**Liverpool John Moores University**

08

**CMPCD3038 – Computer Game Programming and Workshop**

**Game Design and Programming – “Antechamber”**

**Alex Dixon and Alex Bleasdale**

Contents

[1 Introduction 4](#_Toc219380195)

[1.1 Abstract 4](#_Toc219380196)

[1.2 Analysis 4](#_Toc219380197)

[1.2.1 2D Game Genre Decision 5](#_Toc219380198)

[1.3 Development Environment 7](#_Toc219380199)

[2Concept 8](#_Toc219380200)

[2.1 Game Concept 8](#_Toc219380201)

[2.2 Influences 9](#_Toc219380202)

[2.3 Concept Art 10](#_Toc219380203)

[2.4 Sound 11](#_Toc219380204)

[3 Game Design 12](#_Toc219380205)

[3.1 Overview 12](#_Toc219380206)

[3.2 Game Rules 13](#_Toc219380207)

[3.3 Levels 14](#_Toc219380208)

[3.4 Player Mechanics 15](#_Toc219380209)

[3.4 Enemies 17](#_Toc219380210)

[3.4.1 Electro Pulse Orb 17](#_Toc219380211)

[3.4.2 Drop Block 18](#_Toc219380212)

[3.4.3 Turrets 19](#_Toc219380213)

[3.5 Collectables 24](#_Toc219380214)

[3.6 Control Panels 25](#_Toc219380215)

[3.7 Acid and Lava Pools 26](#_Toc219380216)

[4 Engine 27](#_Toc219380217)

[4.1 Engine Architecture 27](#_Toc219380218)

[4.2 Sound Manager 28](#_Toc219380219)

[5 Game Mechanics 29](#_Toc219380220)

[5.1 Collision Detection 29](#_Toc219380221)

[5.1a Point in AABB 29](#_Toc219380222)

[5.1b AABB vs. AABB 29](#_Toc219380223)

[5.1.1 Object Vs Level 30](#_Toc219380224)

[5.1.2 Player vs. Level 31](#_Toc219380225)

[5.1.3 Object vs. Object 32](#_Toc219380226)

[5.1.3 Token Interaction Matrix 33](#_Toc219380227)

[5.2 Sprite Animation 34](#_Toc219380228)

[5.2.1 Pixels to Texels 34](#_Toc219380229)

[5.2.2 Sprite Config / Scripting 35](#_Toc219380230)

[5.2.3 Putting it together 36](#_Toc219380231)

[5.3 Player Character State Machine 37](#_Toc219380232)

[6 Conclusions 38](#_Toc219380233)

[8.1 Alex Dixon Conclusion 38](#_Toc219380234)

[8.2 Alex Bleasdale Conclusion 38](#_Toc219380235)

[9 Appendices 39](#_Toc219380236)

[9.1 Major Bugs 39](#_Toc219380237)

[9.2 Version History 39](#_Toc219380238)

[9.3 User Manual 39](#_Toc219380239)

[9.3.1 Game Guide 39](#_Toc219380240)

[9.3.2 Level Editor Guide 39](#_Toc219380241)

[9.5 Tile map Equations 39](#_Toc219380242)

# Introduction

## Abstract

For your project you will need to design and implement a 2D computer game using DirectX. You may create any type of game you wish, with the agreement of your tutor, and implement it on any DirectX version (7, 8 or 9) available in the school or in your own machine. If you do not have an idea, your tutors will help you to choose your game.

The game developed for this project must be different from what you have developed in the L2 in any of the modules. For the Design and Implementation you are required to use a mix of CRC, Tokenisation and, an advanced Object Oriented Design / Programming techniques such as Yourdon or UML, from level 2 modules. Your game will contain at least the following characteristics: 2D, Animation, Sprite Management, Collision Detection, Sound, Game AI and Interface implementation. Some background research will have to be done on one of these subjects and explained in your report.

The game will include at least 3 different levels of difficulty, and a suitable interface showing the state of the game and enabling the interaction. The interface will have to be specified, designed and implemented along with the game.

This assignment will give you an opportunity to begin work on the specification and design for your game, and use Windows, C/C++ and DirectX programming. To receive full credit your design document will need to contain non-trivial sections as defined below.

## Analysis

Speaking from previous experience working on a shot time scale project is often difficult to get good results. Due to time constraints and the time it takes to make a solid game engine worthy of creating large amounts of content is a long process. With this is mind it is probably the best approach to keep the game simple, so that a solid game with good game play and a good amount of content can be created within the time scales. In the previous 2 years of university we were asked to create 2d RPG’s, and RPG is a very long and drawn out game type to implement because it is so heavy in content requiring a solid back end to the engine to deal with large amounts of data such as statistics of players and enemies, inventory items, locations, quest logs and many more things. Due to this the 2d games implemented during university projects have been somewhat incomplete although solid in the engine back end trying to create a full RPG between 4 people in 3 months time is not feasible and our games always lacked in content and looked a little rough around the edges (not enough time to create custom game assets as most time was spent on programming inventory systems and map systems). With these experiences now complete and many lessons learnt from the process It is now apparent that a simple 2d game with good game play that can be reproduced with ease is the best approach. Anybody would rather play a game that is complete and simple than an ambitious, complex game which is un-finished. We can analyse a few different game genres to make a decision on the type of game we want to create.

### 2D Game Genre Decision

**RPG**

As previously discussed the RPG is a large game which requires in depth story and character development. This game type is probably too ambitious for a 2 man team in such a short time but still it’s important to assess the good and bad points of an RPG game.

* Pros
  + Very immersive experience – The game size can allow almost film like story lines to develop and keep the user immersed in n exciting world.
  + Large world – RPG’s traditionally have large worlds with different races and types of characters. Lots of places to visit and areas to explore.
  + User satisfaction – Normally in an RPG game the playable character has “stats” which will increase and develop over time allowing the user to customise their character to suit their playing style.
* Cons
  + Requires large amounts of content – Most RPGS should have a great deal of playing time to complete (in excess of 20 hours playing time) on top of that the game should still be playable away from the main storyline so that the user can participate in side quests and “levelling” their character. This means lots of in game art will be required, as well as lots of story line.
  + Heavy in “stat” systems – an RPG game needs to be extremely balanced because its essentially a modern day version of card based games like dungeons and dragons. Enemies need to vary in level and have vulnerabilities to certain attacks or powers.
* Overall
  + The RPG genre as a whole is extremely popular and makes a very good game. But in this situation as previously discussed at the stage we are now as programmers to a game type like this justice will take to much time and inventible be a poor incomplete version of what we feel we are truly capable of. In short this game genre is probably too ambitious for this project.

**Casual Game (Puzzler / Mini Games)**

2D casual games these days seem to be all the rage. Lots of websites offer free games many of them puzzle type games or extremely simple games which focus on one mechanic.

* Pros
  + Quick to develop usually easy on content with levels being simple or the game itself providing harder difficulties as the game progresses
  + Popular to many people because its easy to pick and play for a short time period
  + Can be very addictive, a quickly game with simple yet fun mechanic can be described as a hook with hi-scores or something similar bringing the user back for more.
  + Minimal need for content so more time can be spent developing game mechanics.
* Cons
  + Extremely simple and can often be quiet minimal in appearance and implementation. This could be considered as an un-ambitious project.
  + Good fun for a short while, but this will fade over time as the games limited game play can become repetitive. It is important to note that this will not be the same for all casual mini games some will have a longer lifespan than others it just depends on the user and how long the limited game play will take to become stale.
* Overall
  + This type of game is popular in today’s market but for this project this could be too small a project to take on with very limited game play mechanics and content. I will allow time to find a solid game play mechanic and focus solely on that but creating a type of game like this that is original is hard due to the sheer number of casual and mini games already on the market.

**2D Side-scrolling game**

The 2d side scrolling game can come in many forms beat “em” up, platform, shooter and many others. It’s a game genre that I grew up on. Games like Sonic, Mario and Contra all hold a special place in my heart. Many games are still released in this genre nowadays notably now on handheld platforms more than consoles but it’s still extremely popular.

* Pros
  + Identifiable character – User take control of a character in the game that they are responsible for giving a nice base for the player to relate to the game and its character.
  + Many different styles and game types can be implemented.
  + Action based – most action is usually done in real time keeping the user interested and occupied throughout the games life cycle.
  + Easily scalable – The games size and complexity can be scaled easily allowing for a vast range of games to be created some being heavy on content others on game mechanics.
* Cons
  + Real time game play – requires the game mechanics to be solid and precise so that the user doesn’t become frustrated.
* Overall
  + This game genre is ideal for this project. We can scale the complexity to suit our needs so we find a good balance between complexity and completeness of the final product. The game style itself feels very user driven and the player can become immersed to a certain extent in the game world.

**Final Decision**

After analysing different game types and assessing prior experience we have decided to move forward with a 2d side-scrolling platform game. The RPG seemed too ambitious and time consuming to create a polished finished product and the mini game or puzzle type game seemed to un-ambitious and felt like and easy option. The platform game seemed like a nice in between with a nice balance of exciting game play but not too heavy in “back end” work. Most of the work in a platform game can be seen in how it looks and plays so this will help us immensely when implementing features that can be seen from the outset. With good content management and using editors new levels can be easily built allowing us to create one set of mechanics and reproducing these in different forms from outside of the code. With both group members having excellent game programming skills and experience we went for this approach essentially to make a complete polished game that is worthy of commercial distribution.

## Development Environment

Guidelines for this project have obviously been set in place to ensure the use of DirectX but there are many decisions to be made upon starting a project for the development environment tools and libraries to work with. Making good decisions early on will ensure that development meets the specifications needs. There are many other choices to consider that are sub decisions of graphics API, and Input API as well as the programming language itself.

* Programming language – C/C++ this has been predefined so no decision to make as suck. Being both of the group member’s strongest language this is ideal for us.
* DirectX Version choice – With the choice of DirectX 7, 8 or 9 available this is probably the first thing we should decide. It seems pretty straight forward to me, why use old outdated API’s there is no use for it as most modern graphics hardware supports DirectX 10 these days it seems as though 9 is somewhat the minimum standard and for this reason we will be working with the latest DirectX SDK and version 9 to maximise our abilities.
* Input API – We will use in this case DirectInput 8 which is the latest version and part of the latest DirectX SDK added to this we will utilise the newly developed XInput to enable XBOX 360 pad support. XInput is slowly becoming the DirectX standard hence no DirectInput 9 has been released. It is possible to obtain control pad input using DirectInput 8 but it does not maximise the potential of an XBOX360 controller for instance analogue triggers are only seen as digital buttons and vibrations cannot be set.
* Windowing API – Win32 as we will not be using this for much else other than creating a window and on some level Win32 will have to be used regardless if we use a cross platform API like *wxWidgets* this will just take care of windowing for us via Win32. We will just use Win32 for window initialisation.
* Sound API – FMOD will be used for sound due to previous experience using FMOD a simple 2d system is easy to implement. Having not used DirectSound before it makes sense to eliminate any unnecessary work load by learning DirectSound. Added to this most commercial games seem to veer away from using DirectSound and too use FMOD for sound most notably crisis recently used FMOD for sound and being the most advance game of all time our decision to use FMOD is support by the fact how highly revered FMOD is within the industry.
* IDE – Microsoft visual studio 2005 and 2008 will be used as our IDE based on what each group member has available to them.
* SVN – We will use tortoise SVN and a repository stored online to maintain version control. In any group project version control is a necessity it allows multiple users to work on the project simultaneously merging files and flagging any conflicts within these files so that it takes no time to combine 2 peoples work. Before using SVN we would manually have to combine our work usually taking a long time. Adding new files was not a problem but combining 2 files multiple people had worked on was difficult and often caused more errors in the process. The added benefits of SVN keeping track of versions will allow us to rollback if any major problems occur or if any work is lost our server has all the data saved.

# 2Concept

## 2.1 Game Concept

With the decisions made in the previous section on game type and development environment in mind we can now progress to the over game idea and concept before we begin to design it. We will be making a 2d side-scrolling platform game based around controlling a central character. The overall concept needs to be complex enough to have addictive game play which offers difficulty progression for the play but also must be simple enough to implement solidly and completely in time.

The name of the game will be **“Antechamber”** meaning a smaller room before a main room, it is derived from Latin meaning “room before”. The paradox will be that every room in the game will be an antechamber and the “main room” will never be reached.

You will play as subject AB14 a character that has no identity and knows very little about himself. You will be presented a series of challenges as part of a testing program where subject AD14 will have to use his cunning and agility to traverse traps and dangers each room. Each of these rooms will be considered an antechamber with subject AB14 thinking that once he has completed a set number of challenges some goal will be reached. The trick here is that the tests are never ending and subject AD14 is forever trapped in an eternal cycle of progression from room to room.

AB14 will be guided through the initial stages of the game and communicated with by only one entity a robotic voice giving instructions along the way. There will be no other human like characters in the game leading AB14 to think he is special but at the same time very alone.

The rooms will be considered as the size of a screen with set challenges in each to progress to the next. Each room can be considered as its own entity totally unique from any other and no pre requisites required from other rooms. Traps in the rooms will involve turrets which can shoot projectiles, to balance difficulty different projectile types will be implemented (bullets, homing missiles and lasers) subject AD14 is electronically tagged so turrets and other traps are ware of his location so they can respond accordingly. Gates and bridges will be in each room and subject AB14 will have to access control panels to open a gate or bridge which in turn will enable them to progress to a different part of the level.

To add more game play mechanics the game will run on a life system where if subject AB14 is hit by a projectile or trap he will lose 1 life and restart the current room. Collectable items will be available to get points and additional lives. The players score will simply serve as a way to measure your skill and compete with other people to obtain the best score. This will give the game a bit more playability keeping people to replay the game again to get better scores.

Difficulty will be balance throughout the game with a global difficulty setting where less lives are awarded at start up but this will lead the user to be awarded with more points for risking less lives to begin with. Rooms will become progressively harder, adding faster firing turrets and more obstacles to avoid.

At the end of the game subject AB14 will reach a door glowing with white light looking like an exit, this lead ab14 and the player to think that they have reached the end but upon entering the door no end is in sight just more rooms and more traps which seals ab14’s fate of infinitely trapped in a series of dangerous rooms with no end in sight.

## 2.2 Influences

With a huge media market these days with games and films there are very little concepts that have not been already implemented in films, games or both. Due to this fact it is extremely difficult to come up with a 100% original idea many things can be unique but there will always be similarities and whether we think it or not a lot of our subconscious decisions can lead us to duplicate something we have seen elsewhere. It is important to acknowledge influences on a game and speaking from experience in the games industry it is encouraged. Firstly applying some examples of what you want a game to be like immediately helps get everyone on the same page because other people can relate to your ideas better. This game has a number of influences although we didn’t sit down and suggest we are going to make a game that’s like a cross between game A and game B with the atmosphere of film C. Some of our influences became apparent after we had started work on the game. So to give some background on the game and a general idea of what its about the following influences can be considered as similar comparisons.

* Cube – The film cube is set inside a huge network of cube shaped rooms, each interconnecting with 6 others imagine a huge rubrics cube with each smaller cube being one of these rooms. Some of the rooms are trapped and must be dealt with caution. This is obviously similar in over all concept to antechamber as in our game you have rooms of traps and puzzles to evade along the way.
* N – N the game centres on a ninja character having to reach the goal in a series of rooms where you have to release switches and gates while avoiding turrets and traps along the way. Again very similar in concept to antechamber, we will have turrets and gates in self contained rooms. The game mechanic of antechamber will be very different to N as in N you play a ninja and the mechanic revolves around movement flow and velocity.
* Portal – Valves portal focuses on a text subject character pitted through a series of puzzles using portal technology. Again similarities to antechamber lie in the puzzles and rooms elements as well as a test subject character as in our game AB14. In addition to this the story is developed using an AI “GLaDOS” who speaks in robotic voice giving you instructions along the way.
* Knytt Stories – Is an exploration puzzle game. There are not many similarities in terms of game play to antechamber. However the art style knytt stories is very nice simplistic yet very distinctive and we have definitely drew influence on antechambers artistic style from this game.

## 2.3 Art Specification

In the past, our projects have usually used graphics created for other games, or rushed out some textures that look stretched or don’t match. With experience it is clear that to create a polished, professional look to a game, the key is consistency. This is also a project by programmers, so the custom art needs to be quickly generated. These were the primary factors when deciding on a visual style.

As a result, the art style of the game will be a classic retro, low resolution look. This allows content to be created quickly, and also gives a very distinctive visual style. Below are some examples of early art tests.

Character



Menu gui

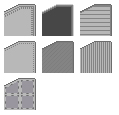


Environment



The above art was developed as an environment for a different game concept. After the chamber idea was developed, the visual style was kept but the environment changed to a more metal, futuristic world.

Level concept



## 2.4 Sound

Sound in a game is as important as visual impression to help the games atmosphere and ambience be transmitted to the player. It gives the player a greater sense of involvement and makes the game in turn feel more responsive as sound effects occur when key actions are completed. If anyone has ever watched a horror movie without the sound it will become apparent how crucial sound is, horror films don’t seem as scary with all the tension the music builds. Added to this is the fact that in any computer program if a button is pressed or action made it is often easy to be unsure if you have successfully completed the action if there is no sound or visual response. An example in the game of this could be when the player presses a control panel, with no visual response or sound the user could be unsure as to whether the control panel has been successfully activated. By adding a simple sound effect to this the user will be safe in the knowledge that they have performed the action and can then continue about the game.

Sound in the game will be simplistic to complement the simplistic stylised feel of the graphics. As you are in a futuristic puzzle world and can be considered as a prisoner or test subject we will be going for a futuristic sound. Minimalist synth and electronic drums for an atmospheric ambience will be in the background, simple digital sound effects for the turrets and lasers in the foreground.

Most notably in this game the main conveyance of story will be told by a robot voice who will instruct you of you task. This is the driving force of the game and the decision was made for a number of reasons.

* Spoken story line gives the player greater feeling of immersion that just static text.
* The voice gives the player something to relate to in the game, as their character is the only thing remotely human in the game visually. A human like voice adds atmosphere and gives the player a feeling of abstraction, they are alone tackling the puzzles and challenges in the rooms but it is not random he is there for some reason as he is being instructed. This will hopefully give the player drive to move on and unlock the mysteries in the story.

Sounds will be provided as MP3’s and the robotic voice will be obtained using free online speech to text converter “vozme”. With this tool we can write text files and convert directly into MP3 files which contain a robotic like voice.

# 3 Game Design

## 3.1 Overview

Antechamber will follow a distinct story like approach to game flow. It will have a strong beginning, middle and end.

* Introduction – The game start as described in section 2.1 will focus on “subject AB14”. You are dropped in the game to see a blank screen with command like line commands being typed across the screen. A robotic voice reads the technical jargon as if a program or application is initialising. The commands reach an end and almost TV like distortion tunes in to black screen with an open door and bright lights. Subject AB14 is dumped out of the door into the blackness. The robotic voice speaks and explains you are a test subject and you may suffer from some symptoms due to test protocols.
* Game Start – The first level of the game will consist of a series of rooms, each teaching the player of the dangers involved in the games outlining the rules and the objectives. This obviously wont be a straight forward list but embedded in story line to make the games flow feel more natural. In each room the robotic voice will explain a game element (backed up with textual display of the dialogue at the bottom of the screen).
* Middle – This will be most of the game body and be quiet scalable for us as a development team. In here we will construct as may levels as we see fit or have time to produce in the projects time scale. In these levels we will use combinations of the game puzzles and traps progressing in difficulty to give the player a nice learning curve. Different levels will look different in appearance so the player can identify how far they are. Levels will consist of a number of rooms which will act as their own contained mini levels. Once a room is completed the player will move off screen to the next and be unable to go back to a previous room. Once a collection of rooms is completed this will be the end of the level, the player will move into a teleport device and move to the next level.
* Ending – At the end the player will reach a door with bright light (similar to at the start of the game) you will be led to believe that this is the exit and subject AB14 will find some answers and possibly freedom. This will not be the case subject AB14 will find that the door just leads on to yet more rooms and levels of puzzles sealing his fate being eternally damned to battle through this never ending nightmare.

## 3.2 Game Rules

The game rules in the game in a nutshell will be very simple and the main difficulty level will progress by providing more puzzles for the player to solve and traps for the player to dodge.

* Life system – The player will be given a set number of lives at the start of the game this will depend on the difficulty level they select. The player will have the opportunity throughout the game to collect additional lives along the way.
  + Easy – 5 Lives granted at game start.
  + Medium – 3 Lives granted at game start.
  + Hard – 1 Life granted at game start.
* Losing lives – The player will lose lives on a 1 hit 1 kill basis so if they get hit by a projectile or trap that will cause the loss of 1 life. After losing a life the player will be placed at the start of the room they are currently in, in a safe position.
* Traps – Will be placed through the game causing dangers for the player along the way and hindering progress by reducing lives.
* Gates – The player may have to open gates to progress to previously unreachable areas, in some cases he may need to lure a projectile to hit a button and unlock a gate.
* Game over condition – The player’s game will be over when they reach 0 lives remaining. They will then have to go back to their last save or start the game from scratch
* Attack system – The user controlled player will be unable to attack or destroy any of the in game traps this will lead the player to have to avoid traps and dangers along the way.
* Score system – During the game there will be a series of collectable items the player can pick up collectables will add points to the players score. When the player reaches the end of a room or end of a level they will be awarded a bonus multiplier based on points collected, number of lives and game difficulty.
  + Bonus score = level points collected \* level multiplier \* lives \* difficulty
  + Difficulty bonus = (easy – 1, medium – 2, hard – 3)
* Hi-Scores – After a completed game the player will have the opportunity to submit their score to a hi-score list to see if they are the best at the game.
* Levels – Each screen will be considered as a room with puzzles and traps to solve and avoid. Once the player moves off the screen to the next this will be a new room and a bonus score will be applied. A collection of rooms will be grouped into a level where a much larger bonus will be applied upon completion.
* Game completion – Once the player has battled through many levels and rooms an ending will be reached. For the player this will be the end of the game, but for our in game character it will be the realisation of an infinite destiny to be battling through endless rooms and puzzles.

## 3.3 Levels

As described previously the levels will be split up into smaller sub levels know as rooms. The rooms will contain traps and puzzles to solve in order to exit the current one and move onto the next. A room will serve as a single entity once completed the player progresses to the next. A level will be considered as a collection of 5 or so rooms and once all these have been completed the next level is unlocked. Levels will contain rooms of a similar themed style (could be simple like the colour) so level 1 could consist of 5 red rooms and level 2 could consist of 5 blue rooms. This will give the player a great sense of progression as they will be able to identify which level and in turn which difficult they are on.

The levels will be tile based using hex file for input which will be created using a custom level editor built for this game. A number of different tiles will be available to create levels including solid blocks as well as partial blocks to create ramps and slopes. When placed together shallow slopes can be used to create the illusion of curves and make the level more interesting. Most simple 2d games tend to stick with simple tile maps that are just blocks and don’t include any sloped surfaces, it was for this reason we thought it was important to include surfaces other than square blocks so our levels would have a nice unique feel and more variety could be introduced. Collision detection on such sloped blocks will be more complex than simple solid whole tile blocks and we will explain in detail the collision detection approach in section 5 game mechanics.

The editor will also allow us to position turrets and gate puzzles so that the player has some objectives to complete in each room in order to progress. As the player enters a new room their position will be save so if any deaths occur during the current room the player will restart from this position.

## 3.4 Player Mechanics

The player mechanics will be fairly simple he will not have any attacks only the ability to dodge and evade dangers within the level.

* Walk – The player will be able to walk left and right on top of the level or in this case the tile map of our level. The player will be blocked by walks and be able to fall off ledges if there is nothing below him.
* Run – The player will have the ability to move faster than walking by performing run, again this will depend on the levels tile map.
* Jump – To reach other areas the player will be able to perform a jump to reach high places or evade obstacles.
* Double Jump – Whilst in the air the player has an opportunity to perform a secondary jump to gain yet more height. If timed correct the player will be able to get a maximum jump height by double jumping when the initial jump is at its peak.
* Interact – Some in game objects such as control panels or intercoms require a player invoked reaction, by standing by these and pressing the interact key will trigger the interaction event. A typical interaction event could be a gat opening or a response from the computer system instructing the player what to do in robotic voice.

A choice of control systems will be available to the player using different peripherals. The first will be standard keyboard input which is assumed that all PC’s playing the game will have. The other control system will be using XBOX 360 controller as an additional option for users who have one. Both systems will control all mechanics and hold no advantage over each other. However for a more comfortable and immersive gaming experience XBOX360 controller is recommended.

|  |  |
| --- | --- |
|  | |
| 2dcontrolsk.bmp | Cursor Key Right – Move Right  Cursor Key Left – Move Left  Cursor Key Up – Interact  Space – Jump / Double Jump (when in the air)  Shift – Run |

|  |  |
| --- | --- |
|  | |
| 2dcontrolsp.bmp | DPAD Right– Move Right  DPAD Left – Move Left  DPAD Up – Interact  Left Analogue Stick – Push left to move left, push right to move right  Button A – Jump / Double Jump (when in the air)  Button X – Interact  Right Trigger - Run |

DirectInput will be used to obtain keyboard Input where we can simply pass in any of the DirectInput virtual key codes to see if a key is pressed or a key is hit. We will use XInput the new DirectX standard for input to obtain the XBOX 360 controller input. XInput provides a function to simple retrieve the state of a controller. We invoke this every frame and use a global array to access the game pads states.

## 3.4 Enemies

To add difficulty to the game a host of enemy entities know as traps will be implemented to attempt to hinder the player’s progress. A number of distinct enemy types will be implemented and by using the level editor we can place these traps throughout each level.

* Electro pulse orb – A small entity which moves left and right along the tile map, It will sit on top of the levels tiles (can be considered as the levels floor). If the orb reaches a wall it will change direction and go back the way it came. If it gets the edge of a floor or ledge again it will change direction. This will cause the effect that the orb will patrol a certain contained area of the map. If the player collides with the orb it will electrocute the player causing him to lose a life.
* Drop Block – A block which remains still until the player’s position enters a certain range. When the player enters the influence region the block will drop down quickly to try and crush the player causing him to lose a life.
* Turrets – There will be a number of different turrets which will fire projectiles
  + Bullet – The turret interpolates its rotation to aim at the player if possible (rotation region will be locked to a certain range) the turret will then fire a bullet at a designated fire rate, The bullet will travel along the aim vector of the turret and stop after it hits the levels tile map or hits the player causing him to die.
  + Homing Missile – Like the bullet turret the barrel will rotate to aim at the player, once a projectile is launched it will track the player’s position and attempt to home in on the player changing direction and rotation as the player moves. The rotation update will be balance so the homing accuracy isn’t 100% precise causing a nice flowing movement as the missile changes direction.
  + Lazer – The lazer turret doesn’t rotate it stays fixed, and shoots a lazer beam which extrudes until it it’s a collision on the tile map. Lasers will remain on for a certain time period then turn off for a certain time period allowing the player to pass through.

### 3.4.1 Electro Pulse Orb

**Parameters**

The orb will have a speed parameter which will control the speed at which the orb moves this can be increased or decreased to make the difficulty of a stage easier or harder.

**Behaviour**

The orb will move either left or right in the x-axis at the designated speed. Upon collision with an edge or a wall its direction will invert to move the opposite way. In turn it will continue until it reaches another collision and invert direction once again. With a correctly designed map the orb will continually patrol back and forth

**Awareness**

The orb will be unaware of the player’s presence and as a result keep moving back and forth none of its behaviour will be dependent on the player’s position

|  |
| --- |
|  |
| pulseorb.bmp |

### 3.4.2 Drop Block

**Types**

There will be 4 types of drop block. The type is determined by its reaction upon reaction the block will move in one of 4 ways.

1. Move down (y-axis position increases)
2. Move up (y-axis position decreases)
3. Move left (x-axis position decreases)
4. Move right (x-axis position increases)

**Parameters**

To balance difficulty the block will have a few parameters that can be customised.

1. Move speed – The speed at which the block moves in its reaction direction
2. Return speed – The speed at which the block moves back to its starting position
3. Influence region – The boundaries the player has to be within for the block to start moving. For example the influence region could be:
   1. Min Y Boundary – block position Y – 20
   2. Max Y Boundary – block position Y + 20
   3. Min X Boundary – block position X – 20
   4. Max X Boundary – block position X + 20

**Behaviour**

The drop block will be a small block position at a certain place in a level. It will remain still until the player steps inside an influence region. Once inside the block will slam down, up, left or right depending on the blocks type. The block will try to crush the player but will be fixed on a strict path and be unable to veer away from this. Once the block hits a collision it will move back to its original starting position.

**Awareness**

The drop block will need to be informed of the players position at all times. If the player is within the influence region then the block will react moving in its designated reaction direction. If the block reacts it will have to return to its starting position before it can do so again. This will give the player chance to evade it.

|  |
| --- |
|  |
| dropblock.bmp   1. Player is outside influence region block remains still 2. Player enters influence region and the block reacts ( in this case moves down) 3. Block returns to its starting position and the process repeats   This process will be the same for the 4 types of block with the block reacting in different directions depending on the criteria provided at initialisation. |

### 3.4.3 Turrets

**Types**

There will be 4 Types of base turrets and 3 types of projectiles these turrets can fire. The 4 base turrets types will be the origin rotation of the turret. It can be considered that the types will be the following.

1. Upward facing – turrets barrel origin is 0 and pointing upwards.
2. Downward facing – turrets barrel origin is 180 and point downwards.
3. Left facing – turrets barrel origin rotation is 90 facing left.
4. Right facing – turrets barrel origin is -90 or 270 facing right.

**Parameters**

The turrets will take a number of parameters in order to customise them to balance difficulty.

1. Turret type (1 -4) specifies which type of turret will be used as detailed above.
2. Rotation range – specifies how much each turret can rotate about its origin rotation for example if the turrets origin rotation is 0 and the rotation range is 45 the turret can rotate to a minimum value of -45 and maximum value of +45.
3. Fire rate – specifies the time in seconds or fractional seconds that the turret will create new projectiles
4. Fire speed – specifies the speed of a projectile the turret will fire
5. Projectile type – specifies the type of projectile the turret will shoot (to be specified later)

**Behaviour**

The turret will obtain the players position and try to aim at the player by adjusting its rotation value. It will become locked if it reaches the minimum or maximum rotation range. If the fire timer has reached 0 the turret will spawn a new projectile. From here on in the projectile will be responsible for any behaviour beyond this.

**Awareness**

The turret will be aware of the players position at all times so it can update its rotation or aim accordingly.

|  |
| --- |
|  |
| turretbases.bmp   1. Rotation origin 0 2. Rotation origin 90 3. Rotation origin 180 4. Rotation origin 270 or -90   Labels 1 – 4 show the 4 main turret types. |

|  |
| --- |
|  |
| turretsbehave.bmp   1. Turret is not aiming at player, its desired aim is pointing directly at the player 2. Over some time period the turret interpolates its aim rotation until it reaches desired rotation or reaches the minimum and maximum rotation boundaries. |

#### 3.4.3.1 Projectile Types

As previously discussed the turrets will have an array of different projectile types to add verity and more complex game play combinations. We can consider the core mechanics of these projectile types to see how they behave.

**Bullets**

A bullet projectile is spawned by a turret when the fire timer reaches 0. It is given a speed which is dictated by the bullets owner (the turret that fired it) and a direction vector it will move along. The direction vector is obtained from the current aim vector of the turret. The turret is responsible for its own aiming mechanics as detailed in the previous section so when a bullet is fired by a turret it doesn’t have to be concerned of why it got there just what it has to do next.

The bullets will travel along the aim vector they were assigned upon spawning and be fixed along this path. They will be unable to veer from their assigned direction. A bullet will travel along this vector until a number of conditions are met.

1. Bullet hits player – the player dies and the bullet despawns
2. Bullet hits level – if the bullet hits the collision tile map of the level it will despawn and leave a small explosion behind.
3. Bullets life runs out – An optional condition can be set so that bullets life is not infinite, this may be beneficial in some cases so bullets can only reach a certain range.
4. Bullet leaves the screen – If the bullet leaves the screen we can despawn it as it will no longer be nor ever will be in view again.

|  |
| --- |
|  |
| bullets.bmp   1. Turret aims at the player and shoots some projectiles 2. The player has moved but the turrets aim hasn’t updated yet so new bullets are not going toward the player. Any existing bullets stay on this strict path 3. Turret aim has slowly updated to be aiming at the players new position and is shooting projectiles at the player which kill the player on hit and despawn 4. When a bullet hits the level it is despawned |

**Homing Missile**

Homing missiles are spawned in a similar manner to the bullets. The turret again is responsible for aiming toward the player and a homing missile is spawned along this vector. Due to the nature of a missile we will make the sprite longer and thin more like a rectangle than a circle so with this in mind we need to rotate the missile so that its tip is always facing along the aim direction.

When a homing missile leaves the turret its directional course is not set in stone like the bullets. It has the ability to adjust its directional vector based on the player’s position so it creates the homing effect. Like the turret the missile will have a desired aim rotation and vector of which it will strive to get to. The turning circle of the missile will not pin point accurate because the rotation will slowly interpolate to the desired aiming rotation all the while it will be moving along the gradually changing rotation. This will create a nice flowing effect to the homing missiles path.

Conditions to despawn the homing missile will be the same as the bullets conditions. In addition to all this the missile will spawn smoke particles at the rear of the missile causing a nice tail effect.

|  |
| --- |
|  |
| homingbehave.bmp   1. Turret aims at the player and shoots some projectiles. 2. The player has moved but the turrets aim hasn’t updated. The existing missile updates its aim to head to the new player position. 3. Missile spawning smoke behind it to create “tails”. 4. Missile collides with the tile map and explodes. |

**Lazer**

The laser turret will differ slightly to the previous two. A long lazer beam will be projected from turret to where the laser beam intersects the levels collision tile map. The lazer will stay on for a set period of time and remain off for a set period of time. In development we termed it “Boolean lazer” as it is either on or off. Conditions to end the lazer will be simply its life timer, upon collision it will stop and upon collision with the player it will stay on but the player will be dead from lazer shock!

|  |
| --- |
|  |
| lazerbehave.bmp   1. Lazer on – stops when it hits some part of the tile map. 2. Lazer off.   3 – 4. Lazer turret does not rotate regardless of player position. |

## 3.5 Collectables

During the life cycle of a level the player will be able to obtain collectable items to aid their progress. There will be 2 types of these so called collectables one which grants the player a new life the other which will give the player additional points in the form of energy. Simple collision detection will be applied to these objects and once collected they will despawn for the duration of the game so they can only be collected once in one session.

**Energy Cell**

The energy cell will increase the player’s energy score. At the end of the level the player will be awarded these points and given a multiplier based on the amount of energy they have. The player’s energy score will depreciate over a time like a timer. So the longer the player takes to complete the level the players score will have depreciated somewhat in the process. Simple AABB collision detection will be applied and if the player collides with the energy cell he will gain the points and the cell be despawned for the rest of the game.

**Lives**

Lives will work in the same way as energy cell in terms of collision detection, however when a life is collected the player will simply have 1 life added to their current lives tally and there will be no depreciation timer.

|  |
| --- |
|  |
| collectables.bmp   1. The player walks along and can see some projectiles. 2. He walks up and collects the first projectile. |

## 3.6 Control Panels

There will be a number of control panels scatter throughout the levels. The player will be able to stand next to a control panel and interact with it. Depending on the type of control panel it is the player will get a different response. Simple AABB collision detection will be required to determine if the player is stood in front of a control panel. If a collision occurs the panel will check to see if the player has interacted with it (by pressing interact key or button). Once the pane is activated it will react in the following ways.

**Open Gate**

The panel opens a gate allowing the player to move to a new location. Some gates may have a double opening, meaning that there are 2 ways it blocks but only one at a time. So at state-1 the gate blocks route A and rout B is open. The player interacts with the control panel and this activates the gate, in state-2 route A is open and route B is closed. Control panels and gates can be nested to create a larger gate puzzle for a room. So the player has a link of gates and control panels to activate to get to the final goal.

**Open Bridge**

Like the gate mechanism bridges will work in the same way, and will sometimes even be used as gates to block routes the player can go in the y-axis. Due to gravity, we can use bridges to help our player get across dangerous pools of acid or chasms.

**Intercom**

Intercoms will act to progress the story and also teach the player the game. By activating an intercom the player will get an onscreen dialog message with storyline or instructions and this will be accompanied by our robotic voice of the testing facility to help make it more immersive. If an intercom is playing, using it again will end the message early.

|  |
| --- |
|  |
| homingbehave.bmp   1. Turret aims at the player and shoots some projectiles. 2. The player has moved but the turrets aim hasn’t updated. The existing missile updates its aim to head to the new player position. 3. Missile spawning smoke behind it to create “tails”. 4. Missile collides with the tile map and explodes. |

## 3.7 Acid and Lava Pools

A traditional gaming pitfall the acid or lava pool, this can obviously be translated into many other scenarios (water, quick sand and so on) but acid and lava were our initial thoughts. The player simply has to avoid these by jumping over them and has to ensure he doesn’t fall in. If he falls in he melts or burns and therefore dies, causing a life to be lost. A number of different tiles will be specified for this and they will be integrated into the tile map of the level. The acid and lava will animate on the surface to create a bubbling and melting effect. A nice animation on the player sprite will polish up the overall process.

|  |
| --- |
|  |
| homingbehave.bmp   1. Turret aims at the player and shoots some projectiles. 2. The player has moved but the turrets aim hasn’t updated. The existing missile updates its aim to head to the new player position. 3. Missile spawning smoke behind it to create “tails”. 4. Missile collides with the tile map and explodes. |

# 4 Engine Implementation

The Engine, commonly referred to as the “backend” of a game, is the base for all the lower level functions of the application. It controls the window management under whatever operating system is being used, and fundamental things like input, time and content management, and game object management.

## 4.1 Engine Architecture

As stated earlier, this game will use win32 for windowing and interfacing with the operating system. Both team members are confident with its use and have experience with the API, in several different applications. It is also the most commonly used for unmanaged DirectX game implementations. As far as code management is concerned, the backend win32 and DirectX elements will be grouped together, as many of the base classes and structures for Direct3D and DirectInput8 will be used in conjunction with win32 functions.

### 4.1.1 Win32

The application has two “main” functions, which control the entry point and window messages under Windows XP. These are “WinMain” and “WindowProc”. WinMain is the standard win32 callback function, and WindowProc is called within it, using the Translate and Dispatch message functions within the main application loop. Inside the main function, the window class is registered, and the window created. After this, the input manager is initialized. This works from DirectInput 8, and will be discussed later. Finally, Direct3D is initialized, calling init\_d3d() from the d3dapp.cpp file, followed by setting up orthographic projection and the D3D matrices.

After initialising, we enter the application loop. Here we do several things:

* Check for messages from the OS (such as quit)
* Execute the “Scene2D” object, which manages the game state
* Render a frame of the game
* Detect user inputs (mouse, keyboard, gamepads)

There are two ways to exit the game loop; either receiving a “WM\_QUIT” message command from the app or OS, or if the execute function of the 2d scene object returns false. After the loop is exited, DirectX resources are released, and the program will close.

There are some other Win32 functions used globally to manage some low level elements:

* Application timer (***apptimer***). This uses “GetTickCount” to determine the time passed each frame.
* Debugging output (***dxio***). We use “OutputDebugStringA” to print to the output window, using a variable argument list for inputs.
* Random number generator (***rand***). Returns a psuedo-random number using “frand”.

### 4.1.2 Direct3D Management

The DirectX part of the engine has several files. The main one is “d3dapp”. This has 3 functions:

**Initialize D3D**

This function creates the Direct3D interface, sets window properties and creates an instance of the D3D device class. It also creates the instance of Scene2D, which in turn creates a Font instance and pre-loads sound resources.

**Render Frame**

This function quite simply prepares D3D for drawing all the game objects for this frame, using Clear, BeginScene, EndScene and Present functions. The render() function is called from the 2D scene object, which renders the objects in a linked list. Time is also recorded before and after rendering, and after the Present function call. The Present function will wait for vertical sync with the monitor, so the time passed is recorded to obtain the actual framerate.

**Cleanup D3D**

Cleanup simply releases the device context and instance for Direct3D.

Other Direct3D management includes:

* Texture loader (***d3dtex***). The program uses “D3DXCreateTextureFromFileExA” to load texture resources. The data is stored in a D3D texture object, “IDirect3DTexture9”, and this is stored in a structure along with texture dimension data. This data can be loaded optionally when calling the texture load. A Win32 FILE object is used to load the texture and find the dimensions. This is used later in sprite animations, which use a sprite sheet, so precise texture coordinate data is required.
* Global matrices (***matrices***). These are globally declared D3D matrices used for the orthographic and identity matrices. It is just useful to have them accessible from anywhere.

### 4.1.3 Input Manager

The input manager uses a mixture of Direct Input, XInput and Win32. This is simply for convenience as some things are easily done through Win32, with better results. See the ***dappinput*** files in engine > dx filter for source code.

**Keyboard Input**

**Mouse Input**

**GamePad Input**

### 4.1.4 Render Pipeline

The render pipeline works using a doubly linked list. This is a fairly standard method of linking objects together, and works by giving each object in the list a pointer to a previous link, and a pointer to the next link in the list. The list itself has a first link point and last link pointer, enabling an entry point to traverse the list from either end. See ***link.h*** in the engine > objects filter for source code. Below is an example linklist.



This list has 4 links in it, each one linked to the next and previous, except the first link, which has no previous link, and the last link, which has no next link.

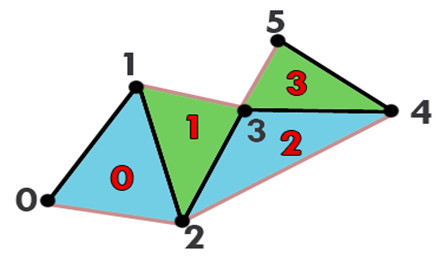
All “objects” (not necessarily all classes) in the game are derived from the Link class, which means when they are created, they are added to the render linklist by the Link constructor. Once an object has been added to the list, it will have the execute() and render() virtual functions from Link, called each frame. Here is a list of the LinkList class’ functions.

* **Add** – takes a Link pointer parameter and adds the link to the end of the linklist, making it the last link.
* **Remove** – This function takes a **handle**. This integer variable is a number assigned to all new objects when they are added to the linklist. The handle is used to traverse the list from first to last, and if it matches the handle for a link in the list, that object is deleted, and links pointing to it are instead pointed to each other. The reason for using handles with linked lists is to combat one of the problems with lists. If a pointer to an object is used, then at some point that object is removed, the pointer will be invalid but still appear to point somewhere. Using handles ensures that if an object has been removed, it cannot be found using a handle, as each handle is unique.
* **Get Object** – This also takes a handle parameter and returns a pointer to the object found, if any.
* **Execute and Render List** – these functions call the execute and render functions for all objects in the list respectively. All links have these functions so that they can be accessed in this way. When an object has been added to the list it will instantly be executed and rendered every frame until the object is deleted. Of course, these functions don’t have to output anything in particular, but the render function is the only place an object can render pixels to the screen.
* **Cleanup List** – This functions traverses the entire list from start to finish, removing each object as it moves through. This is only really done when closing the program, as some objects are always needed for the game to run.

As stated above, the Link class has an execute() and render() function. The execute order of objects is not critical to the outcome of the game at runtime; however, the render order is far more crucial. At first, multiple objects created inside the level would be rendered in the wrong order, meaning they would be obscured by other objects in the scene. To combat this, each link has a render priority. The set\_priority() function takes a float value which is used to move the object through the list until it is in order. The lower this number is, the further towards the end of the list the object will be placed. Since the list is rendered from last to first, this means a lower number indicates that object is to be rendered near the “back” of the scene, with most other objects being rendered over it. Some constants are defined to make the process easier, for example, RENDER\_BACK, RENDER\_MIDDLE and RENDER\_NEAR\_FRONT. This means an objects derived from link, for example an actor, can have priority set to be near the front, so the player character is always visible.

### 4.1.5 Primitives

The basis for all objects that are rendered to the screen is the Triangle Strip primitive. The winding order of strips is important for drawing multiple polygon objects. Here is an example of a triangle strip:



Black lines indicate vectors between vertices. Black numbers are the vertices in the order they were drawn. Red numbers are the polygons in the order they are drawn. Light red lines indicate vectors between vertices that are connected to complete each polygon (they always skip a vertex). Finally, blue polygons are the first of a quad, and green polygons are the second.

The “Quad” class is the basis for all small objects in the scene. It simply stores a position, size, texture coordinate set and is derived from Link. A triangle strip with 4 vetices is used to create a perfectly square quad, with a texture.

### 4.1.6 Game State Machine

### 4.1.Engine UML

## 4.2 Sound Manager

To create sound in the game we will be using the FMOD sound library. Decisions for this were because of the state of DirectSound which is now deprecated. Microsoft’s new solution is the XACT platform (cross platform audio creation tool) now having used this previously with XNA and C# we decided to avoid using XACT in this scenario because of the way it deals with sound files. In XACT you have to compile sound and wave banks using a separate tool and then access sounds using this custom file setup. It means lots of unnecessary work to create a sound bank and cannot be changed and modified quickly. FMOD allows us to simply create files in anyway we want, store them in a folder and then load them or stream them using FMOD.

We will supply a number of functions to play sounds allowing different types of sound playing.

**Load Sound**

This will load a sound into RAM and be primarily used for small sound effects, each one will be assigned an ID (i.e. lazer shot, missile shot, and explosion) and because sound manager will be global we can then play any of these sounds by ID on demand.

**Play Sound**

This will just play a sound from a given ID as specified above. Overloaded functions will allow us to boost the volume slightly or decrease the volume slightly to get a nice sound balance.

**Play Loop Sound**

A sound can be played like above but with a flag to specify it is looping. We use a different function for this because if we play 2 looping sounds like a lazer beam at staggered interval we will get a horrible result with 2 sounds clashing. IF an existing sounds is looping we don’t play it again but flag that our new caller wants this sound to loop, once all callers have decided they want the loop to finish for instance a lazer beam has stopped. If one item stops calling a loop sound then this doesn’t mean we can stop it immediately because other objects may want this sound to still be looping, so once all the callers have finished we can stop the desired sound.

**Play Streaming Sound**

This will play a larger sound file on the fly without loading the whole thing into RAM. This is ideal for music or dialogue which there will be many files of. We just pass in the file name and play the sound as and when we want.

**Stop Streaming Sound**

It will be required to stop a streaming sound which can be done easily by tracking the pointer to the sound stream that’s playing. We can then pass in this value and tell FMOD to stop the stream.

## 4.3 Text

## 4.4 Graphical User Interface

## 4.4 Level Editor

# 5 Game Mechanics

## 5.1 Collision Detection

We will be using some standard collision methods for this game and also some new custom ones which are fit for our needs, before moving on a brief summary of the standard collision methods we will use.

### 5.1a Point in AABB

An AABB is an axially aligned bounding box, and has all of its edges running parallel to either the x or y axis. To check if a point is inside the box, we check if the point’s value on each axis is greater than the minimum extent of the box and less than the maximum extent of the box. If we are within the minimum and maximum extents on each axis then we are inside the box and therefore have a collision.

### 5.1b AABB vs. AABB

This test extends the point in AABB further. In this scenario we check if the boxes overlap on each axis. Because we can overlap in a number of ways we have a few OR cases for this collision test. There is an overlap on an axis is the following conditions are met. The following example for box A and box B

* IF the min extent of A is greater than min extent of B AND the min extent of A is less than the max extent of B
* OR the min extent of B is greater than min extent of A AND the min extent of B is less than the max extent of B
* OR the max extent of A is greater than min extent of B AND the max extent of A is less than the max extent of B
* OR the max extent of B is greater than min extent of A AND the max extent of B is less than the max extent of A

If these conditions are true for both the x and y axis then we have a collision.

### 5.1.1 Object Vs Level

As described in the design section we will be using a tile map to create the levels, but this will not mean we will use tile based collisions. The level will consist of different tile types, of which we will require different collisions.

**Full Tiles**

Full tiles cover the whole of a tile and block that whole square of the tile map. Based on the tile map setup we know that these types of tiles have an ID of 0 to 17. When performing collision we calculate the tile of the tile map where the collider is and then check which tile exists at that position. There could be no tile there so we don’t flag any collision. If we hit a tile with ID 0 to 17 then we perform a simple AABB vs. AABB collision detection test, or a point inside AABB collision test. If we have a collision we respond accordingly.

**Slope Tiles**

Slope tiles allow us to create tiles which cover a portion of a square. The way and which we determine how much of the square is covered is derived from Y = M \* X + C. Because we know the x-position of an object we can calculate if we are colliding with a slop region by using the equation of a line.. We know if we are below the line we have a collision and by using the line formula we can calculate were on the line we need to be to be exactly on top of it. The formulas for each of the slopes will be hard coded and by finding the tile ID we collide with we can lookup the formula for that slope.

For example a slope which covers half the tile going from the bottom left corner to the top right will have the formula:

Y = 1 \* X + 0

Each other tile will have a corresponding formula. Please see appendices for a full list of formulas.

**Underside Slope Tiles**

Underside slopes will work in the same way as slope tiles, however in reverse. The part of collision is if we are above the line when we apply the formula, if so we respond accordingly.

**Acid Pools**

This type of tile will cause the player to die when colliding, it will not be a full tile, but just a flat portion of a tile so will use the formula Y = 0 \* X + 0.7 where 0.7 is 70% of a tile. Collision with these tiles will work the same as slope collisions but with different responses.

### 5.1.2 Player vs. Level

The player will need to collide more precisely with the world to create smooth movement mechanics. So when colliding using a naive AABB vs. AABB or point in AABB test will not be sufficient. We must also determine how the player needs to stand or be blocked by a tile. For instance collision with the top of a block means the player should stand on top of the tile, if he collides with the left hand side of a tile he should be offset to the left and stopped from continuing to walk right. A full AABB will cause stickiness in the collision so instead we will construct a bounding shape around the player using a series of points. We will then check collisions with each of these points and depending on the point we are currently checking we will respond differently.

**Broad Phase**

We check the points of the player separately, so when we select the point we do a quick point in AABB test to find out the tile of the map we are currently in. In the level class we can pass in this tile position and obtain the tile type which occupies this space. Once we have the tile type if there is a tile we proceed and perform more collision tests.

**Floors**

If we collide with either of the bottom points then we know we have collided with a floor, the response to this will be to calculate the top of the collided tile and offset the player by the amount he has overlapped so that he stands nicely on top of the tile. For slope collisions, we will use the lookup table to find the formula of the slope from the given formula we substitute in the x position of the current point we are checking collision with and then calculate the y position of the slope using the formula. If we collide with the slope we offset the player upwards so he stands nicely on top of the slope.

**Left Wall**

If we collide with either of the front points (right side of the player) we know that we are colliding with the left wall of a block. We then find the left edge of the block and offset the player by the difference so he is blocked precisely but the left wall and cannot continue to move right.

**Right Wall**

This works the same as collisions with the left wall, except this time we check the back points of the player (left side) and offset the player right stopping the player to continue moving left.

**Roofs**

If the top points of the player collide we know he has a hit a roof so we find the bottom of the tile and offset the player downwards so he is not in collision. We perform the slope collisions the same as in floors, but the line is obviously in this case preventing the player from moving upwards.

**Acid**

If we collide with the bottom points and acid tile, we perform the Y = M \* X + C formula the same as we do for slopes, we offset correctly and then set the player to death state.

### 5.1.3 Object vs. Object

When objects collide in the world we will use different collision tests to determine whether we have collided and how we need to respond. There are many objects in the world but most of them just collide with the player.

* **Player vs. Projectiles**
  + **Bullets** – Point in AABB – The position of the bullet is the point and the player sprite is the AABB. If the bullet is inside the player region we have a collision. We respond by death to the player, and despawn the bullet.
  + **Missiles** – Point in AABB – The position of the missile is the point and the player sprite is the AABB. If the bullet is inside the player region we have a collision. We respond by death to the player, and despawn the missile.
  + **Lazer** – AABB vs. AABB – the 2 AABB are the player sprite and the lazer sprite, we check for overlap and if we collide the player dies
* **Player vs. Collectables –** AABB vs. AABB the 2 AABB are the player and the collectable sprite, if we collide the collectable is despawned and score is increased.
* **Player vs. Control Panel –** AABB vs. AABB the 2 AABB are the player sprite and control panel sprite. If we collide we only respond if the user wishes to interact and if so the control panel is activated, it will open a gate, play a dialog sound file or save the game.
* **Player vs. Gate –** Custom method similar to the level collision check will be made here. We check the players bounding points described in 5.1.2 if we detect collisions with front back top or bottom we know how to respond, and respond similarly to a complete level tile.

Those collisions along with projectile vs. level which is point in AABB are the only collisions which will occur in the game. A full comprehensive token interaction matrix is shown in the next section to summarise the collisions.

### 5.1.3 Token Interaction Matrix

## 5.2 Sprite Animation

Sprite animation is essential in any 2d game it brings characters to life and even though it has no bearing on the movement mechanics it makes games feel a lot smoother, responsive and more pleasing to control. In order to perform sprite animations we will use multiple frames for animation which will periodically incremented and looped to create the effects of the player walking, jumping and dying. The sprite class will perform animations, so a the player and electro pulse orb will be the animated sprites in the game. For simplicity we will use the player sprite as an example of how animation is implemented but this code will be in sprite so other objects will benefit from sprite animation if needed. The sprite class performs animation. The actor class controls the finite state machine which controls the animation states to display and how to respond.

We will be using large sprite sheets which contain all our frames of animation so that file overhead is low and we don’t have a file structure littered with many small animation frames. We will then display a small window of this texture to be the frame we see on screen. Changing the window of the texture we display will cause the sprite to appear animated.

|  |
| --- |
| Figure – Player sprite sheet |
| ab14.bmp |

### 5.2.1 Pixels to Texels

If we specify the pixel dimensions of a frame we need to calculate the texel size for the current image. Because texture space runs from 0 to 1 with 0.5 being the middle we can simply divided 1 by our image dimensions to get the texel size.

|  |
| --- |
| Example – Calculating texel size |
| For a 64 x 64 image |
| Texal size x = 1 / 64  Texal size x = 0.15625  Texal size y = 1 / 64  Texal size y = 0.15625  For a frame which is 12 x 12 we multiply texal size by 12 and get a window on the texture which is 0.1875 x 0.1875 |

For the above example we supply the values as the u and v coordinates of a texture. We may want to display a frame in the middle of our sprite sheet so we need to know where this frame starts not just its dimensions so we need some way of describing the contents of our sprite sheet and how to determine the start frame of an animation, how many frames to run for and many other things. If we do this robustly we can even allow animations to have varying sized frames and many other abilities.

### 5.2.2 Sprite Config / Scripting

We created a small scripting language to deal with this, so each sprite has a script or config file to accompany it. We load in values from this when we initialise the sprite and will use the values to aid us in the animation process.

We use a method similar to the command design pattern where we read in a file if we hit a keyword we then perform a specific operation reading in the appropriate data. In the code we can hardcode cases for each of these keywords so if we hit an identifier such as “state\_frames” we know we have to read in 2 integers, the first being the state ID the second being the number of frames in that state.

Sprite configs will be read in using std::ifstream so the notion is quiet simple:

* Open the config file for a sprite (same file name as the sprite)
* While its not the end of file
  + Read in a character array which will be an identifier
  + Switch cases depending on the identifier
  + Read in any subsequent data that the keyword identifier requires

The keywords we are looking for in a sprite config and subsequent values we need to read in are as follows:

* State\_frames (int frame id, int number of frames)
* Number\_of\_states (int number of states)
* Total\_frames (int total number of frames)
* For (int loop count) “ , “ (int frame x size, int frame y size)
* #End – file is finished

Due to time constraints error handling was not implemented for the sprite config, so dry runs on the sprite config were needed to prevent any issues, this was not a problem just additional car was required.

Using the keywords we read in the number of states we wish to have and the number of frames they include. We then use the total frames command to loop through and assign each frame a size, the “for” command can be used to assign sizes to multiple consecutive frames so writing out every frame dimension is unnecessary.

|  |
| --- |
| Example – Player sprite config |
| “Ab14.asc” – main character |
| number\_of\_states 6  state\_frames 0 1  state\_frames 1 8  state\_frames 2 8  state\_frames 3 3  state\_frames 4 2  state\_frames 5 1  state\_frames 6 9  total\_frames 32 : for 20 : 8 20 , for 1 : 9 18 , for 1 : 10 19 , for 1 : 8 20 , for 9 : 9 20  #end |

### 5.2.3 Putting it together

With the contents of the sprite config we fill in some arrays to store this data. Firstly a vector2 array to store the frames dimensions each frame will have a dimension. So if we are displaying frame 10 we can access frame\_size[10].x or frame\_size[10].y. We then set the players size to the size of this sprite and then calculate the size of Texels to set as the u and v coordinates as described in 5.2.1.

We constantly update animations if our animation timer reaches the animation rate we then increment the animation frame to the next frame. If we go past the final frame of the current animation state then we go back to the beginning if we wish to loop. In some cases when we reach to the end of an animation we change states, for instance when performing death animation we play the animation once, stop at the end, wait for a respawn timer and then set the state to the respawn state. The respawn timer then sets us back to idle state when it is complete.

To find the start position in texels of an animation frame we loop though the sprite sheet contents summing the frame dimensions of any frames or states preceding the current frame. We know the current frame of our animation by summing the number of frames of preceding states. When looping through the sprite sheet to find the texel frame origin of the current animation if the texel is greater than 1 on the u axis then we move down on the v axis and go back to 0 on the u axis, effectively moving down one line in the sprite sheet.

|  |
| --- |
| Figure – Calculating frame origin |
| framedivision.bmp |

|  |
| --- |
| Example – Finding the current animation frame |
| For state 3 |
| State 1 = 1 frames at 8 x 20 in size  State 2 = 8 frames at 8 x 20 in size  State 3 = 8 frames at 8 x 20 in size  State 3 start frame = state 1 frames + state 2 frames  State 3 start frame = 9  Frame\_origin = 0  For(I < 9)  Frame\_origin += frame size[i] \* texel size.  If frame\_origin.x > 1 , frame\_origin.x = 0, frame\_origin.y += row height in texels. |

## 5.3 Player Character State Machine

# 6 Conclusions

## 6.1 Alex Dixon Conclusions

## 6.2 Alex Bleasdale Conclusion

This has been the most complete project I have worked on at university. I am very pleased with the result. For the project I completed the core engine, level rendering and the level editor, the art, the graphical user interface and game state machine, sprite management, text display and fonts, and initial player character development. Most of these are backend or frontend elements of the game. The design process was shared, and I am very happy with the game concept.

Experience has shown us that the way to approach game development is to not get fussy with details of a game idea, but rather develop an idea that makes development easier. The original concept for a story and game meant that a lot of content would need to be generated for very small parts of the game. This is not sustainable in such a short timeframe, as content creation takes time and this makes it much more difficult to get a completed project. This is why the level editor was developed and integrated into the game. It allows new levels to be created extremely quickly, which is part of the reason why we have been able to focus on so much of the inner workings of the game, and not worried so much about content creation.

# 9 Appendices

## 9.1 Major Bugs

## 9.2 Version History

## 9.3 User Manual

### 9.3.1 Game Guide

### 9.3.2 Level Editor Guide

## 9.5 Tile map Equations