood lasted longer.

The main shortcoming was that the system would disappear quite suddenly. This was not an issue with the sawdust as the entire particle limit was created in the first instant, then the system was destroyed before the particle lifetime expired and more could be created. However, with the longer-lasting blood particle system, the cut-off was very noticeable and unpleasant. This could not be solved without a more advanced particle generation system than that provided by the engine.

It was found that the physics engine could not correctly deal with a physics entity being parented to a moving entity. Because of this enemy characters weapons were not physics objects. However, this meant that when enemy characters were killed their swords, separate objects parented to a bone in the models hand, would remain floating in mid-air as the original object was destroyed. In order to solve this problem the sword had to be destroyed and replaced with a new dynamic physics entity. This led to the addition of a new feature, in which an external object could be specified to be destroyed and replaced with a new object of a different type. While limited, there could be some other possible uses for this feature.

The weapon component stored the length of the blade, the weapon damage, and the force applied during a cut. The length was used for determining when an object had been cut. The damage was for combat. The force was used when applying force to the pieces left after a cut. The user could pick up different weapons with different attributes.

Because the weapons could not be physics objects, this had to be done by swapping all the weapons attributes with the values stored in the players weapon component, and by swapping the geometry mesh name of the player weapon with the new one. Unfortunately this is imperfect in that the weapons may have significantly different sizes or shapes, meaning that the free weapon may end up with a mismatched physics shape. Additionally it limits the system, for example it prevents the use of weapons with light sources or attached particle systems such as a flaming brand.

As mentioned previously, when cutting complex models the game would freeze for a brief moment. Only the enemy character models were complex enough to cause this. Therefore a trick was used to reduce the effects. When the program detected the cut, but before it was actually performed, a blood-splatter sprite is placed along the path of the cut through the character. This is rendered, then the cut is performed, then the sprite is removed. Therefore, during the moment of hesitation, the position of the cut is highlighted with a blood splatter through the character. This created a dramatic effect that increased the impact of the cut on the player, similar to deliberate pauses used by some games during high-power moves. In fact some play-testers expressed the opinion that it worked even better than if there was no hesitation.

During development concepts for several possible puzzles based on the system were considered. The simplest use would be to cut through obstacles such as tree branches and doors in order to pass. This could be extended to include walls that look similar to normal, non-cuttable walls except for a small yet recognizable distinguishing feature. This would allow the alert player to cut through to hidden areas and be rewarded for his diligence. Cutting standing objects such as thin trees in the correct angle could create bridges or ramps to cross ditches or climb up walls. Smaller objects would make useful steps to access higher ground. Cutting a rope or chain could drop hanging objects. Cut a box at the right angle and a ramp is created. It can also be used for other things, for example a rope bridge can be cut at the right moment to drop an enemy into a ravine.

4.7 Conclusions

The system proved very viable for use on simple models, meaning it would work well as a game mechanic. For example, it can be used to cut environmental objects in order to solve puzzles. This alone could be used to good effect.

Cutting models as complex as characters, however, is less feasible. There is room for improvement in the system that may bring speeds to usable levels. The need to save the mesh to file and read it in again was a major source of slowdown. Intersection testing could be sped up using trees. The triangulation of new surfaces was not optimised, a more efficient system could have a significant impact. It may even be possible to perform some operations on a GPU. Therefore on powerful hardware, with a more efficient purpose-built engine it should be feasible.

Despite the problems with character cutting in the current system, feedback from play-testers was generally positive. People found it enjoyable, combat in the game was made more entertaining by the greater impact and realism of a kill.

The recent trend of more versatile control systems would work very well with this system. The Nintendo Wii controller and Nintendo DS touch-screen are worth mentioning at this point. The DS already has a surgery game called Trauma Centre, in which the stylus is used as various medical instruments including a scalpel. The Wii has games in which the user controls a sword and other weapons using the controller, such as Samurai Warriors: Katana by Koei. These would work very well with the cutting system.

Additionally, on modern multi-core systems there is more likely to be spare processing power available, making the system more feasible. All these factors make the cutting system a realistic and promising prospect for future games.

4.8 Taking It Further

The system could be made more sophisticated by using more realistic cutting planes. Using a semicircle or a quad could work. One possible change is to use a single line to cut. When the line is moved by a minimum amount, the old and new positions can be used as opposite sides of a quad (or if the positions intersect, sides of two triangles) which is used as the cutting object. While using a finite plane means there are far more possible intersection cases to deal with, the biggest issue is creating the new surface. Using anything other than an infinite plane creates complications as the edges may not form complete loops.

Using different data structures could speed up the basic processes. For example, arrays of addresses rather than actual objects may save time in functions where large arrays need to be re-sized. Instead of building an entirely new list of triangles, the old vector can be modified and new triangles added to a separate, smaller list (since most triangles aren't cut, there is no reason to have to add these to a new list then copy them back later).

Since only a small number of the triangles will normally intersect the cut, using trees such as AABB trees to check simple intersections with the plane would likely speed up the process of calculating the intersections [19]. However, re-calculating the trees for the new meshes may cost more time than is saved, it would require experimentation to find out if there is an overall benefit or not. This means that trees would be ideal for complex meshes that would only be cut once, as they would not need to be recalculated. Alternately, some form of tree may be needed anyway, for example for shadowing; efficient design could take advantage of this.

Textures could be linked to triangles in a different way (e.g. each triangle has a reference to its texture, then they can be sorted when building vertex lists), then multiple textures can be used for one mesh, allowing for far better texturing of new surfaces. What's more, it may be possible to use 3D shaders with a model for far more convincing internal texturing after a cut. Alternately some form of material-based procedural texture generation could be used.

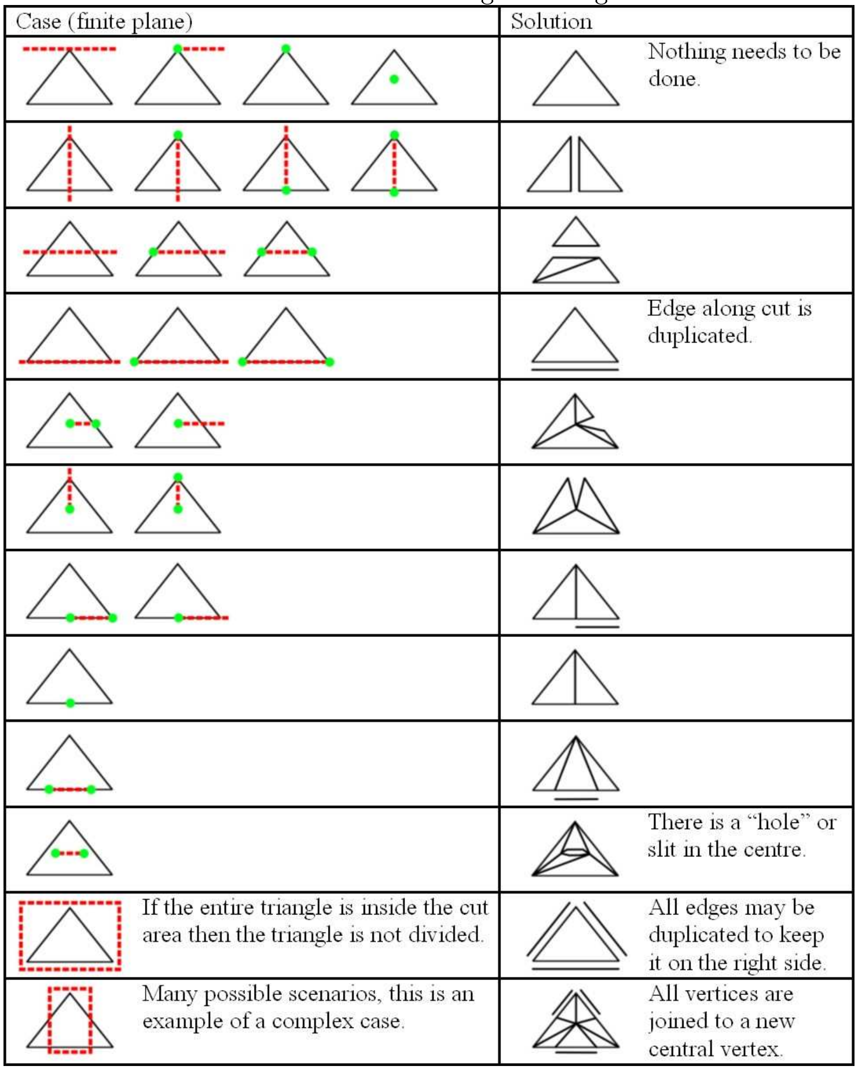
With a more advanced "connectivity" system it could be used for models with multiple meshes, or maintain skeletal data e.g. for rag-doll physics. This would take some work to find the best way to do it for each application. For example, for multiple meshes vertices that are "shared" by more than one mesh can be marked or referenced in a list. New vertices created between two marked vertices would also be marked or added to the list. In a later stage, after all the new meshes have been sorted into new objects the marked vertices could be checked to find which of the new meshes are connected through them.

In theory, in the development of a full game integrating with other systems, like Natural Motions euphoria; an in-game technology that creates intelligent reactive behavior of non-playable characters [20]. For example a character has a limb cut off and tries to keep fighting, perhaps picking up a fallen weapon in his other hand.

The triangulation of the new surfaces can be a rather expensive operation. Currently triangulation is performed separately for each new part, however realistically the shapes of new surfaces are identical above and below cuts, so it should be possible to devise a scheme where the same surface is only triangulated once then copied rather than being triangulated above and below the cut. Furthermore, it should be possible to optimise some operations in the triangulation phase.

4.9 Tables and Figures

Table 4.1: Possible Situations When Cutting A Triangle With A Finite Plane.



Case (finite plane)	Solution
	Nothing needs to be done.
	Edge along cut is duplicated.
	There is a "hole" or slit in the centre.
If the entire triangle is inside the cut area then the triangle is not divided.	All edges may be duplicated to keep it on the right side.
Many possible scenarios, this is an example of a complex case.	All vertices are joined to a new central vertex.

Table 4.2: Possible Situations When Cutting A Triangle With An Infinite Plane.

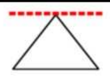
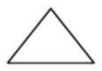
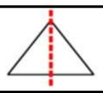
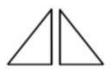
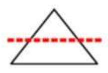
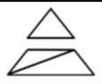
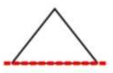
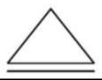
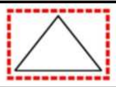
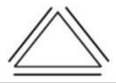
Case (infinite plane)	Solution
	 Nothing needs to be done.
	
	
	 Edge along cut is duplicated.
	 All edges may be duplicated to keep it on the right side.

Table 4.3: Extra Cases For Soft Bodies.

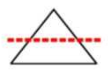
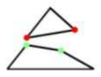
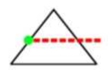
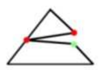
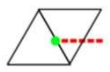
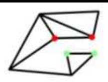
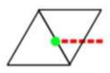
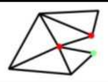
Case	Result	
		The points must be separate for a soft body as the coordinates will change.
		However, if the cutting plane's edge intersects the triangle edge it must be treated as a separate case.
		If we treat it as the same case, then we get incorrect results as the adjacent triangle should have only one vertex.
		Therefore some extra considerations are needed for soft bodies with finite planes.

Table 4.4: Triangulating A New Surface.

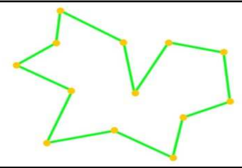
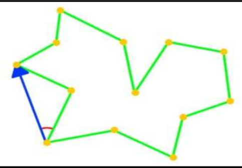
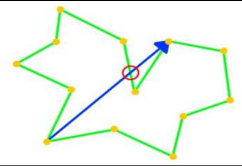
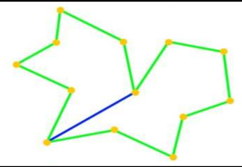
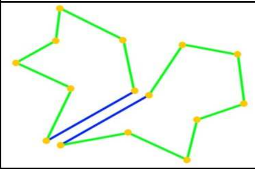
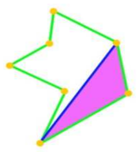
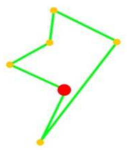
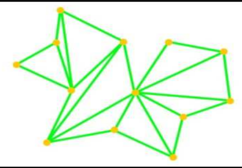
	Each ring is a polygon that needs to be triangulated to form a new surface. The triangulator tries to create new edges between vertices.
	This edge goes on the wrong side of the first edge, and therefore is rejected.
	This edge intersects another edge and therefore is rejected.
	When an edge is found to be acceptable, the polygon is subdivided into two new polygons and the function is called recursively on each one. This edge is added to the object's list of edges.
	In fact a new edge is added twice to the list that represents the polygon as it is part of both polygons. This is a separate list from the object edge list. While only one list is used, iterators mark the first and last edge of each polygon.
	When a triangle is found it can be added to the object triangle list.
	In this polygon, the original first point cannot form an edge to any other points. Therefore the list is "rotated" but removing the first edge and adding it at the end of the segment of the list. There is now a new first vertex, and the function is called recursively again.
	Eventually the entire polygon is triangulated.

Table 4.5: Triangulating Concentric Rings.

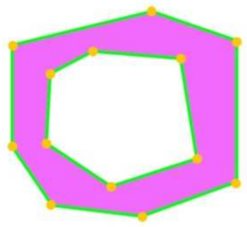
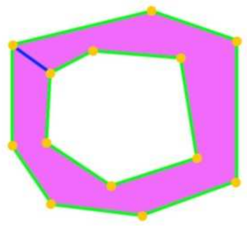
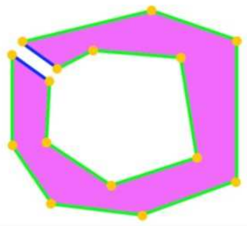
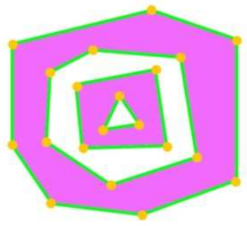
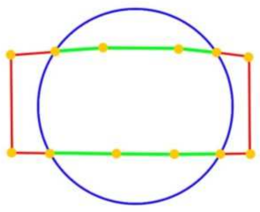
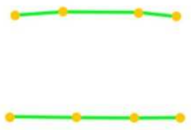
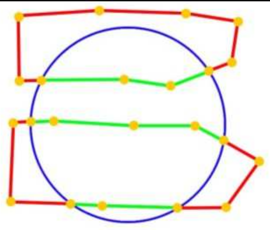
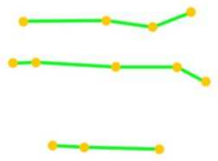
	The main complication that can arise is if two rings are in fact inside each other. This would be a very useful situation to be able to handle, as it would allow hollow objects such as boxes or hollow bamboo to be cut.
	Unfortunately the obvious way of checking whether any rings are concentric is very inefficient as it involves checking a vertex from one each ring against all the edges of all other rings. Once concentricity is found it could be solved by connecting two vertices from the two rings.
	In essence this would work like a single polygon of unusual shape, and could be solved in the normal way.
	However this would not be enough, as it is possible to have many concentric rings. The possibility of such situations makes analysis of concentric rings difficult and slow, which is why it was not considered in this project. Any concentric rings that arise would simple each be triangulated as a simple polygon.

Table 4.6: Triangulating Cuts From A Finite Plane.

	A finite cutting plane, however, introduces a new complication. It is possible to have incomplete loops. This is a situation that is not considered by typical Delaunay triangulators.
	This means that the program has to join them together to create full loops. In theory this is possible by joining the end points together to create a loop.
	However there is no guarantee that there will be exactly two loops, there may even be an odd number of loops.
	Using the face normal of the triangle would be necessary to resolve this issue, but even then the solution is complex. Since it is possible for an object to be cut but not split, a triangulator for a finite cutting plane would need to be re-written to triangulate the same face twice for one object.

4.10 Acknowledgments

I would like to thank:

- Jon Macey, obviously.
- Bournemouth University.
- All of Ergophobia, for being so great and especially for putting up with me for so long.
- Ben Chandler for helping with the maths.
- EA studios for all the help and guidance.
- Dare to be Digital, for the great opportunity.
- All my friends and family for all their support.

Chapter 5

The AI System

By Hasan Atieh.

Acknowledgement

First of all I thank God.

I would like to take this opportunity to thank Jonathan Macey at the NCCA, Simon Pick at Electronic Arts, the kind people at Instinct Technologies and Haytham Mustafa for their efforts in providing guidance and motoring through out the developing the AI system.

Also a special thanks goes to my parents for their support.

5.1 Introduction

As the game development has become a main stream industry in recent years, all aspects of game development have been witnessing considerable advancements. And game Artificial Intelligence, or AI, has recently been receiving increased awareness and attention amongst game developers and players alike. This has been helped particularly by the introduction of powerful consoles and computers with increased processing power, especially by transferring the processing of graphics from the CPU to the GPU which freed up CPU cycles for the AI programmers to utilise.

In this chapter, the two main tasks carried out by the AI programmer will be discussed. The first task described is the development and implementation of a relatively industry-compliant Finite State Machine (FSM) system for the game AI. The developed system needs to make it fairly easy to add new states and transition conditions and also to allow non-programmers to change and design the behaviours of a non-player character (NPC) in the game. Moreover, it needs to make it possible for the game designers to change various properties of the agents in order to personalise the different agents that share the same behaviour.

The second task is to design a behaviour model for the agents in the game in a way that would serve the stealth gameplay elements of the game. This also includes writing the senses and the states libraries suitable for this task. The aim of this work is to suggest a behaviour that would convey a sense of intelligence to the agents inhibiting the game world utilising the developed FSM system. Again, the designed behaviour should handle the interactions with the player to emphasise the stealthy nature of the game and most importantly in a way which makes it fun to play against.

Other tasks and contributions including Subtitles tool in addition to debugging and optimisation are not discussed.

The work carried out on the AI of the game is an amalgamation between a FSM system suggested by Simon Pick, who is a senior AI programmer at Electronic Arts - UK and the FSM system described in Matt Buckland's book "Programming Game AI By Example" (2005). The implementation was extended to include the ability to change the FSM of the agents during runtime which is known as a hierarchical FSM.

Unlike the case in a conventional academic research, the work carried out on the game in general took on a development approach rather than a research one; the difference being that in development, a problem must be iterated over and over until a solution is found, whereas the research method involves analysing a technique and possibly concluding that it does not solve the problem at hand.

5.2 Previous Work

The FSM structure is probably the most established AI technique used in video games so far. Almost every game will have a representation of FSM. One of the most obvious implementations of FSMs in games would be the 1980 Pac-Man game. A ghost in Pac-Man can be either in the chase state, the evade state, the dead state or the wander state and the transitions between states are triggered by different conditions. For example, eating the power pill is the condition that triggers the transition from the chase to the evade state, the wander state is triggered after the player dies. While the ghosts had the same actions for some states like the evade state, they had different, more personalised implementations of other states. For instance, in the case of the red ghost, the actions of the chase state were to chase the player directly while the blue ghost in the same state would simply wander randomly. (Buckland, 2005)[15].

Many other games have implemented the FSM technique. Players in sports simulations such as the soccer game FIFA2002 are implemented as state machines. The NPCs in RTSs (real-time strategy games) such as Warcraft, make use of finite state machines. Car racing games fall in the same category too (Buckland 2005)[15]. Brian Schwab's (2004)[13] discusses in his book "AI Game Engine Programming" major game genres and the AI techniques used in them. Not surprisingly, FSM appeared in most the genres he discussed except for flight simulators.

The Quake and Quake II games showed higher potential of the FSM. It is since ID Software released the source code to the those two projects, that people have noticed that the movement, offensive, and defensive strategies of the bots were controlled by a simple FSM. The use of FSM in these games was extended to control the behaviour of entities other than the bots of the game. For example, the rocket had a FSM which contained states like spawn, fly, explode, etc. (Schwab 2004)[13].

FSMs are not the only technique used in decision making for NPCs. Fuzzy Logic has also be used in many areas of game AI. In the case of 3rd person shooters, it can be combined with FSM to form a Fuzzy State Machine or FuSM adding an element of non-linearity to the whole behaviour thus making it less predictable (Watt & Policarpo 2001)[9]. Among the reasons for not implementing a FuSM, is the small time of player-guard interaction.

Artificial Neural Networks (ANNs) can also be used as an NPC decision making technique. The network can have several inputs representing the facts upon which the decision is to be based. Examples of such inputs can be the health of the agent, the weapon it has and the distance to the player. The output can be to chase, evade or patrol. ANNs was not used in this project due to their inherently non-deterministic nature. The produced behaviour would have been hard to test, debug and tweak. However, if time had allowed, neural networks might have been implemented to control which FSM an agent needs to use at any given time in the game.

The Sims offered a new approach to controlling NPC behaviour. It provided a novel combination of A-Life and fuzzy logic to control the behaviours of the agents. The idea behind which is what the game designer Will Wright refers to as "Smart Terrain". According to him, the rule based approaches to AI are very inflexible. In Smart Terrains, the actions a character performs when interacting with an object is embedded in the object itself rather than the character. This allowed for the introduction of new objects to the environment, making sure that the characters would be able to interact with them. This explains the many expansions and add ons that have appeared after the launch of the game.

(Woodcock 2007)[14]

Although this kind of approach can be useful for this type of game (The Silence of Night), it is beyond its requirements especially after considering the short time of player-guard interaction. Also, the agents are not required to have that level of interaction with the environment which makes the use of this method unnecessarily complicated.

Other AI techniques can be applied to control lower-level NPC behaviours. An example of which would be the use of Potential Functions for chasing/evading movement and obstacle avoidance. It can also be used for swarming and flocking. The basic theory behind it is to apply positive or negative force to an entity proportional to the squared distance between it and the other entity of interest (Bourg & Seemann 2004)[8]. This technique is classified as a low-level AI because it acts directly on the coordinates of an entity. For example It can help an agent avoid obstacles while moving from point A to point B but it does not initiate the decision to move in the first place.

5.3 Theoretical Background

Historically, FSMs were first proposed and used by mathematicians in representing and solving problems. Perhaps among the earliest references to finite state machines would be the Turning machine which Alan Turing talked about in his 1936 paper "On Computable Numbers". (Bucklands 2005)[15].

In mathematics, a FSM (also known as Cellular Automaton) is usually represented with a quadruple of sets, these are:

- A set I called the input alphabet.
- A set S of states that the automaton can be in.
- A designated state S_0, the initial state.
- A next state function $N : S \times I = S$, that assigns a next state to each ordered pair consisting of a current state and a current input.

(Luger 2002)[7]

Some books add a fifth set F of final states which is a (possibly empty) subset of S to the representation of the FSM.

Here is a more descriptive definition of FSM provided by Buckland:

> A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and

> can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time. (Buckland 2005)[15]

Many types of FSMs exist and can be classified according to different factors. For example a FSM can be classified as a deterministic FSM if for every state, a possible input would match only one state transition. On the other hand, in the non-deterministic FSM, a possible input for a given state would result in one or more than one state transition. Other classifications exist according to where an output is generated as in the Moore and Mealy machines. (Black 2006)[10]. Implementations of FSMs in games is hard to classify under a specific type and in many cases a FSM can fit under more than one. However, a more games-related classification of FSMs can be found in Martin Brownlow's book "Game Programming Golden Rules" (2004)[11]. He specifies two kinds of FSMs; these are Explicit vs. Implicit.

The difference is that in the case of explicit FSMs, the FSM does not need to know about the object it is maintaining the state for. It acts like a black box and has events as its inputs that it uses for changing the states. An extension on the concept at the opposite end is an implicit FSM. Implementing explicit FSMs have many advantages over implicit ones; the source code becomes smaller and easier to maintain, the behaviour of game objects can be altered quickly and easily without recompiling code and the designers are now free to experiment with object behaviours without bothering the programmers. (Brownlow 2004)[11].

The FSM system developed for this project can be classified as a deterministic explicit FSM.

5.4 The FSM Solution

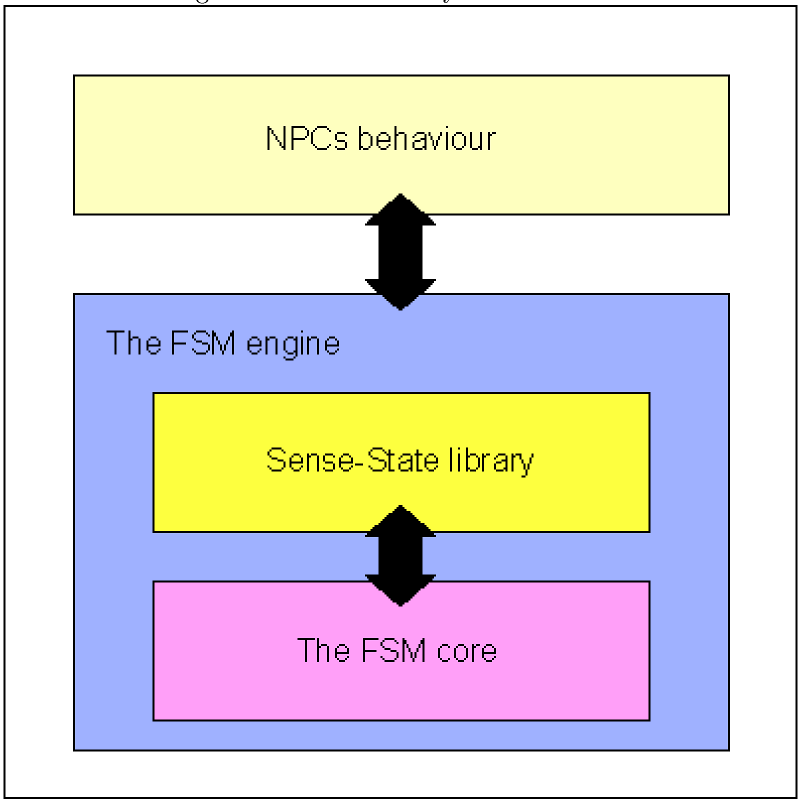
Many factors contributed to the decision of implementing a FSM solution for the game AI. Among these, is the power and effectiveness of FSM in modelling NPC behaviour. Given the requirements of the guard's behaviour of the game, FSM has the potential to meet these requirements to a high standard if properly used. It is true that other less-deterministic AI techniques can also provide a very good illusion of intelligent NPC behaviour but implementing such techniques implies taking higher risk due to their inherent non-deterministic nature and also due to the limited development time frame available on hand.

Other reasons for using FSMs can be attributed to the characteristics of this technique. These include their simplicity, ease of debugging, little computational overhead and flexibility. (Buckland 2005)[15].

Initially, the implementation of the FSM started by implementing the FSM explained in Buckland's book. The output of that FSM was in the form of plain text printing to the console each state execution and transition. Fundamental changes were done afterwards on that implementation in order to reflect Simon's approach to FSMs which mainly implied moving the rules affecting the state transition from inside the states to the agents running the states. After that, changes took place in the direction of integrating it into Instinct which included developing senses and states to produce a basic NPC behaviour. Finally, work was carried out on designing and fine-tuning the behaviour of the guards which also went in parallel with adding more senses and states to extend their behaviour.

This section is divided into two subsections, the first one deals with the development of the FSM engine. This includes the development of the core system and the senses and states that plug into the FSM. The second section deals with designing the behaviour of the NPCs that inhibits the game world. Figure 5.1 shows an over-all structure of the FSM system.

Figure 5.1: The FSM System Structure



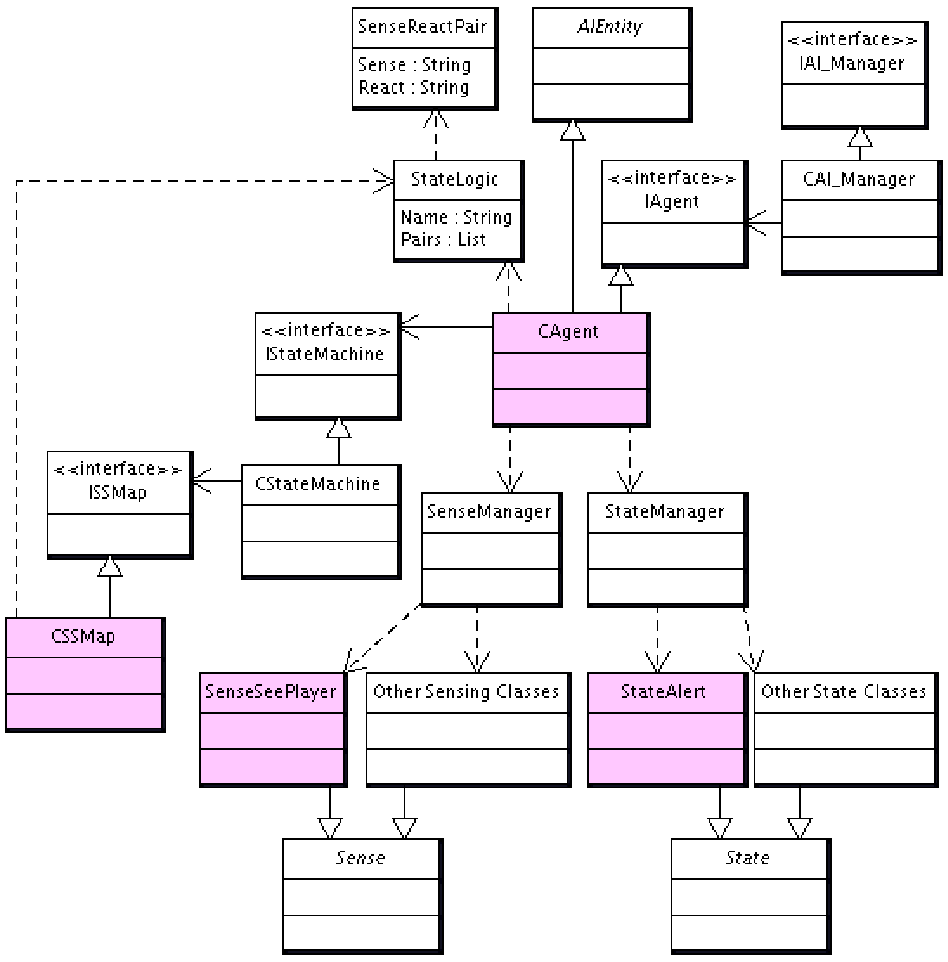
The over-all structure of the FSM system in the game.

5.4.1 The FSM System

5.4.1.1 The FSM Core.

At its heart, the developed FSM system has four main classes or types of classes that construct the core of the system. These are the senses classes, the states classes, the Sense State Map entity component (SSMap) in addition to the Agent entity component that drives the overall thinking process. Of course there are many other classes implemented but their main purpose is to add more flexibility, abstraction and organisation to the whole FSM system. This subsection (The FSM Core) will focus on discussing the main classes in details while the other classes will be discussed whenever the context requires but in less details. Figure 5.2 shows the over-all classes digram of the implemented FSM system with the four main classes having a darker background. Note that only one Sense class and another State class are shown as examples to avoid ridiculously increasing the size of the graph.

Figure 5.2: The Class Diagram



An over-all look at the classes that construct the FSM system. The core classes are coloured in pink.

The Senses Classes

The senses inherit the abstract Sense class template. Moreover, they are all single tone classes which means that only one instance exists of any given sense and is therefore shared between all the agents that use that sense. All sense instances have a public method called CheckSense that accepts a pointer to an agent and returns a boolean. By passing a pointer to the agent for which the sense is called, the CheckSense performs the checking according to that specific agent. For example, SenseZeroHealth will call CurrHealth method of the Agent pointer to get its current health.

In order to make it easier to retrieve and check senses, a SenseManager which is also a single tone class, is created to hold a std::vector of all the senses available to an agent.

The States Classes

A state class is very much similar to the sense class. It inherits an abstract State class template and is also represented as a single tone. Instead of a CheckSense method, a State class has three public methods. These are Enter method, Execute method and Exit method all of which accepts a pointer to an Agent. As the names suggest, these three methods are called on various times according to the life span of the state.

The Sense-State Map (SSMap)

The SSMap is actually the entity component that declares the CSSMap class which is short for Sense-State Map and not to be confused with Cascade Style Sheet.

What makes this class an important one is the fact that it holds a std::vector of a structure called StateLogic. This vector represents the transition table of the FSM. Table 5.1 shows an example of this vector with two entries.

Table 5.1: A StateLogic Example

First entry	StatePatrol	SensePlayerDead	DoNothing
		SenseSeePlayer	StateAlert
		SenseHearPlayer	StateTimedAlert
		SenseFellowAgents	StateAlert
Second entry	StateTimedAlert	SensePlayerNear	StateFight
		SenseTimedAlertTooLong	StateDefault
		SenseSeePlayer	StateAlert

An example of a StateLogic vector with two entries.

The StateLogic structure is basically the representation of the logic that needs to be followed when the agent is in a certain state. That state is indicated by an std::string member of the StateLogic structure. In the example above, it is StatePatrol for the first entry and StateTimedAlert for the second.

However the actual logic of the state is represented using a std::vector of another structure called the SenseReactPair structure. This is a much simpler structure and it holds two std::string members one representing a sense to be checked and the other representing the reaction if the sense evaluates to true.

To add a layer of separation between a suggested behaviour and the system which runs it, the behaviour of an agent was provided to the SSMap component via an external XML file which is parsed into the StateLogic vector upon initialisation. See lesting 4.

Listing 4 An Example of the state-logic XML

```
<?xml version="1.0" encoding="utf-8"?>
<SSMap>
  <state name="StateAttack">
    <pair sense="SenseAttackDone" react="StateFight" />
  </state>
  <state name="StateRecoil">
    <pair sense="SenseAgentZeroHealth" react="StateGotCut" />
    <pair sense="SenseSecAttacked" react="StateFaint" />
    <pair sense="Default" react="StateFight" />
  </state>
</SSMap>
```

The Agent.

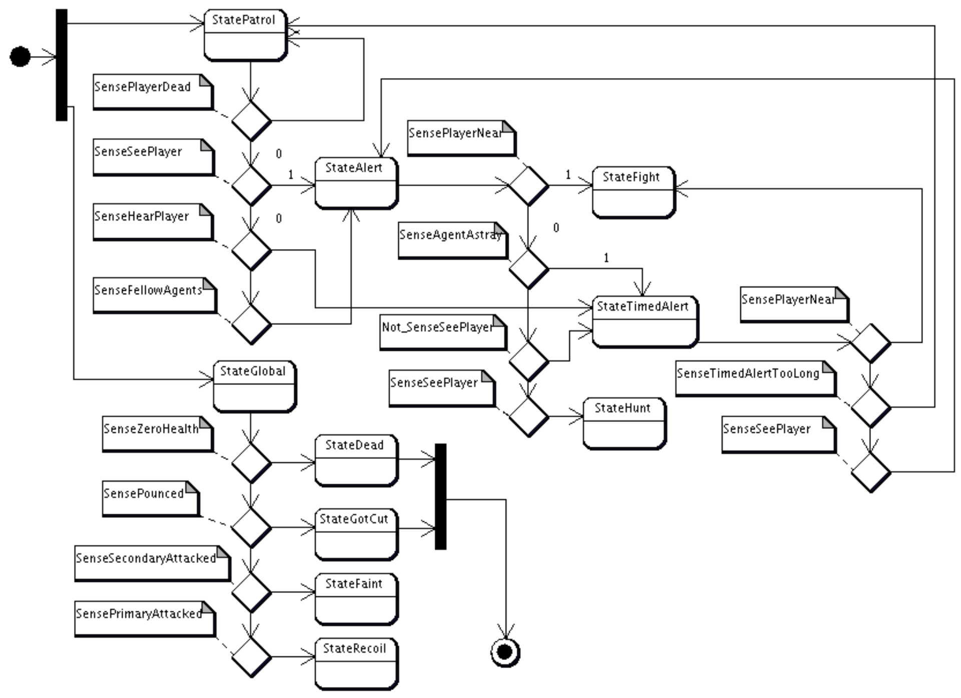
Now that enough traction has been built explaining the previous classes and components, it is possible to discuss the Agent entity component. This entity component gains its importance from the fact that it serves as the central point where all the other classes come together. Just like the SSMap, the Agent entity component declares the CAgent class which defines member variables and methods to drive the FSM. Among these variables are four pointers to states; these are for the current state, the previous state, the global state which is executed along side the current state, and the default state which defines the main job of the Agent in the game be it patrolling or guarding or anything else. The most important method of the CAgent class is the HA_Update method which is called consistently at a fixed time intervals. Listing 5 shows a simplified version of the pseudo code for this method which can also be thought of as the pseudo code for the core FSM system.

Listing 5 A Simplified Pseudo Code for the HA_Update Method.

```
Execute current state
Execute global state
Loop over the StateLogic list of the Agent
    If name of a state in StateLogic list == name of the current state
    or if name of a state in StateLogic list == "StateGlobal" then:
        Loop over the SenseReactPairs list of that StateLogic entry
            If this Agent check sense in SenseReactPair then:
                handle react in SenseReactPair and break from the inner loop
            end if
        end loop
    end if
end loop
```

The way this algorithm works is that for the state that matches the current state's name, the senses in the list of sense-state pairs are checked in order until a sense evaluates to true. At that point the reaction that is paired with that sense is handled by a specific method of the agent and the algorithm breaks from the inner loop but carries on with the outer loop. The reason why the outer loop needs to carry on is to guarantee that the logic of the global state is executed. See figure 5.3.

Figure 5.3: The States Transition Diagram



A diagram showing part of the state transitions.

In the SenseReactPair, the Sense string can be a name of a valid sense which can be suffixed with "not_" to inverse the value returned by the CheckSense function. The other choice for sense is simply using the string "Default" which will always validate to true when passed in the CheckSense function. The "Default" can have the effect of the else keyword when used in the last SenseReactPair in a StateLogic entry or it can be used to serve as a debugging option, for example forcing a state change. Howerver, valid values for the React part in the SenseReactPair are "StatePrevious", "StateDefault" and "DoNothing" in addition to a state name. Note that "StatePrevious" and "StateDefault" serves as a memory of the agent. "DoNothing" can be used to keep the agent in its CurrentState or as a another debugging option.

Any entity - including the player character - thet needs to utilise the FSM, must have the Agent component among its list of components. In the case of the player character, it's Agent component runs a simple FSM that checks to see if the agent's health has dropped to zero. If so, then it will react by going into the dead state which plays the dying animation and turns off the components of the agent. This could have been used to add a bored state to the player when the player remains idle for a long time but was given a low priority for obvious reasons. Among its properties, the Agent component has a property that holds the name of the AIManager in the game which is also an entity component. The Agent will register to the AIManager upon initialisation if the name of an AIManager is provided.

5.4.1.2 The Senses/States Library.

After having built the core of the AI FSM system, there needs to be a number of senses and states classes that it can operate on. For that reason, a group of senses and states was developed. In addition to the importance of making what we had an actual game, the implemented senses and states had to emphasise a certain gameplay style of the game which is the stealth style. As a result, almost every interaction with the player, required the guards senses to be aware of environment elements like the level of light or the noise the player was making. Also, other senses were developed like testing if a guard knew about the player before getting hit by it in addition to other senses of which results were reflected in the score manager. In all, 27 senses and 18 states were developed which were combined to propose an intelligent behaviour for the guards and introduce the gameplay elements of the game. The rest of this subsection (The Senses/States Library) will explain only one sense and another state class. These are the

SenseFellowAgents and StateAlert and are provided just as an example. For the rest of the classes please refer to the accompanying CD.

SenseFellowAgents

As mentioned before all the senses classes has a CheckSense method which accepts a pointer to a CAgent class and returns a boolean of whether the sense evaluates to true or false. Most of the senses depend on other external entity components or even other senses to be able to check for that specific sense. An example of which can be found in the SenseFellowAgents class which depends on the AIManager component in testing the state of fellow agents. Remember that the AIManagre knows about the agents in the game since they register to it upon initialization. This sense will return true if a near by fellow agent is in an alert state or a similarly tensed state. Listing 6 shows the pseudo code for the CheckSense method of the SenseFellowAgents class.

Listing 6 The Pseudo Code for the SenseFellowAgents

```
Get a reference to the AIManager
Get a pointer of the agents list from the AIManager
Loop over the agents list
FellowAgent = Agents list [i]
If the FellowAgent is the player character or the same agent then:
Continue
Test the state of the FellowAgent
If state name != "StateDead"
&& state name != "StateAlert"
&& state name != "StateTimedAlert"
&& state name != "StateFlee"
&& state name != "StateFight"
&& state name != "StateFaint" then:
Return false
Check the distance between the agent and the FellowAgent
If distance is close enough then:
Return true
Else
Return false
```

Again, a sense can rely on other senses to check its value. For example the SensePriAttacked will call the CheckSense for "SensePlayerFacingAgent" and for "SenseInPlayerRange" in addition to testing if the player is performing the attack animation. The design of the system lends itself smoothly to this kind of implementation which is made possible by passing a pointer of the agent to the CheckSense method of the sense in addition to making sure the senses are checked through a member method for the agent.

This adheres to code re-usability by making simple senses and combining them together in order to construct more complicated ones.

StateAlert

The states classes have three main public methods that can be used to specify how the agent will behave while in that state. These are the Enter, Execute and Exit. While this is not always the case, sometimes an agent does not need to perform tasks in all three methods of the state. Nevertheless, the three methods are called automatically in various places in the code. The calling for the Execute method has been shown in Listing 5. Listing 7 shows the pseudo code for the ChangeState method of the agent which is where the Enter and Exit methods for a state are called.

Listing 7 The Pseudo Code for the Agents ChangeState Method

```
If NextStateName != StateCurrent -> Name then:
StateNext = StateMngr -> GetState(NextStateName)
StatePrevious = StateCurrent
StateCurrent = StateNext
StatePrevious -> Exit
StateCurrent -> Enter
Return true
Else
Return false
End if
```

The AlertState, however, needs to make changes either on the Agent or on other components when entered or exited and while executing by an agent. See listings 8, 9 and 10.

Listing 8 The Pseudo Code for the Enter Method of the AlertState

```
Get a reference to the agentInputController
Get a reference to the agentSoldier
Set the animations in the agentInputController to false (walk, run, strafe, attack,
etc)
Set the fight stance of the agentSoldier to true
Call the agentSoldier CreatVocals method
Display the subtitles of the created vocal
Get a refenence to the ambient sound manager
change ambient music to that of the suspense track
Get a reference to the score manager
Increase the number of agents alerted in the score manager
```

Listing 9 The Pseudo Code for the Execute Method of the AlertState

```
Turn the guard in place to face the player
```

Listing 10 The Pseudo Code for the Exit Method of the AlertState

```
Get a reference to the agentSoldier
Set the fight stance of the agentSoldier to false
Get a refenence to the ambient sound manager
change ambient music to that of the suspense track
```

5.4.2 The Proposed Behaviour

Of course developing an FSM system is simply half of the work that needed to be done on the agents AI. The other half was to design the behaviour of the guards utilizing the developed FSM. This task is crucial not only to push the game in the 3rd person stealth direction, but also to make it 'a game' rather than just a simulation of a character running around.

Naturally this task is more of a soft skill; i.e. the behaviour of NPCs needs to feel right rather than necessarily be right. According to Simon Pick, it is general practice to assign the task of tweaking and fine tuning of the NPC behaviour to the designers. Doing so leads to better more engaging AI behaviour. (Personal communication, 06 Sep. 2007)[12].

The work done on designing the NPC behaviour did not accurately adhere to this code of practice mainly because of the relatively small scale of the project either time wise or team-size wise. Also, since the development of the FSM system was roughly going in parallel with the design of the behaviour, it was convenient to keep iterating on both to build the basic behaviour. Instead, the behaviour was mainly designed by the AI programmer with considerable contributions and suggestions from all the other team members.

The designed behaviour of the guards is shown in table 5.2. It is fairly easy to read through it however, StateFlee might need some explanation. The flee state is entered if the health of the guard goes beneath a certain level which is a result of receiving damage by the player. This is being tested in the SenseLowHealth of the StateFight. While in the flee state, the guard runs in the opposite direction of the player and performs three tests; if the player is dead it will return to its default state, if the player is far it will go into the alert state, and if, while fleeing, an obstacle is faced it will go into a desperate state machine. This last state is actually a separate FSM and sets the StateMachine component of the guard to point to the desperate SSMap component in the scene. The main difference between the two FSMs is that the guard will not continue fleeing while in the desperate FSM since it is theoretically trapped. Instead, the guard will go into the alert state facing the player and will only chase the player if the player's character turned its back to it. This approach resulted in more intelligent behaviour on behalf of the guards especially preventing them from running into walls.

Table 5.2: The Guards State Transition Table

Current State	Sense	React
StateIdle	SensePlayerDead	DoNothing
	SenseSeePlayer	StateAlert
	SenseHearPlayer	StateTimedAlert
	SenseFellowAgents	StateAlert
StatePatrol	SensePlayerDead	DoNothing
	SenseSeePlayer	StateAlert
	SenseHearPlayer	StateTimedAlert
	SenseFellowAgents	StateAlert
StateTimedAlert	SensePlayerNear	StateFight
	SenseTimedAlertTooLong	StateDefault
	SenseSeePlayer	StateAlert
StateAlert	SensePlayerNear	StateFight
	SenseAgentAstray	StateTimedAlert
	not_SenseSeePlayer	StateTimedAlert
	SenseSeePlayer	StateHunt
StateHunt	SensePlayerDead	StateDefault
	not_SenseSeePlayer	StateTimedAlert
	SenseAgentAstray	StateFight
	SensePlayerFar	StateAlert
	SensePlayerNear	StateFight
StateFight	SensePlayerDead	StateDefault
	SenseAgentLowHealth	StateFlee
	SensePlayerIdle	StateAttack
	SensePlayerFar	StateAlert
	not_SensePlayerNear	StateHunt
	SenseAgentThreatened	StateBlock
StateFlee	SensePlayerDead	StateDefault
	SensePlayerFar	StateAlert
	SenseObstacle	StateDesperateSM
StateAttack	SenseAttackDone	StateFight
StateBlock	SensePlayerIdle	StateFight
StateRecoil	SenseAgentZeroHealth	StateGotCut
	SenseSecAttacked	StateFaint
	Default	StateFight
StateFaint	SenseFaintTooLong	StateDefault
StateDead		
StateGotCut		
StateGlobal	SenseAgentZeroHealth	StateDead
	SensePounced	StateGotCut
	SenseSecAttacked	StateFaint
	SensePriAttacked	StateRecoil

This table shows the state transition table of the proposed behaviour of the guards.

It can be noticed from table 5.2 that the StateDead and StateGotCut does not have any sense-react pairs to them for obvious reasons. Also sensing that the player is dead in most of the states aside from the StateIdle and StatePatrol, will cause the guard to switch to its default state while the same sense in StateIdle and StatePatrol will prevent the guards from testing the rest of the sense-react pairs.

5.5 Discussion

According to S. Pick, the way the industry approaches game AI, is that the AI programmer develops the system so that it allows the game designers to change the behaviour of the AI entities externally without having to refer to the AI programmer for each change. The reason being that the game designers need to test and tweak many times in order to get the right feeling of the game AI. This kind of approach can be very beneficial especially in saving the time of both the AI programmers and the game designers. Moreover, the compiling time in general will be substantially decreased as well (personal communication, 06 Sep. 2007)[12]. Of course in order to achieve such a level of flexibility, the system must have a clean object-oriented design in the first place.

The developed FSM is relatively compliant with the industry's approaches to game AI and relies heavily on the use of scripting in AI. Among the areas that can be improved, is the use of a proper XML parser for parsing the state transition XML table. The currently implemented parser is actually a text processor developed in collaboration with Ali Derweesh. It searches the XML for certain strings and patterns in order to populate the StateLogic list and with no error handling. This approach was adequate for the scope of this project because in most of the cases only one person was working on the AI behaviour which made it easy to spot any errors. However, in larger projects, a graphical bespoke AI-editing tool with XML can be more user friendly and less prone to errors (S. Pick, personal communication, 06 Sep. 2007)[12]

Throughout the game, many of the player-guard's interactions were handled through the AI. It is not the sword's collision with the agent's mesh that triggers the mesh cutting, it is the guard's AI testing if its in the player's range, the direction of the player and the animation of the player. Although this has helped in providing a rapid solution, along with avoiding the overhead of having the physics testing for collisions, it had its own problems and the focus here is on the SenseAttack. The problem was that the attack animation of the player's character starts by pulling the sword to the back and then swinging it. Now testing if the player is in the attack animation would return true from the first frame to the last one. This resulted in the guards getting cut as the player hits the attack button before the sward reached the guard. In an attempt to solve this problem, the normalized timing of the animation was tested to roughly specify the time when the sward would naturally collide with the guards mesh. For example after 40% of the animation time has elapsed. Unfortunately this solution did not work because the time values returned was incorrect for the first frame which made this approach useless. As an alternative, the developers decided to try to solve this in the cutting algorithm rather than in the AI. The new solution was to perform a short delay in the cutting component before preforming the actual cutting. The results of this last solution looked more realistic, however, this approach would be hard to work with if there were many different attack animations with different timings.

One very important feature the game could definitely benefit from is the implementation of a path finding algorithm. Although Instinct engine implements one, it is incomplete and is provided as an example. The effect of having a path finding feature can help make the guards look more intelligent and perhaps more challenging to the player. It would allow the guards to chase the player for longer distances and most importantly avoid running in walls and obstacles. In an attempt to work around this problem, the guards were able to sense how far away they were from their initial position. Guards would stop chasing if they're astray so that the chance of walking into obstacles is minimized after they return to their default state. The astray threshold of a guard was made accessible from within instinct studio for convenience and guards outdoors were given higher values compared to those indoors.

Turning to the NPCs behaviour, in a three-day event called Protoplay (12-14 August 2007), the developers had the chance to exhibit the game and watch people of the public playing the game and breaking it on some occasions. Doing so helped the developers put the theory to the test and practically identify what worked and what did not.

Among the comments some of the people had about the game AI behaviour, is that the guards did not pose a real challenge to the player. Watching people playing the game, it was noticed that in most of the cases, when a guard killed the player character, it was when the person controlling it did not know the controls very well. The combat AI could have possibly been improved by utilizing the block state in the behaviour of the guards which is currently available only for the player character. Of course that would also require more senses to be developed to help the AI guard identifying when an attack is imminent.

Also observing people playing the game, when the guard runs away from the player, players did not leave the guard and got on with the game. On the contrary, they chased the guard and made sure it was dead before they moved on. As a result, the behaviour of the guards trying to pounce the player was never experienced by people who played the game making it an increased overhead. That stresses the fact that designing a NPC behaviour which 'feels' right is a soft skill that requires experience.

5.6 Conclusion

This chapter has shown how efficient FSMs can be used in modelling NPCs behaviour. By no means can the work done on the AI be considered complete and it can benefit from a number of features namely a good path finding and perhaps upgrading it to be a FuSM.

In general, the AI system served its purpose. It helped emphasising the stealth gameplay elements of the game and people enjoyed sneaking up to guards and either cut them in half or stun them, an experience which was also stressed by the on-screen feedbacks.

The developed FSM has the following features:

- It is relatively easy and straight forward to add new senses and states to the system.
- It is easy for non-programmers to deploy new added senses and states in the behaviours of agents and also to change and tweak the behaviours of the agents.
- It allows for changing the whole behaviour of an agent during the running of the game
- It makes it possible to personalise the agents so that no two agents are identical.
- It -to a certain extent- adheres the industry practices by modelling the behaviour of an agent using the widely used XML files.

In conclusion, games development is quite a challenging field and game AI requires skill-sets of different backgrounds. Programming a game AI engine evolves a lot of problem solving in addition to research and design. In order for game AI engines to be most efficient it needs to enable the game designers to rapidly iterate and test behaviours of NPCs. Designing a NPC behaviour however requires an artistic background and experience as well. (Pick, S. personal communication, 06 Sep. 2007)[12].

Chapter 6

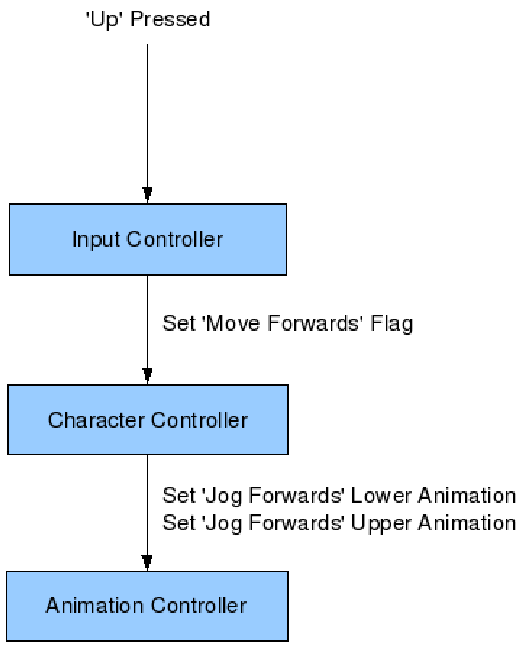
Input And Character Control, Audio/Visual Programming, Gameplay Engineering

By Ben Chandler

6.1 The User Input To Character Control Process

Control of the character is broken down into several stages. The first stage is receiving the input from the keyboard, mouse or gamepad. This input is handled by the input controller component which decides how a given input effects the game. For example when in game pressing 'up' on the controller will cause the character to run forwards, whereas pressing start will cause the main menu to appear. The input controller can influence more than just the player, it effects the whole scene and as such there is only one such component per game scene (level). In the case of a character input the input controller sets one or more state flags in the character controller depending what has been pressed. For example if 'a' is pressed then the 'jump' flag is set in the character controller to indicate the character is jumping. The third and final stage in the control process is the animation controller. The animation controller receives input from the character controller when it must change animation. For example, with the aforementioned jump state change, the jump flag set in the character controller would be passed on to the animation controller by setting it's own 'jump launch animation' flag. At this point the animation controller takes charge on the character and will transition through the launching, air, and landing jump animations. With that said it is still possible for the character controller to query the animation controller to find out what state it's in, this proves useful for things such as timing attacks.

Figure 6.1: Flow Diagram Of Player Being Moved Forwards



A final note before discussing the controllers in depth is that the actual implementation of the player's controller components and the AI's controller components is different. Initially they were using the same components, however as the project progressed there were various things that we wanted to do with the player character that meant using it's own version of the controllers. In addition to that the implementation of these components are in a modified version of the ieExample.dll file rather than in BC_Components.dll. This is because it was much faster to build upon the existing player controller than build another from scratch.

With the above brief summary out of the way I will now describe some of the specifics of the controllers.

6.1.1 Input Controller

The input controller itself was largely an extension of what was already supplied with the engine. That is to say, largely it came down to more controls needing to be mapped however I still had to implement some small changes such as the ability to invert the camera controls should the player prefer it. For the players input controller I also the ability to turn on and off the sound shader.

6.1.2 Character Controller

The character controller required a lot of new and in many places replaced code over what existed previously due to the large number of possible states that the character could be in. In addition to this many non-control related changes were made, such as syncing footstep sounds to the animations. I also supplied some helper functions here for use by the AI, for example getting the amount of light falling on the player or the amount of noise the player is making as well as the more simple functions such as determining how far through an attack the player is. The other task that the character controller is responsible for is that actual physical movement of the player within the world. Little had to be added in this area, although neither jumping nor sneaking were present in the supplied version of the controller and as such they had to be implemented by hand.

6.1.3 Animation Controller

The animation controller, much like the character controller, required not only an extension of what had been done before, but in many places, a rewriting of the code. It's biggest shortcoming for what we required was that it considered the character to have animations blended and applied to the whole character and would not take into account things like jumping and attacking without a messy blend between the two animations over the whole body. To combat this it was necessary to write a tool to split the animations into separate files for different portions of the body so that we could, for example, attack with the upper body while running forwards. Without having to have every possible combination of actions as an animation. The splitter takes a full body animation, such as running, and splits it above and below the hips, in both upper and lower animations the hip bone is present for the purpose of smooth blending across upper and lower body sections.

In order to use this technique much of the animation controller had to be rewritten in order to take advantage of the separated animations and blend upper and lower body separately. The upper and lower body states are managed by the animation controller itself and not, as one may expect, the character controller. This is because from the point of view of the character, the character only needs to know that they are running and attacking, not that their lower body is running and upper body is swinging a sword. Technically this tidies up the code, since when the AI is controlling a character for example, it can tell the character to run without having to tell it's upper body and lower body both to run, which would involve all of the subsequent 'can I run?' type state check on the animations. One way to look at it would be the character controller tells the animation controller what it would 'like to do', the animation controller then sets the animations correctly. For example the character controller would pass on that it wants to run and attack, the animation controller would realise that that required the lower body to run, while the upper body performed the attack, none of which is worried about by the character controller.

6.2 Sound

6.2.1 Background Music

The manager the background music transitions the Ambient Sound Manager component was created. When making the ambient sound manager, I used 2D sounds as build into the engine for the music tracks and attached them to my own manager. The manager controls them and mixed them based on flags, for example 'player seen' and 'player attacked'. These flags are in turn controlled by the AI. The blending into and out of tracks itself is logarithmic as opposed to linear. This means that it is perceived volume (loudness) that is linear (roughly), rather than actual volume. This is to counterbalance the non-linear volume of a sound that the human ear hears and gives a smoother transition. Technically this is not quite true since human hearing also varies with frequency, that is to say, it is likely that a 1000Hz sound at 1dB will have a different percieved intensity to a 2000Hz sound at 1dB, despite them both having the same loudness. However for the puropose of the game this level of accuracy was unnecessary.

6.2.2 3D Sound Sources

3D sound sources within the game are normally part of a larger entity. In almost all cases they have a sound shader light and controller attached to be visualisable. Before diving into the topic of shaders and the creation of the sound shader I will first discuss one very important property of the 3D sound source component provided and that is the 'destroyOnFinish' property. This means that when sounds such as footsteps are created they can be automatically destroyed if this flag is set, thereby making managing memory for such objects trivial. This also allows automatic destruction of any objects attached to the sound which means that a lifetime is not needed to be specified for the shader since it is (optionally) destroyed when the sound is.

6.3 3D Graphics

6.3.1 Shaders

Looking at the visual side of things, I will begin with a quick introduction to the shader system. Shaders in Instinct are written using either HLSL or assembly language, both of which I learnt for the purpose of understanding the existing shaders that were build into the engine. By way of strengthening my understanding I began by writing a simple water shader in HLSL. Once it was working I then went on to try and tackle (what I thought would be) the harder task of writing the sound shader.

Listing 11 Example Of An Assembly Language Shader

```
pixelShader BaseLightDiffuse_1.1
{
   ps_1_1
   def c1, 1, 1, 1, 1
   tex t0 ; light projected
   tex t1 ; diffuse
   tex t2 ; normal
   tex t3 ; L (cube normal map)
   dp3_sat r1, t3_bx2, t2_bx2 ; N . L
   mul r1, r1, t1 ; ... * diffuse
   mul r1, r1, c0 ; ... * light color
   mul_x2 t0, c1, t0 ; light projection x2
   mul r0, r1, t0 ; ... * light projection
}
```

Listing 12 Example Of A Similar HLSL Shader

```
pixelShader
{
   #include "base/materials/BaseHLSL.material"
   // Constants
   sampler2D lightProjectionMap : register(s0);
   sampler2D lightFalloffMap : register(s1);
   sampler2D diffuseMap : register(s2);
   sampler2D normalMap : register(s3);
   sampler2D specularMap : register(s4);
   float4 lightColor;
   float specularPower;
   COLOR main(in BASE_LIGHT_PS In)
   {
      COLOR c;
      float3 light_map = tex2Dproj(lightProjectionMap, In.lightProjectionUV)
* 2.0f;
      float3 falloff_map = tex2Dproj(lightFalloffMap, In.lightFalloffUV);
      float3 diffuse = tex2D(diffuseMap, In.diffuseUV);
      float3 normal = tex2D(normalMap, In.normalUV) * 2.0f - 1.0f;
      float3 L = normalize(In.L);
      float3 bump = clamp(dot(normal, L), 0.0f, 1.0f);
      float3 V = normalize(In.V);
      float3 H = normalize(L + V);
      float3 specular = clamp(dot(normal, H), 0.0f, 1.0f);
      specular = pow(specular, specularPower.x) * 2.0f;
      specular = specular * tex2D(specularMap, In.specularUV);
      c.color.rgb = (bump * diffuse + specular) * light_map * falloff_map *
lightColor;
      c.color.a = 1.0f;
      return c;
   }
}
```

6.3.1.1 Shaders and Materials Within Instinct

The shader system within Instinct is intimately related to the material system, as touched upon in 3.3.5, used to create materials to texture scene meshes with. Indeed shaders are located at a deeper level of the inheritance provided for material files. If parent material files were traced sufficiently far back eventually a shader would be encountered. However the inheritance hides much of this and when defining a material such as in figure 3.17 the existence of the shader is almost hidden other than for saying which parent material to use.

Going into more depth now, material files essentially comprise of the following structure [5]

Listing 13 Material File Structure

```
BaseMaterialName
{
   States
   {
      Blending, depth and culling options for the surface are defined here
   }
   pass 0
   {
      Class
      {
         Shader language and required hardware specified for pass here
      }
      Texture TexName1
      {
         Texture flags set, for example number of frames for TexName1
      }
      Texture TexName2
      {
         Texture flags set, for example number of frames for TexName2
      }
      ...
      VertexDeclaration
      {
         Vertex shader inputs are defined here, for example position
      }
      VertexShaderConstants
      {
         Define vertex shader constants here
      }
      VertexShader
      {
         Define vertex shader here
      }
      PixelShaderConstants
      {
         Define pixel shader constants here
      }
      PixelShader
      {
         Define pixel shader here
      }
   }
   Pass 1
   {
      ...
   }
   ...
}
```

With the above material defined we can now override it and give the shader a new texture, as was shown in the reference given immediately above, using the following material definition.

Listing 14 Overridden Material Example

```
NewMaterialName : BaseMaterialName
{
   TextureAliases
   {
      Here we can now override the default textures by adding for example:
      TexName1 = NewTextureName
   }
}
```

It is also possible to do more complicated inheritance within Instinct, for example a state could be overridden in the following way.

Listing 15 Overridden State Example

```
NewMaterialName2 : BaseMaterialName
{
   States
   {
      cullMode = none
   }
}
```

This would serve the purpose of turning off back face culling on any surface to which the material was applied. In addition to overriding material elements in this fashion, it is also possible to define commonly used elements outside of a material and parent the material's element to it. For example

Listing 16 Externally Defined Material Elements

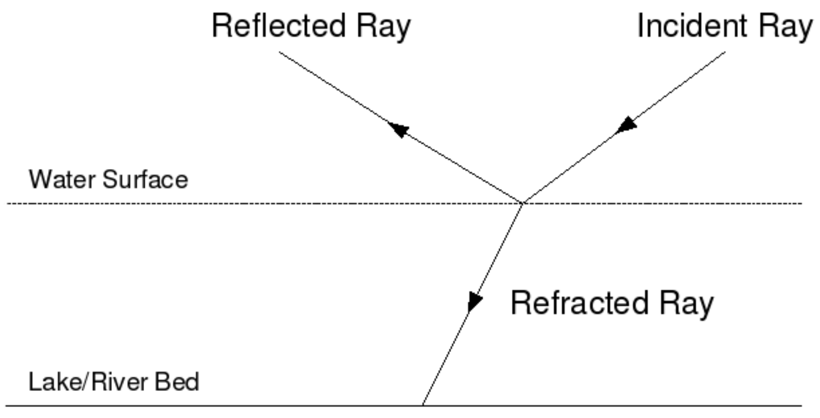
```
States CommonState
{
   cullMode = none
}

NewMaterialName3 : BaseMaterialName
{
   States : CommonState {}
}
NewMaterialName4 : BaseMaterialName2
{
   States : CommonState {}
}
```

Of course this is not just restricted to states, in fact it is possible to use most pieces of a material in this way. Elements all the way from the shaders up to the passes can be inherited and overridden in this fashion which gives a great deal of code reuseability.

6.3.2 Water Shader

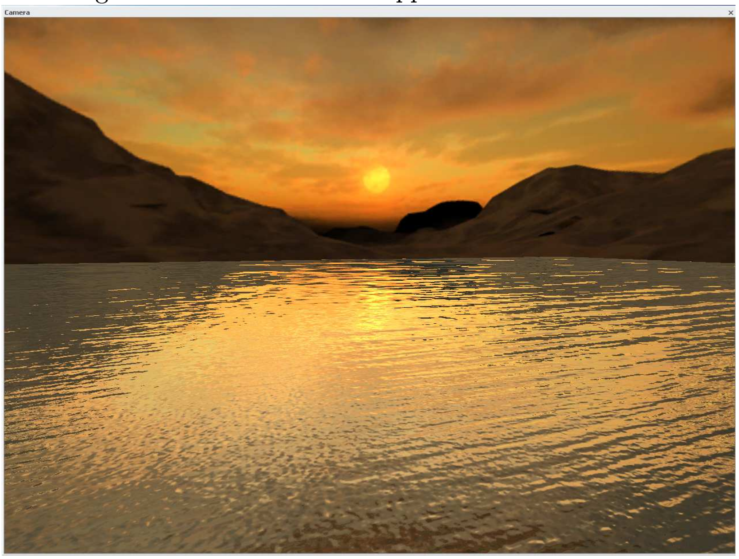
The water shader is calculated by using the incident eye ray, per fragment, to calculate the reflection and refraction rays based upon the surface's normal map at that point. The reflection ray is used for an environment map lookup (the sky box) and the refraction ray traverses a fixed distance under the surface before doing a 2D texture lookup for the lake/river bed.

Figure 6.2: Water Shader Calculations



The final pixel colour is then calculated by blending the two ray contributions taking into account the Fresnel term. The shader was also capable of accepting animated and scrolling textures. For example in the river a non-animated version is used which scrolls, in the lake and pond an animated non-scrolling version is used.

Figure 6.3: Water Shader Applied To A Test Scene



6.3.3 Sound Shader

The problem of representing sound through vision has many different approaches that could be used. When implementing the sound visualisation I found that some would be more convenient to apply than others due to the architecture of the game engine.

6.3.3.1 Initial Approach

When starting out the initial approach was to create a shader that would be applied to every surface, the result of which would be the intensity of the incoming 'light' from the sound source. Formally, if

x is the point in space being lit

x_l is the position of the sound source 'lighting' x

t is the time since the sound's creation time

I is the intensity of the incident light at x from an individual sound source

then the distance between them is $d = |x - x_l|$

It is at this point that I diverge from the traditional diffuse lighting model in that I calculate the intensity of incident light in the following manner:

Firstly define $T(d)$ as the time taken for the sound to travel a distance d, typically this will be linear.

Then

If the sound has reached the point, i.e. $t > T(d)$

$I = f(t - T(d))$, where $f(t)$ is the intensity of the sound's waveform at time t after creation

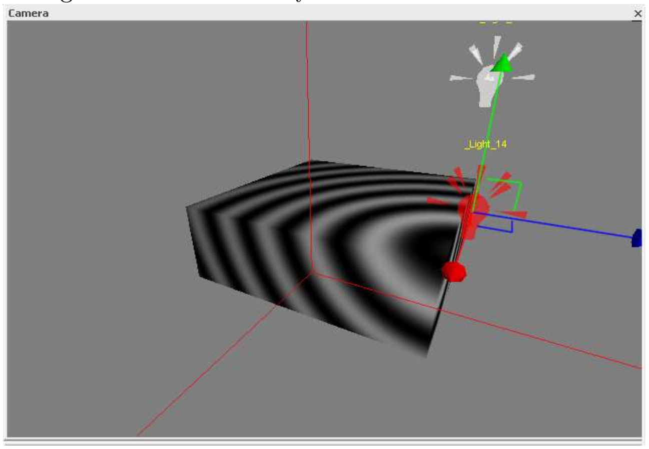
Else

$I = 0$

Once the sound has played out the sound source (and with it the light source) is destroyed. As such the waveform, due to it being finite in duration, can either be represented in it's 'mathematical' form in the shader (e.g.,$f(t)=\frac{sin(t)}{t+1}$), or in order to save computation can be pre-calculated and used as a 1D texture lookup. The approach can be generalised by the use of environment maps to take into account periodic waveforms.

In it's favour this approach is highly customisable and (fairly) accurately represents the 'feel' of the wave being emitted (e.g. A spike in intensity of the sound will show up visually). Unfortunately this approach is very hard to implement given the time constraints based on the architecture of the engine [6]. While I did get a version of this working with only a single sound source, generalising it to many sources would have proved too time consuming to implementing and in all likelihood too expensive computationally.

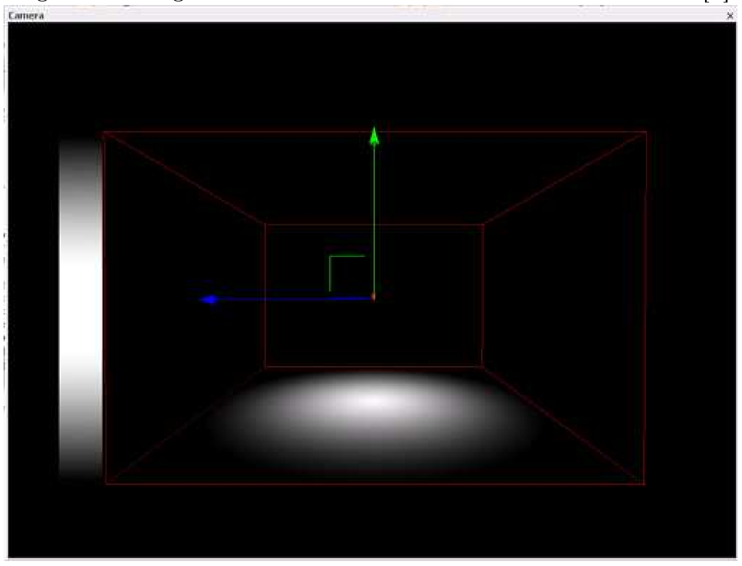
Figure 6.4: Preliminary Sound Shader Screenshot



6.3.3.2 Chosen Approach

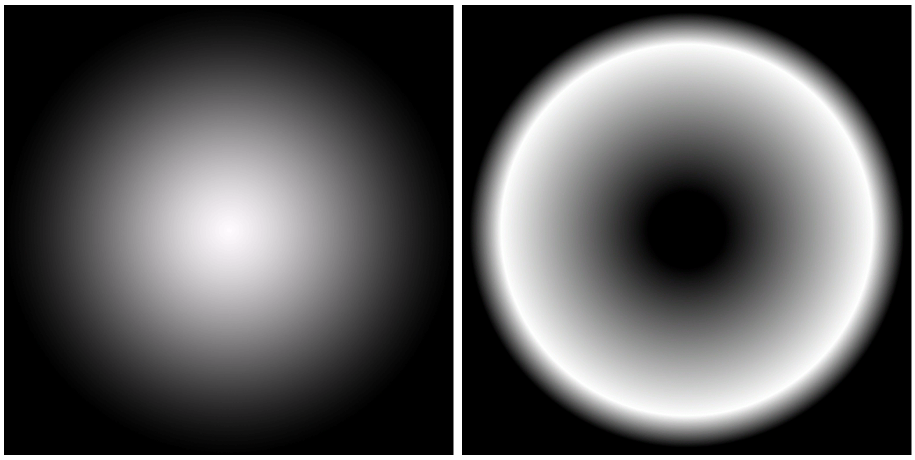
The final approach we decided upon is a compromise between artistic controllability and ease of implementation. It relies on the built-in functionality of the engine's default light sources, to which I attach a component to control them as I wish. The built-in 'box' light sources accept an attenuation texture which is used to specify the intensity in the horizontal plane based upon the distance from the light source position. They also make use of a falloff texture which is used for the vertical falloff. In the case of spotlights these textures are handled differently but since these are not used in-game I will not discuss them here. The box light itself is defined by a point together with a box specifying how far it can effect over which the textures are scaled in their respective faces. The lighting intensity is then calculated for a point within the box by looking up the intensity in both the attenuation and falloff textures (by projecting the point into the plane and line respectively). These looked up values are then multiplied to give the final intensity. For clarification see figure 6.5 below.

Figure 6.5: Light Attenuation Box and Falloff Taken From [5]



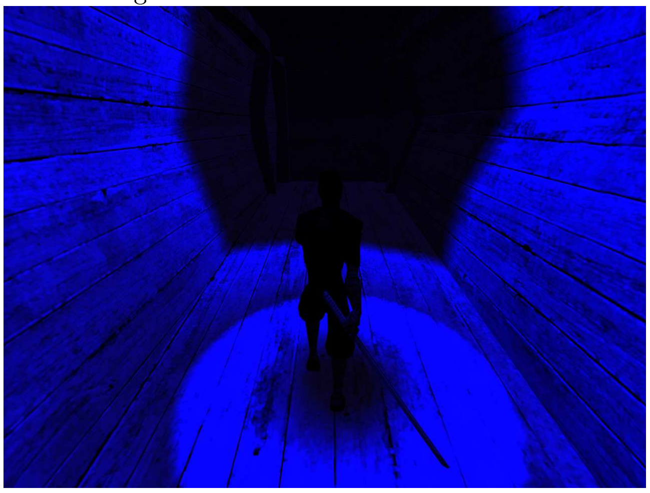
I modify this for the sound shader by firstly changing the texture for a halo type texture

Figure 6.6: Original Light Texture Compared To New Light Texture


(a) Original Attenuation Texture (b) Modified Attenuation Texture

The new component then takes care of the rest by calculating how long the sound source has been 'alive' for and scaling the attenuation box, as well as intensity of the light source by linearly interpolating the values based on the age between start and end times for the sound. This is not strictly correct in the sense that sound waves do not physically behave like this, but visually it gives an acceptable effect without too much of an impact to the hardware unlike some of the previous techniques. The final effect in-game is as below in figure 6.7.

Figure 6.7: Sound Shader Screenshot



6.3.4 Skydome

The skydome was created using the skylab in Bryce 5. I rendered out the sky from six different directions in order to form an environment map which was then put together in Photoshop. Once constructed I exported it using the Nvidia DXT exporter. Once it was in DXT format I wrote a shader to apply the texture as an environment map onto the skydome mesh.

6.3.5 2D Unlit Shaders

For some surfaces such as loading screens, overlays and the HUD lighting was not important. For these cases a collection of shaders was written to deal with several of the specific cases and different blend modes required. For example the HUD background required 8-bit transparencies, whereas other overlays required 1-bit while some required no transparancy at all.

6.4 2D Graphics

2D graphics in the game were handled by creating a quad based upon the aspect ratio of the screen. The quad's transform was then parented to the current camera and it's transform set so that it was a fixed distance away. Parenting it meant that it would remain fixed in front of the camera during gameplay. It was using this technique that all 2D graphics were drawn to the screen.

6.4.1 HUD

In the case of the HUD, a template was made consisting of several pieces. Below is an outline of the template [Listing 17]:

Listing 17 HUD Entity Template

```
EntityTemplate
{
   _name = "BC_Package/HUD"
   _description = "HUD Entity"
   _components = "Transform:Transform,
              HealthTransform:Transform,
              LightTransform:Transform,
              SoundTransform:Transform,
              HUDManager:HUDManager,
              Mesh1HealthBar:Mesh,
              Mesh2BG:Mesh,
              Mesh3LightBar:Mesh,
              Mesh4SoundBar:Mesh"
  Mesh1HealthBar.MeshFilename = "MO_Package/models/HUDquadhealth.mesh"
  Mesh1HealthBar.SmmFilename = "MO_Package/models/HUDquadhealth1.smm"
   Mesh1HealthBar.TransformName= "@this.HealthTransform"
   HealthTransform.ParentTransform = "@this.Transform"
   ...
}
```

Each HUD element, for example the health bar, has a mesh and a transform associated with it to control it's position on screen. The health level, light level and sound level bars are all parented to the background transform, which in turn is parented to the camera. Additionally there is the HUDManager component which controls how the transforms are updated. Each frame it acquires, for example, the current health of the player and updates the respective transform in the entity.

6.4.2 Menus and Loading/Ending Screens

The main menu is acheived using a 3D rendered scene for the background. The flashing text that appears on the start menu (saying "press A") was acheived by using a HUD entity. While the full functionality of the entity type was not needed, for the text, it sufficed to set the background transform and mesh up. The other transforms were then set to be off the screen and as such only the text remained. The flashing text that appears was acheived by putting an animated texture on the quad used for text.

The loading and ending screens we acheived by swapping the HUD background (the sprite that 'held' the status bars) with the respective screen background. Additionally the other elements were moved off screen all of which was acheived via a collection of scripts.

6.4.3 Pickup Notification

Pickup notifications refer to the text that flashes up on the screen during gameplay to inform the player of events such as a 'stealth kill' or 'artifact collected'. The implementation was similar to that of the HUD entity, however due to the way in which animated textures work in the engine it was not suitable to simply apply an animated texture to the screen quad. This is because there was no way of telling it when the animation should start so if attempted this way the start frame of the animation would not necessarily be the one intended. To get around this materials were made for each frame separately and a PickUpNotifier component was written in order to swap over these materials and eventually kill the text once it had been displayed.

6.5 Gameplay & Scripting

Instinct supports Lua scripting which we used for many of the scripted gameplay elements since often it is quicker to add and modify than the equivalent functionality in a piece of code. The other invaluable component supplied is that of the 'trigger block'. Trigger blocks define a box in space such that when a particular object or selection of objects passes through them they trigger a script to be run. Linking the trigger blocks to the scripts allows for various gameplay techniques to be implemented, for example, menu/end screen transitions, object pickups and powerups. I provide more detail and examples below.

6.5.1 Score Manager

One component that had to be written from scratch was the score manager. The score manager was a simple device which (unsurprisingly) kept track of the various elements used to produce the score. For example, had the player been seen by guards, had they been heard, number of guards killed, etc. These various variables were updated mainly by the character controller & AI during the course of the game. The final score is then calculated by a function built into the class which outputs a file detailing the players performance during the level which is displayed upon completion of the level.

6.5.2 Menu Transitions

The menu transitions work differently depending of where in the game the transition is occuring. If the player is in the main menu or loading screen then due to lack of a player, there is no player input controller. However the engine comes with what is called a command mapper. This takes input from the keyboard, mouse or gamepad and can be used to perform scene transitions by mapping the appropriate input to a script call to change scene.

In game, things work a little differently. Rather than use the comand mapper for input, the player input controller is used instead. In essence there is little difference between what actually happens to change scene. A script is called in both cases, however in the case of the player input controller, it is called from within the code. In the case of the command mapper, it is a script calling another script.

6.5.3 End Screen Transition

Unline the menu transitions, the end screen displaying is an automated process whenever the player gets within range of their mission target. The end screen is activated by a trigger block around the destination that the player had to get to. This then runs a script telling the HUD manager to put the 'mission success' screen over the background as well as calling a function in the score manager to output the statistics of the players performance to a file. This file was then read in and displayed on the screen.

6.5.4 Pickups

The pickups are derived from a base pickup template the important lines of which are listed below [Listing 18]

Listing 18 Base Pickup Template

```
EntityTemplate
{
   _name = "BC_Package/PickUpBase"
   _description = "A base template for pickups."
   _parents = "example/selector/TriggerVolume"
   _components = "TriggerCondition:VolumeTriggerCondition,
                  Trigger:Trigger,
                  Transform:Transform,
                  Shape:BoxShape,
                  PickUpMesh:Mesh"
   Trigger.scriptFile = "BC_Package/scripts/PickUpTrigger.lua"
   ...
}
```

For illustrative purposes I also give an example of how the health pickup inherits this [Listing 19]

Listing 19 Health Pickup Template

```
EntityTemplate
{
   ._name = "BC_Package/PickUpHealth"
   ._description = "A health pickup."
   ._parents = "BC_Package/PickUpBase"
   ._components = "PickUpHealth:PickUpHealth"
   ._completeEntity = yes
  .PickUpMesh.meshFileName = "MO_Package/models/selectorHealth.mesh"
   .PickUpMesh.smmFileName = "MO_Package/models/selectorHealth.smm"
}
```

The pickups, much like the end screen transition, each use a trigger block as part of a pickup template in order to detect for proximity of the player. When the player triggers the block, a script is run, which in turn runs a member function of the pickup entity component within the entity. This member function performs the pickup-specific functionality, for example adding health, followed by spawning of any feedback entities and finally destroying itself.

6.5.5 Footstep Toggling

In order to change the footstep sounds over various surfaces trigger blocks were once again employed. This time the character controller had a member variable added which contained the name of the template that would be used to create the footstep entity. When the animation triggered the footstep entity to be created it would use this template for it. Similarly a template was specified for the particle system to use for any dust or splashing particles to create on a footstep. Each of these variables was changed by a script called when the player entered/left an area (trigger block). For example going from a stone floor indoors to grass outdoors changed the footstep sound template from a stone sounding footstep to a grass one. It should be noted that these footstep templates aren't just sound sources, the templates also contain a sound shader light source and a sound shader controller for managing the sound shader light.

6.5.6 Floorboards

The creaky floorboard was simply created using a trigger volume which called a script to create a creak sound. Like the footsteps it also had a light source and sound shader manager as part of the entity. The entity itself cound be used for a more general group of effects in which the player triggers a sound, it is not specific to the creaking sound. It could even be used for fairly complicated effects such a water dripping on the floor, which if you stand underneath it it stops. Sadly due to time constraints this did not get into the demo in time.

6.6 Particle Effects

Instinct supports a very powerful particle effects system with many specifyable properties. The properties that can be specified fall under three main categories. These are

- Emitter
- Behaviours
- Graphics

The emitter section allows the designer make choices about the shape and extents of the object that the particles are emitted from as well as the rate at which they are emitted. The behaviours section controls parameters such as the forces to be applied as well as any colour transitions that should occur over the particle's life. In the graphics section one can change the orientation of particles with respect to the camera as well as any material properties they wish to specify beyond the simple transitions specified in the behaviours section.

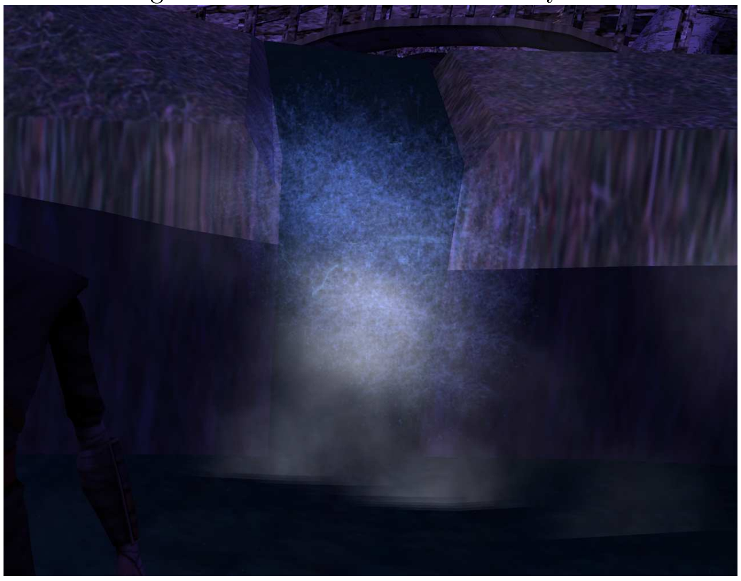
Particle effects were used extensively in the game, some of the uses of which I detail below.

6.6.1 Waterfall

The waterfall particle system was used in the garden at the beginning of the level where the stream from the mountains flows into the pond. It uses two particle systems in order to create the effect, one for the falling water droplets from the top of the fall and one for the mist generated at the bottom of the fall. Each of these systems has a different sprite associated with it as visible in the screenshot below.

The emitter volume for each system was a long thin box alligned with the edge of the waterfall. One at the top and one at the bottom. In the case of the droplets they were given no initial forces and left to simply fall under gravity. With the mist gravity was ignored and the initial forces were random upto a certain maximum force.

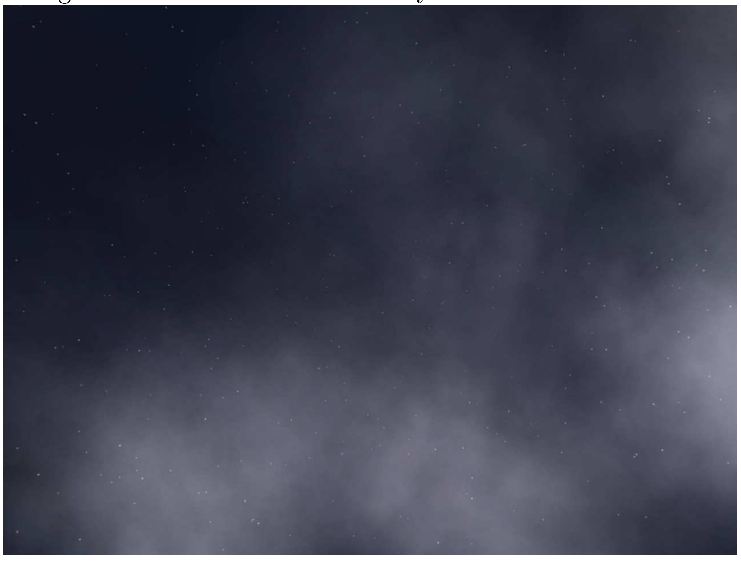
Figure 6.8: The Waterfall Particle System



6.6.2 Clouds

The cloud particle system, as one would expect, is positioned in the sky. Additionally however it is used for the puropse of mist over the lake. The emitter volumes for these are, as with the waterfall, long thin boxes. The colud emitter is positioned just behind the mountain, the other under the jetty. The system itself is the simplest used in the game and consists of only one sprite. Due to the slow nature of the cloud movement it was possible to use very few particles to cover the sky provided they were sufficiently large and had quite a long lifetime. In-game there are approximately 150 particles used at any one time on the clouds and another 150 for the mist. The way in which the clouds work is by additively blending the sprite over the top of the skydome background using a very low alpha value.

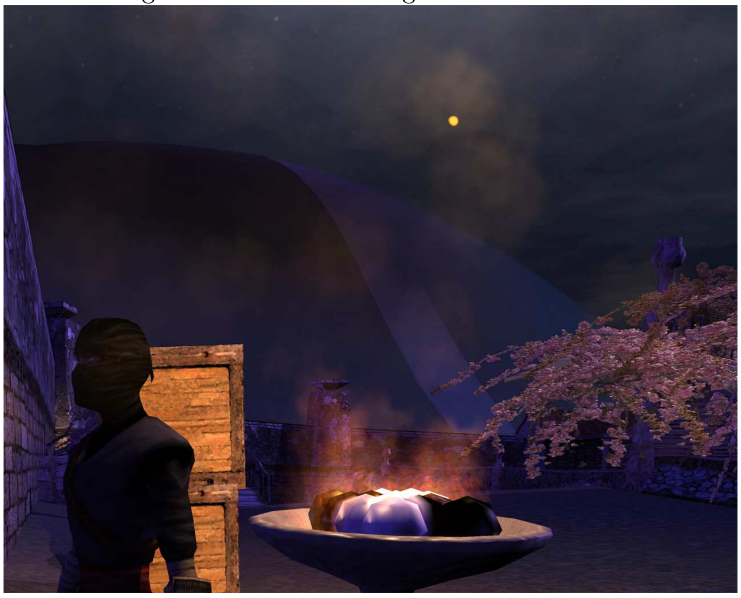
Figure 6.9: The Cloud Particle System Within The Game



6.6.3 Torches

The flaming torches have different implementations depending on which torch type it is. In the case of wall torches 6 particle types were used, in the case of the freestanding torches 7 types were used. The torches are the most expensive particle systems in the game each one coming in at around 400-800 particles. In the case of the freestanding torches an additional effect I added was to apply refraction when looking through the flame to what was behind, as if the heat was making the torch's background shimmer. This was acheived by creating a single billboarded quad within the flames and applying a refraction shader onto it which refracted the framebuffer.

Figure 6.10: Freestanding Torch Screenshot



6.7 Tools

During the course of making the game I wrote several helper functions, scripts and applications just to make the process flow more smoothly. The functions themselves were added to BC_Components.dll and the header was supplied to the other packages for use in external code.

6.7.1 Raycasting Functions

One set of functions that proved useful was the ability to perform raycasts which I mainly used to determine the correct position for the camera so that it did not go behind walls. Instinct supports raycasts as part of the build in physics, however these functions made it simply one line to perform various ray queries, rather than numerous preparatory ones. One thing we had to be careful of was the use of the various physics objects because due to the raycast being a physics query it collided with the collision mesh for an object which in many cases was not it's actual mesh. More problematically the trigger volumes used for many things as discussed in 6.5 counted as as collidable geometry. Therefore walking through a door for example which had a trigger volume to change footstep template could force the camera forwards since it couldn't 'see' the player due to the volume being in the way. This issue was got around by the use of 'collision masks' which allow the designer to select which types of items the ray can collide with.

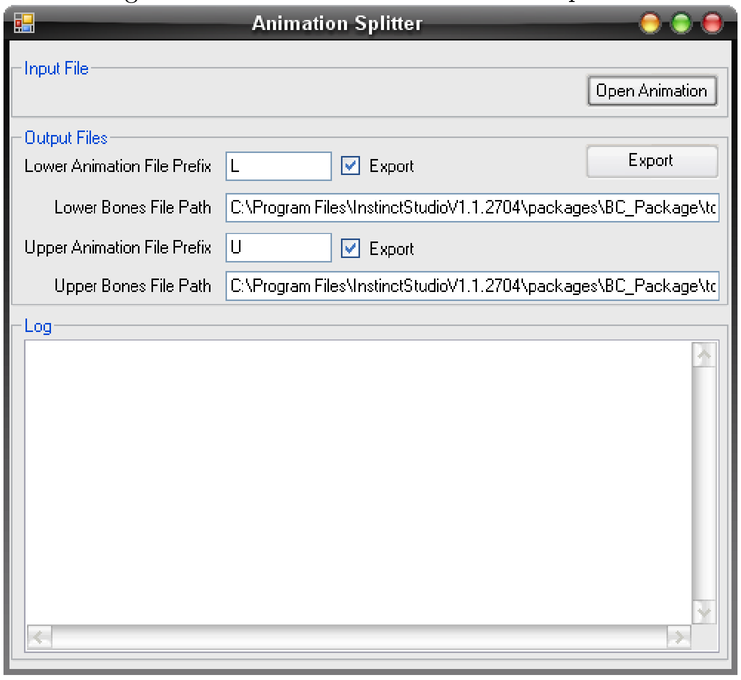
6.7.2 String Conversion Functions

Instinct has it's own string format, but often we would want to use STL strings, for example. Therefore I wrote various conversion functions both to and from their format and STL.

6.7.3 Animation Splitting Tool

In order to use the animations with our customised animation system, the animations needed splitting first. I created a GUI driven tool written in Visual Basic for use by the artists to split up the animations from Instinct's own format into two animations, still in the correct format for Instinct, but separated into upper and lower body files. This was acheived by supplying the tool with two files, each listing the bone names that would be searched for and kept, one for the upper body and one for the lower body.

Figure 6.11: Screenshot Of Animation Splitter



Lower Animation File Prefix: L ✓ Export
Lower Bones File Path: C:\Program Files\InstinctStudioV1.1.2704\packages\BC_Package\tc
Upper Animation File Prefix: U ✓ Export
Upper Bones File Path: C:\Program Files\InstinctStudioV1.1.2704\packages\BC_Package\tc

6.7.4 Code Writing Tool

During the writing of the character controller it became necessary to write a tool to automate some of the repetitive code writing. The reason for this was due to the large number of animations that needed to be set. In terms of setting and getting anamations there were 96 functions to write, each function being approximately 25 lines long. While it would have been possible by hand the functions themselves were not trivial to write and various uses of the functions were required in the code. For example the definition and declarations for the functions were in the same file, however the functions needed defining in the class, so a seperate list of predeclared functions had to be written to go there. Additionally interface functions had to be written so that the various scripting processes within Instinct could function. This meant that animations alone ran into literally hundreds of functions. For that reason I implemented a method of automating the code writing whereby the user could specify a template for the code, together with what varied from function to function. The tool could be run and code was output, based on this template and the inputs for the variables, for example function name.

6.7.5 Package Synchronisation Scripts

I wrote the scripts we used to synchronise our work over the network. They consisted of scripts to upload a newer version of our own work to a central server, as well as download the latest (uploaded) version of everyone else's work. The scripts were simple batch files that relied on using the command prompt (cmd) commands.

Chapter 7

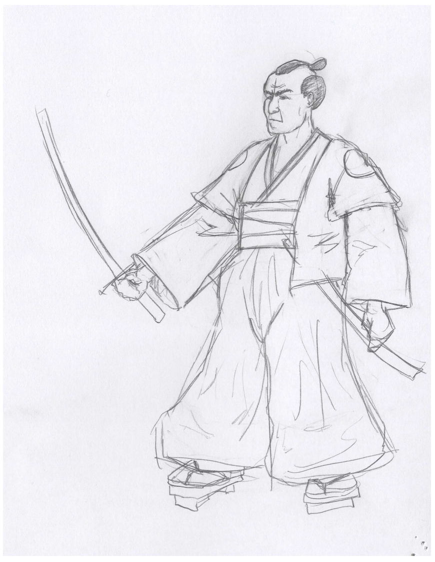
Characters and Animations

By Seb Huart / Matt Osbond

7.1 Character Design

The enemy characters were based on sketches of Imperial Guards from the Edo period of Japanese rule. This gave the characters a more stylistic feel, with elements such a large sword, baggy trousers and tied back hair combining to give an artistic look.

Figure 7.1: Design Sketch

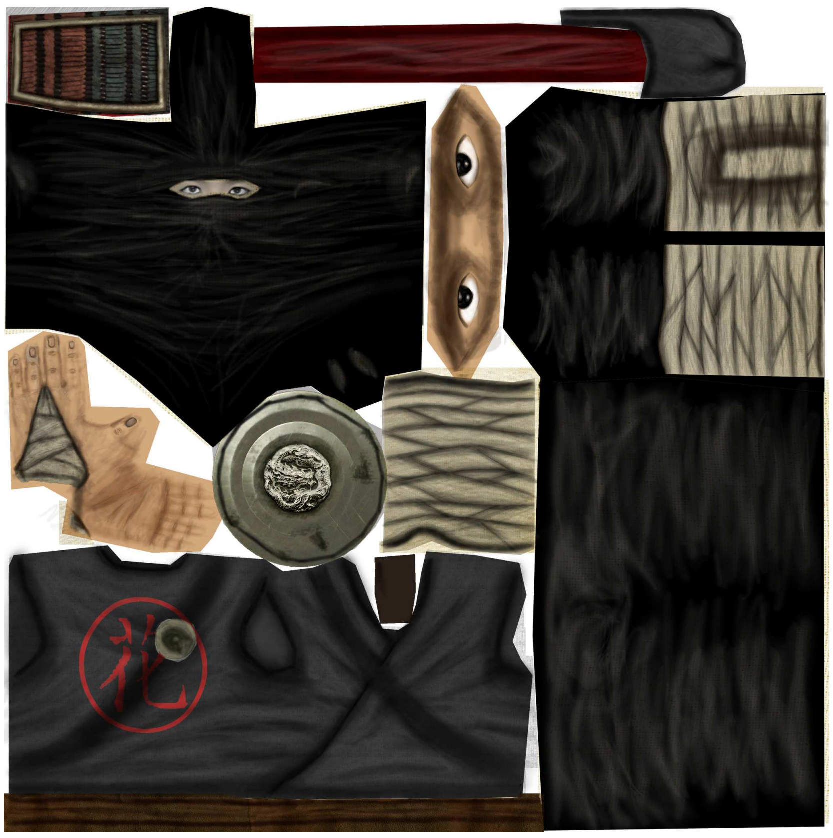


(a) All major assets in the game began as a 2D design sketch.

7.2 Texturing

It was important to allow the textures of the characters to be considerably high resolution. This was especially the case with the main character (the ninja) as it would be permanently within the players view, not to mention close up.

Figure 7.2: Ninja Texture Map

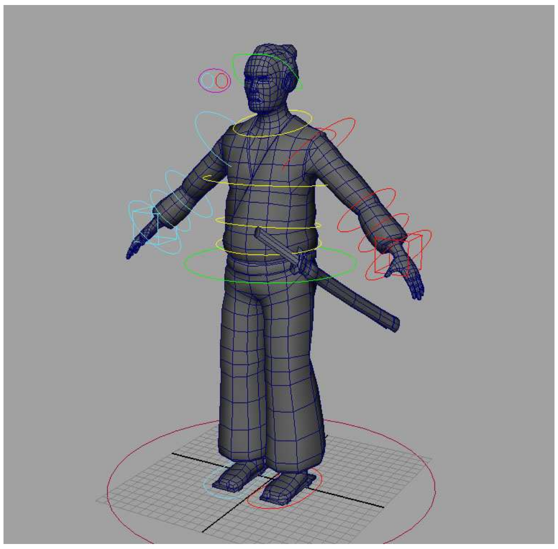


(a) The texture map was created from one alrge UV output.

7.3 Animation Cycles

The complicated nature of the movements involved with a stealth game meant that the rigs for the characters had to be versatile. The cycles were created as a small loopable animation clip that were then belnded together using the character controller.

Figure 7.3: Guard Animation Rig



(a) The rig was developed to allow for a greater freedom of movement.

Chapter 8

Feedback and Critical Analysis

8.1 Feedback Sheet - Protoplay

At the finale of the Protoplay event, the teams were presented with a sheet of feedback. This was a collated list of responses from the judges in the competition:

- Should have spent more time refining fewer features
- Looked somewhat unfinished
- Gameplay simple but not particularly unique
- Lot of potential with further development
- Sound detection visually interesting
- Dissection technically impressive
- Nice character profiles. Clear to see what was happening in game world
- Seemed to be trapped within graphic styles
- Nice ideas - good job of executing difficult plan
- Technology good but let down by being typical ninja game
- Ambitious project but don't think they focussed on what they wanted to do very well

- Left me with unmatched expectations
- Good backgrounds, sonar nice but not different enough, story well thought out
- Public vote - bottom third of table

The general consensus is that there is nothing wrong with the game, it's just that it was a) not polished enough, and b) not original enough.

These are two valid points, but unfortunately it is too late to do much about the second one. However, we were given the opportunity to focus on polishing the game, and as such have done so for submission.

8.2 Conclusion

The game presented us with numerous challenges along the way, from small hurdles such as textures not mapping correctly to game breaking engine troubles. The ability of the team to overcome each of these challenges is a testament to both their determination and their team spirit. Communication amongst members was critical to success.

The final product is of a standard that many judges and professionals confirmed was closer to industry level than any other production at the event; on one occasion a senior member of a respected company mentioned that if they were to ask their programmers to develop the features of the game within the same time frame then they would struggle to do so. The main drawbacks seemed to be, as aforementioned, the unoriginal idea and the lack of polishing. These elements, in fairness, were not our primary area of focus (impressive visuals and technology were), but knowing that a simple element such as last minute iteration let us down is a lesson we will all take away with us.

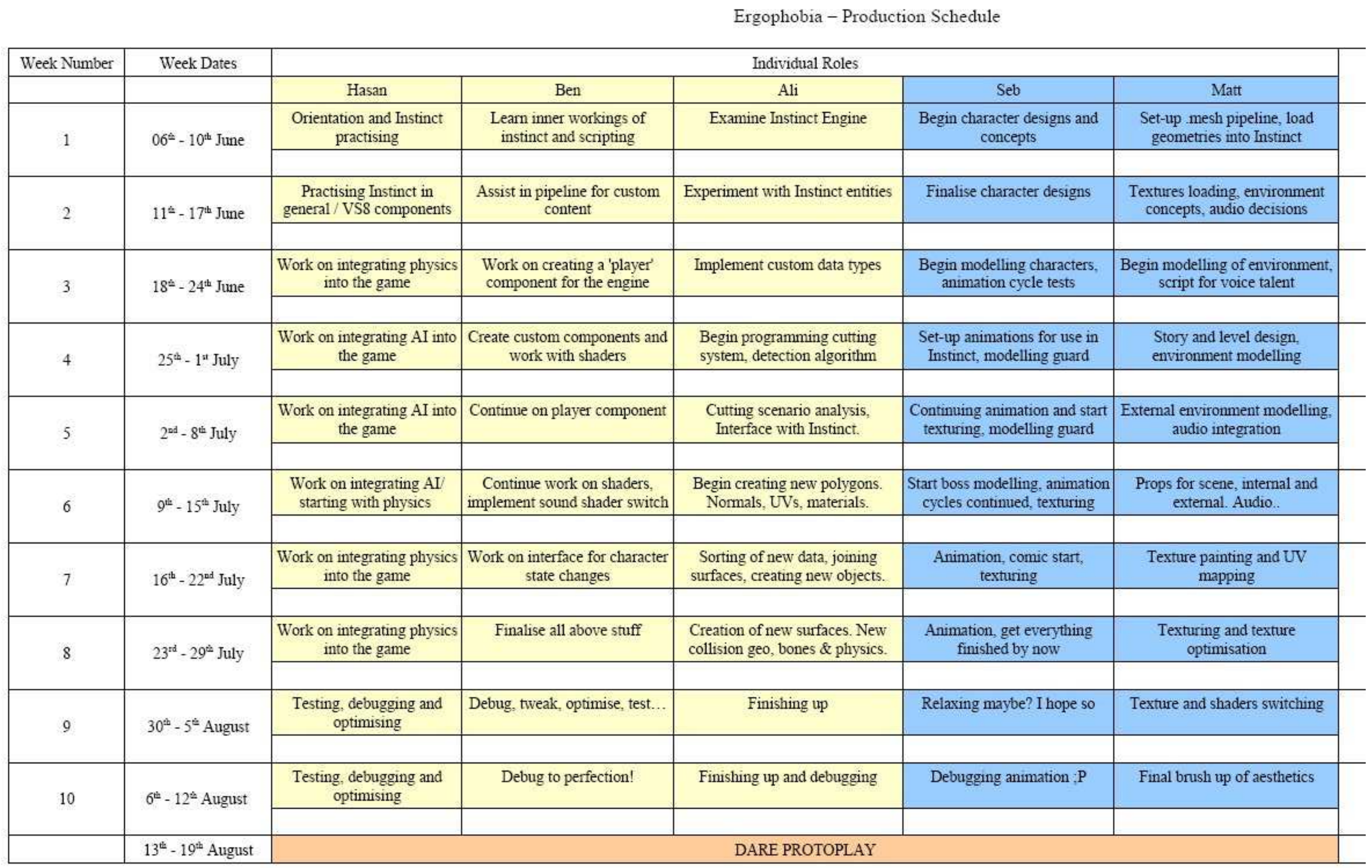
Although it is of course disappointing that we did not make it through to the final round, the experience of working within a professional environment at industry level is invaluable. Likewise, successfully seeing an entire production through from concept to realisation is valuable to us all in terms of experience and employability. The final game is of a quality that surpasses any expectations that were held at the beginning of the production. This is due partly to the pressure that was placed on us via the submission of the project for not only the competition, but also our dissertation. However, the main factor is that the production was a steep learning curve for us, as we all left at the end with a far greater knowledge of not only our subject areas but also the game production industry in general.

Appendix A

Scheduling and Project Management

A.1 Initial Schedule

Figure A.1: The Team's Schedule

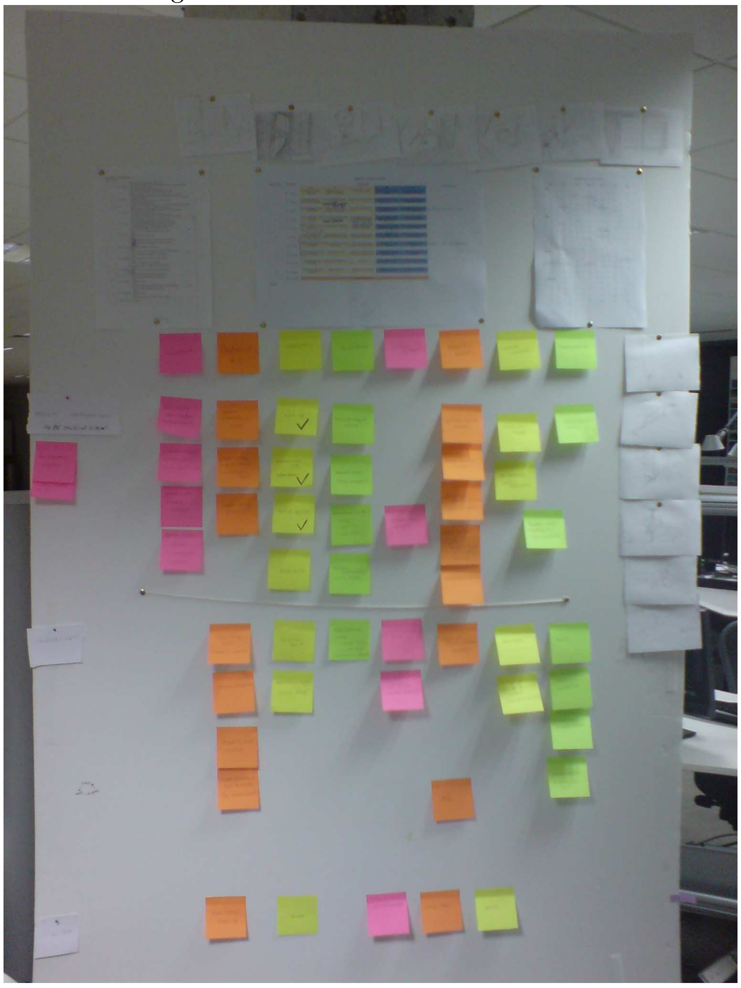


Ergophobia – Production Schedule

Week Number	Week Dates	Individual Roles				
		Hasan	Ben	Ali	Seb	Matt
1	06th - 10th June	Orientation and Instinct practising	Learn inner workings of instinct and scripting	Examine Instinct Engine	Begin character designs and concepts	Set-up .mesh pipeline, load geometries into Instinct
2	11th - 17th June	Practising Instinct in general / VS8 components	Assist in pipeline for custom content	Experiment with Instinct entities	Finalise character designs	Textures loading, environment concepts, audio decisions
3	18th - 24th June	Work on integrating physics into the game	Work on creating a 'player' component for the engine	Implement custom data types	Begin modelling characters, animation cycle tests	Begin modelling of environment, script for voice talent
4	25th - 1st July	Work on integrating AI into the game	Create custom components and work with shaders	Begin programming cutting system, detection algorithm	Set-up animations for use in Instinct, modelling guard	Story and level design, environment modelling
5	2nd - 8th July	Work on integrating AI into the game	Continue on player component	Cutting scenario analysis, Interface with Instinct.	Continuing animation and start texturing, modelling guard	External environment modelling, audio integration
6	9th - 15th July	Work on integrating AI/ starting with physics	Continue work on shaders, implement sound shader switch	Begin creating new polygons. Normals, UVs, materials.	Start boss modelling, animation cycles continued, texturing	Props for scene, internal and external. Audio..
7	16th - 22nd July	Work on integrating physics into the game	Work on interface for character state changes	Sorting of new data, joining surfaces, creating new objects.	Animation, comic start, texturing	Texture painting and UV mapping
8	23rd - 29th July	Work on integrating physics into the game	Finalise all above stuff	Creation of new surfaces. New collision geo, bones & physics.	Animation, get everything finished by now	Texturing and texture optimisation
9	30th - 5th August	Testing, debugging and optimising	Debug, tweak, optimise, test…	Finishing up	Relaxing maybe? I hope so	Texture and shaders switching
10	6th - 12th August	Testing, debugging and optimising	Debug to perfection!	Finishing up and debugging	Debugging animation ;P	Final brush up of aesthetics
	13th - 19th August	DARE PROTOPLAY				

A.2 Post-It Board

Figure A.2: The Team's Post-It Board



Appendix B

Screenshots

Figure B.1: Early Screenshots

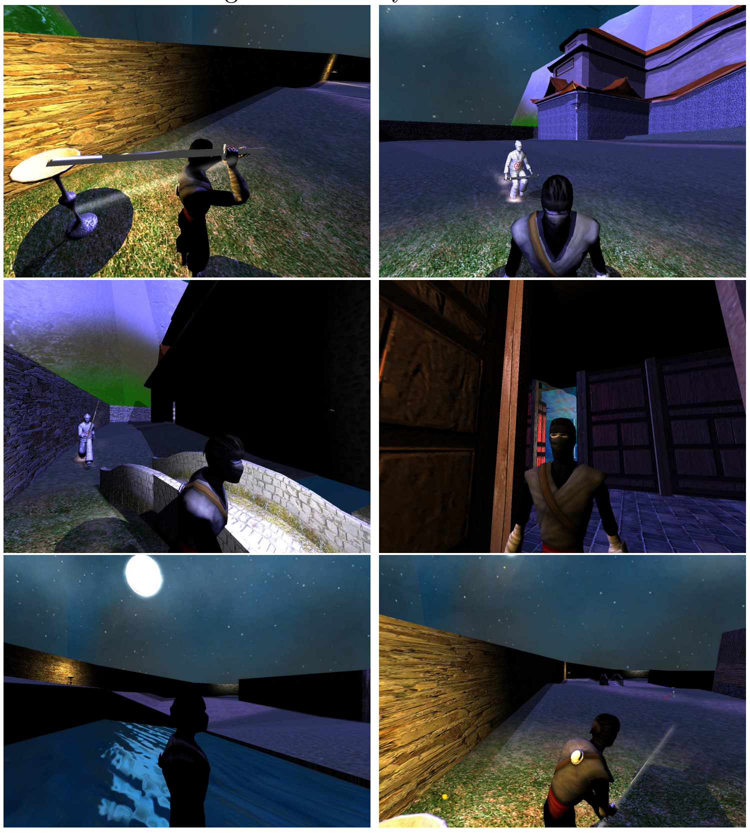
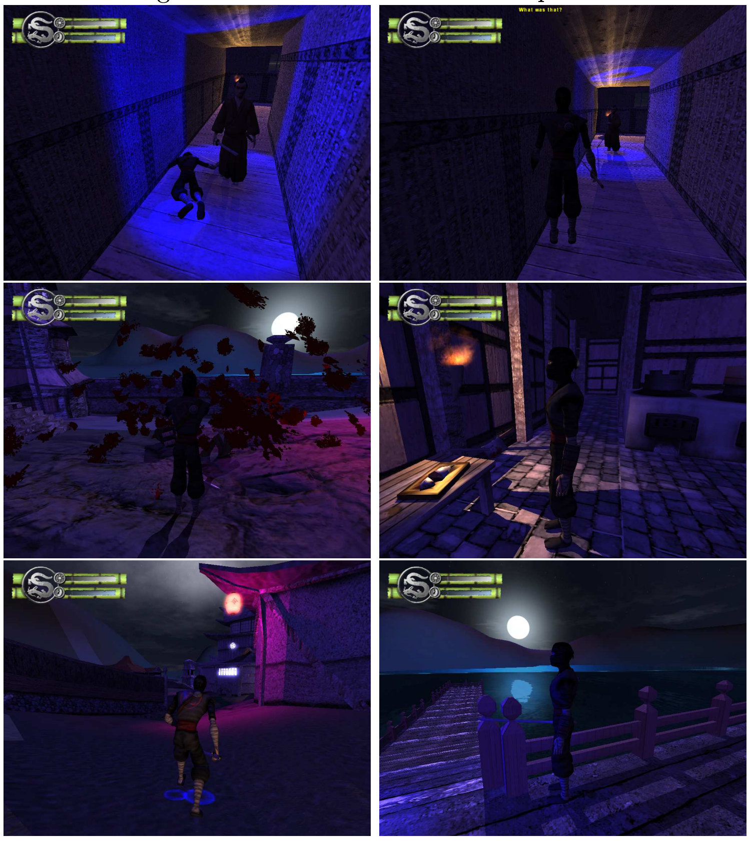


Figure B.2: Screenshots Near Completion



What was that?

Appendix C

Design Document

This page is intentionally left blank.
The Design Document is located at page 124

Bibliography

[1] Szalai, G. 2007. Video game industry growth still strong [online]. California, The Hollywood Reporter. Available from http://www.hollywoodreporter.com/hr/content_display/business/news/e3if5f9e6af1f789e8c28399b0253e7b78d

[2] Cook, D. 2007. The Chemistry of Game Design [online]. California, Gamasutra. Available from http://www.gamasutra.com/view/feature/1524/the_chemistry_of_game_design.php?print=1

[3] Scrum Methodology: http://www.controlchaos.com/ and http://www.softhouse.se/Uploades/Scrum_eng_webb.pdf

[4] Picture Reference: http://www.rockstargames.com/maxpayne/main.html

[5] Instinct Engine Documentation: located within the Instinct 'docs' folder

[6] Instinct Engine Package Reference: located within the Instinct 'docs' folder

[7] LUGER. G.F. 2002. Artificial Intelligence structures and strategies for complex problem solving. Pearson Education Ltd: Harlow, England UK

[8] BOURG. D. M. and SEEMANN. G., 2004. AI for Game Developers. O'Reilly Media, Inc.: USA.

[9] WATT. A. and POLICARPO F. 2001. 3D Games , Real-time Rendering and Software Technology. Pearson Education Ltd: Harlow, England, UK

[10] Black. P. E., "finite state machine", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 24 February 2006. (accessed 31 August 2007) Available from: http://www.nist.gov/dads/HTML/finiteStateMachine.html

[11] BROWNLOW. M., 2004. Game Programming Golden Rules. [online]. Charles River Media., VA: books24x7.com. Available from: http://library.books24x7.com/book/id_10420/

viewer.asp?bookid=10420&chunkid=0164695988 [Accessed 8 September. 07]

[12] Pick, S., (_____@europ.ea.com) 6 Sep 2004. RE: Please Advice. e-mail to Atieh, H. (has981@hotmail.com)

[13] SCHWAB. B., 2004. AI Game Engine Programming. [online]. Charles River Media., VA: ebrary.com. Available from: http://site.ebrary.com/lib/bournemouth/Top?id=10074871&layout=home [Accessed 30 August 07]

[14] Woodcock, S. M., 2007. Game AI Resources: State Machines & Agents. Available from: http://www.gameai.com/ [Accessed 8 September 2007].

[15] BUCKLAND. M., 2005. Programming Game AI By Example. [online]. Wordware Publishing, VA: books24x7.com. Available from: http://library.books24x7.com/book/id_9482/ viewer.asp?bookid=9482&chunkid=0722540724 [Accessed 9 September 07]

[16] BIELSER, D., MAIWALD, V. Interactive cuts through 3-dimensional soft tissue, Computer Graphics Forum 18(3): C31-C38, 1999.

[17] BRUYNS, C., MONTGOMERY, K. Generalized Interactions Using Virtual Tools Within the Spring Framework: Cutting, Medicine Meets Virtual Reality (MMVR02), Newport Beach, CA, January 23-26, 2001.

[18] SHEWCHUK, J. Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, 1st workshop on applied computational geometry, Association of Computing Machinery, Philadelphia, pp 124-133.

[19] VAN DEN BERGEN, G. 1998. Efficient collision detection of complex deformable models using AABB trees. Journal of Graphic Tools 2 (4) 1-13

[20] http://www.naturalmotion.com

Index