Games Programming Design Report

By Thomas Butterwith
090004683

Table of Contents

1.0 - Introduction

This document contains the detailed design and thought processes that went into the creation of my platform game Hats. Hats is a simple two dimensional platform game which incorporates a puzzle aspect creating a unique gameplay experience. The user or player interacts with the game through a set of two onscreen characters with the main aim of the game being to manipulate these characters to check points at the end of each level.

During each level the player must traverse a series of puzzles to reach a checkpoint, with each puzzle only being passable after collection of a certain hat. Each hat, one located on each level, will enhance a particular attribute of the onscreen character, allowing the player to complete the puzzle. One of the major game play elements is the problem of how does the player get both characters through the puzzle when only one is able to wear a particular hat at a given time. Hats can be kept by the player through each level meaning the player will have to select the appropriate hat for the puzzle at hand. This could involve a combination of hats between the two playable characters.

My main focus during the design process of Hats was to create a game that is simple enough for anyone to pick up and play but be challenging enough that people continue to play the game after getting to grips with the mechanics. Over the course of the build process I aim to create four playable levels. This, I feel, is a manageable amount for me to create and gives the player enough time to get a real taste of what the game is about.

2.0 - Game Design Overview

Title

Hats

Genre

Platform puzzle game

Target Audience

16-35 year olds. Those who have played indie games in the past and enjoy the charm of a simple yet creative game.

Concept

A user must guide two characters through a series of levels, each level involving a set of puzzles. Each puzzle can be solved using a hat the player has collected throughout the game. A series of different hats are available in the game, each hat changing one or more of the characters attributes e.g. jump higher, move faster, become taller etc. The user must manipulate both characters through the level to pass, combining the abilities of the two characters to pass each level, creating the main game play mechanic. The player can only control one character at a time and will have to switch between the controls to progress through the level.

Unique Selling Point

Although the game is similar to other platform games in terms of gameplay, the game will be unique in terms of how the puzzles are solved and the use of multiple characters that, though active at the same time, must be controlled independently. By introducing a series of collectables that directly modify gameplay I believe I have created a unique, fun to play game which is easy to pick up and understand.

Plot

TBC

Similar Games

Braid, Super Mario, Thomas Was Alone, Super Meat Boy

2.1 Game Design Decisions

The process of designing a game from scratch is something I have found to be very complex. It is difficult to strike a balance between a game that is both original and fun to play. The platform genre has been around for a long time, evolving over the years in games people know and love, Mario being a prime example. It has grown continually since its heyday on arcade machines, and modern versions such as Braid and Super Meat Boy have continued to push the genre forward.
I knew from an early stage in the project I wanted to create a platform game. Developing in such a simple yet charming genre provides a well-rounded project encompassing many different areas of game design. This includes level design, collision detection, implementation of gravity and the creation of simple gameplay. My first step was to decide on a unique selling point, something that would set my game apart from every other platform game on the market. Puzzle games are something I've always enjoyed and I feel they would fit in well with the platform genre.

To begin with, I analysed other games in the genre and noticed many are based on an element of skill; the challenge being reaching the end of the level without being eliminated by challenging level design or AI elements. Alongside this the player can earn extra points and rewards by collecting items. By adapting these two ideas I can create a puzzle element to the gameplay. Not only will the player have to complete a level and overcome the terrain, they will be required to collect certain items (in this case a particular hat) and utilise them to complete puzzles.

The idea of collecting hats came from my experience playing other games where collectables are a main part of the gameplay. If the collectable serves no plot purpose, a player is far less likely to go out of their way to collect it and will see almost no direct gain in doing so. By using hats as a key feature in the game it also adds a unique aspect to the gameplay. Each hat collected will change the characters attributes slightly allowing them to jump higher or move faster. This leads onto level progression as the player will have to select which hat to wear to solve a particular puzzle rather than being spoon fed on one hat per level.

I chose a target audience of 16-35 year olds as I feel this is the age range that should still enjoy playing simple enough games, but will have enough intelligence to solve the puzzles I present throughout the game. I intend to target indie gamers in particular as similar games have proven to succeed in this market. By selecting the PC as my release platform it opens up channels like the 'HumbleBundle' should I chose to release the game in the future.

3.0 - Hardware and Software Decisions

3.1 - Hardware

During the initial design process I looked at a series of hardware platforms, each with their own advantages and disadvantages. Hats is predominantly aimed at the indie gamer market, those who are interested in small but creative games that provide the player with a challenge in the later levels. This leads to a hardware platform that should reflect this, for example the PC, Android and XBLA platforms.

Android

A major plus for the android system is its accessibility to such a large market, not just gamers. Android is one of the largest phone platforms in the western world and has an already well established app store that is simple to publish to while reaching a large audience. It also commands the casual gamer market, a market group I believe Hats would work well in. The entire app library is coded in Java, a language I have used before and a language that has a series of easy to use game libraries that could be easily learnt.

The disadvantage with Android being predominantly on mobile devices is that these almost always comes with touch screen input, an input that makes playing platform games difficult. Personally I have found platform games or anything that requires directional input much more enjoyable with a controller or a keyboard input.

XBLA (Xbox Live Arcade)

The XBLA has shown to be a tried and tested platform for releasing indie games to market. It is a well advertised platform with a series of, now well known, indie games such as ‘Super Meat Boy’ and ‘Braid’. Having looked at these games in the past this is a very promising platform that has a lot of positives associated with it. A key component of XBLA is its game pad, a comfortable and easy to use interface that is used by approx. 46 million online users (A. J. Agnello, 2013).
The XBLA uses C# and the XNA library for its games creation. XNA is a very well documented games library, with Microsoft providing a series of tutorials, as well as third party developers/bloggers writing a number of tutorials showing how to put together a game from scratch.

XBLA does have one major disadvantage when it comes to indie games and that is actually being able to get into the XBLA market. Many developers, Wesley Yin-Poole in particular, have found it much more difficult than it should be to have a game released into the XBLA. The Xbox 360 is also soon to become out dated technology with the release of the Xbox One not too far away and original XBLA games not being transferred across makes the platform relatively irrelevant for new released games. With that said the Xbox 360 is a very popular console and many Xbox 360 owners will continue to use the platform even after the new console is released.

PC

PC gaming is a long established platform having been around 5 or 6 times longer than the other two examples I have looked at. With this age comes a market that is well known and loved by those who use it, and is easily accessible to those who may not consider

themselves gamers. Releasing a PC game also comes with the option of which operating system to market to; either Microsoft Windows, Apple Mac OS or Linux. This means a game can be marketed more effectively based on operating system selection.

PC gaming does have a number of downsides as opposed to consoles. Creating a game only for PC and not consoles isolates the market of console only gamers, a large grouping of people. A large number of PC gamers expect high end graphics and platform games are not really designed to be realistic in appearance. I believe the understated graphics are an expected of the genre and should not put off any otherwise willing players.

Choosing PC as my development platform allowed me to have a much wider choice of programming languages and release methods compared to the other two platforms. It also allows me to switch between gamepad and keyboard input, depending on what the player has to hand.

3.2 - Software

There are many options when selecting a programming language for PC and through this project I was looking to enhance my knowledge in a language I already had experience in rather than trying to learn a completely new language. This limited my choices to C#, Java, C ++ and HTML5. Straight away I could eliminate HTML5 as I was not looking to create a browser based game. During the design process I researched the available graphics and game libraries for each language and compared them to what I thought I would need for the project.

Java is an interesting option due to the highly respectable jMonkeyEngine (JME). Although jMonkeyEngine provides a stable and well documented library to work with I do not feel confident in creating a game in a language I have not used in some time. JME does provide solid cross platform integration meaning I would only have to create the code once and I would be able to compile it for different operating systems. After reading the documentation for JME it is clear the library is built for 3D games and interaction with physics engines and would be too complex for the creation of a platform game.

C# has XNA as a games library, with an extensive set of tutorials and guides for game creation and is geared towards making small event driven games. C# also offers a level of stability as it has been tried and tested by developers and hobbyists so most of the bugs and errors are also well documented. Unfortunately using XNA would limit me to only developing for Microsoft Windows PCs. Microsoft has also stopped developing for XNA so unless it is taken on by another set of developers, updates have stopped.

There are a number of games and graphics libraries built for **C++**, each with their own positive and negative attributes. Allegro and OpenGL are some of the main contenders for 2D games with UnrealEngine and various others available for 3D games. When looking at 2D C++ libraries, I looked to select one that was well documented and had enough tutorials and guides. I could use this to obtain a good handle on the library before having to attempt coding the game. The two main libraries I looked at were Allegro and SFML, with the later being famed for its ease when creating platform-like games.
SFML provided enough support and had clear and concise documentation. This made it simple to get started creating a template game where I could work through tutorials. It is also cross platform allowing me to compile my final game for use on Windows and Linux as well as the Mac OS X native I could build. Contained within the SFML library are enough classes and game specific functions that getting to grips with the library was very straight forward.

4.0 – Graphics and Audio Decisions

4.1 – Graphics

Throughout the game I tried to keep the graphics as minimal as possible to reflect the genre as a whole. The game should be about gameplay and simple graphics allow that to shine through rather than trying to impress the player with the latest technologies. Looking at other platform games that have succeeded it is clear that they have lasted due to their enjoyable gameplay and their understated, often 8-bit, graphics.

The graphics in Hats are set up in a number of different levels, examples of this can be found in the appendix. At the very back is a simple main background layer that gradually slides past the player as they move through the level. By moving the background layer at 1/10 of the speed the player is moving at, the player is given the impression the background is in the distance using only 2D graphics.

On top of the background layer will be the main level itself, the tiles that the player will interact with and think of as the level. This layer is created using a tilemap so it can be changed and edited without affecting the code allowing a player to create and load their own levels. This layer will have a number of elements for the player to interact with such as the floor and walls of a level. Included in this layer will be some of the background textures for the level such as trees or bushes to add the idea of depth to the map. The speed at which this layer passes the player will show the relative move speed of the character.

There is then a layer for items and other objects the player can collect. The items in this layer are again placed using a tile map and will be overlaid above the main level. This layer will be searched using the collision detection class to check if the players character or AI enemy has come in contact with any of the items in this layer. All of the hats the player can collect will be displayed in this layer, ready for the player to collect.

Finally, each of the players' characters and AI will be drawn on their own layers on top of all the others. This means regardless of the other objects on the screen, the player will always be able to view their character. Each of the AI characters will be drawn on their own layer allowing for collision detection with not only the player but other map elements. The starting point for the AI and the player will be hard coded, so although map creation is possible for the user, they will not be able to manipulate the placement of none player characters (NPCs).

Once I had planned out on which layer each texture would be drawn I have to find some graphics to use on my game. Initially I wanted to create all of the graphics myself, having been taught photoshop in the past I was confident I would have the ability to draw everything myself. Unfortunately time was an issue and I had to go elsewhere for my artwork.
For the player character I found a simple character generator located on a Japanese site famitsu.com (copyright restrictions can be found in section 4.3). This allowed me to edit and customise a series of characters very quickly that I felt would suit my game very well.
Map tiles were provided as part of a set called Platformer Graphics Deluxe (again copyright information can be found in section 4.3). This set had enough map tiles in it for me to create a series of diverse levels with relative ease.
All of the hats and AI were drawn by myself as I could not find appropriate artwork elsewhere.

4.2 – Audio

Audio plays a major part in any game. It warns the player of up coming danger, rewards them for picking up an item, changes their mood and absorbs them into the atmosphere of the game world. Like the graphics in Hats I intend to keep the audio as simple as possible, using sparingly only to enhance the gameplay. By having a simple 8-bit backing track I can keep the users attention, more focused on the game. When selecting a backing track it is important to consider looping i.e. how many times a track will be played through. If a track has a series of very recognisable points, the player will begin to notice these as the track is looped.

In conjunction with the ambient music I intend to include a series of audio cues to Hats, signaling a number of different events. Theses cues will alert the player of any important information they may need to take notice of. They can also be used to alert the player of up coming danger. A number of cues I intend to include are:

- Enemy approaching
- Player death
- Enemy death
- Collection of hat
- Collection of jewel
- Completion of the level

By using all of these audio cues the player can be made aware of what is happening on screen without having to lose concentration from the on screen character.

4.3 – Copyright Restrictions

As with all content made by someone else, most of the graphics and audio in the game come with some form of copyright restrictions. I have made sure to select images and audio files that adhere to the creative commons attribution license or a derivative thereof. This means the work is free to use as long as the original creator is given the appropriate acknowledgement. A list of acknowledgements can be found below.

Character sprites are from the famitsu generator : http://www.famitsu.com/freegame/tool/chibi/index2.html

Level tiles are made by Kenney Vleugels :
http://www.kenney.nl

Audio files are from New Grounds:
http://www.newgrounds.com/audio/browse/tag/8_bit/tag/8_bit

5.0 – Game Code Structure

MVC

To create the main structure of my code I am using the MVC (model, view and controller) design pattern. To organise the code, this pattern uses three base classes;

View – this class is responsible for drawing the GUI and updating the interface the user will see. This class does not have to deal with any of the functions of the game other than what is going to be drawn to the screen. A program may have more than one view class depending on what needs to be drawn to the screen.

Controller – the controller class is responsible for running the main functions of the game. This class will hold the main game loop which will make calls out to other classes for information and will then pass the objects across to the view class to draw the object to the screen if necessary.

Model – the model class is actually a series of classes that contain all of the data processing functions for the game. These classes are initially separated by deciding which objects will appear in the game, and are then refined by introducing design patterns to make object creation easier.

Class Discovery

Initial class discovery is completed by moving through the various objects in the game and picking out important nouns or verbs, anything in the game that could require an object or process of its own (a list of potential classes can be found in the appendix). This is a very simple process for acquiring classes, the classes can then be adapted to fit with MVC and other design patterns. I looked at a number of design patterns before deciding on the singleton and state design patterns. These include iterator and the template method.

The iterator design pattern could have been used in collision detection to move through the array holding the map tiles. It would have allowed me to add a simple checkNext() method to check through each of the separate tile maps containing the layers. On reflection I decided this approach was too complex for simply moving through a series of arrays when the length of the array is already known.

The template method could have been used to add extra functionality to a player upon collecting a hat. The template method would have made it easy to adjust the attributes of each character depending on the hat they were wearing. I chose not to use the template method as it was more suited to adding functions to a class rather than adjusting or overwriting existing functions.

6.0 – Design Patterns

6.1 – Singleton

The singleton design pattern can be used to create a single instance of a class that is globally accessible at any time. The same effect can be achieved using a static class but a static class would be initialised before the main function. This implementation has the potential to cause errors if the static class relies on information held in other parts of the program. I have chosen to use the singleton design pattern to create my menu class. By its definition the menu should be accessible at any point in the game and there will only ever be one instance of it. The menu needs to be initialised after the main function as it will have to be drawn to the window that is created at the start of the main function and will have to pass function calls to the constructor that would not yet exist if it were a static method.

6.2 – State Pattern

Throughout the game the state of the player will change depending on whether or not they are wearing a hat and which hat they are wearing. Rather then creating a class for each of these and having to create a new object every time the player changes state, the state design pattern can be implemented. The state design pattern consists of a main class which will hold all the regular attributes and functions, a connector class which provides the transition between the states, and a class for each of the states with overriding functions or attributes. Implementing this pattern allows the programmer to encapsulate all of the functions and state pointers into the relevant classes, rather than having one monolithic set of if statements for each state.

In relation to Hats, the state pattern will be used to change the players behaviour depending on whether they are wearing a hat. The main context class will be the player class with noHat, bowlerHat, topHat and fedora being the separate states the player can be in. Each one of these states effects how the player moves and overwrites a set of functions. Within each state is a pointer to the next state so the player will have to cycle to the next hat in their inventory whenever they are looking to swap hats.

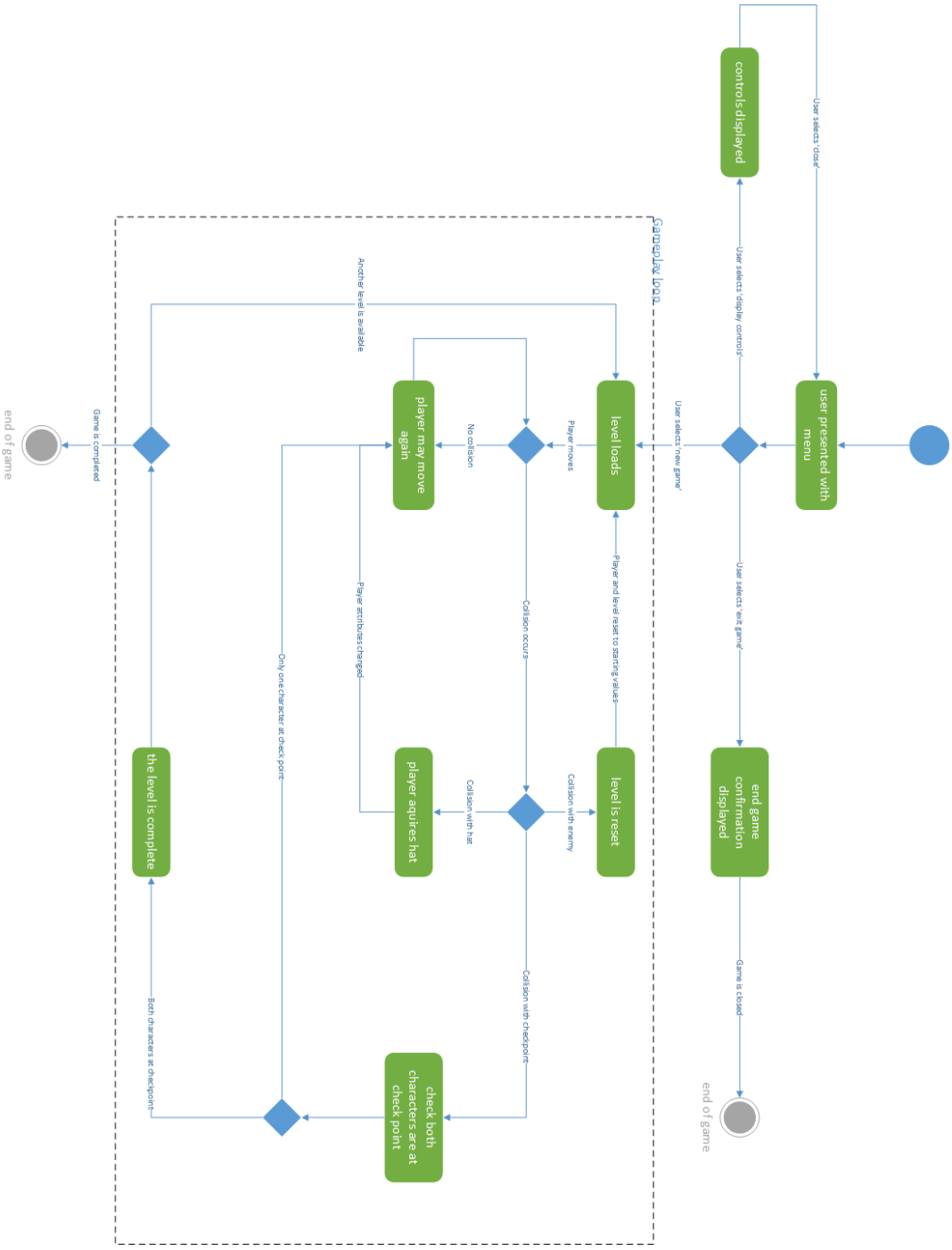
6.3 – Factory Pattern

The factory pattern can be used when the game needs to create a series of similar objects. It consists of a main factory class which passes the creation of an object to a number of smaller classes depending on the type of object that needs to be created. The object will then be able to interact with the interface that initially requested its creation.

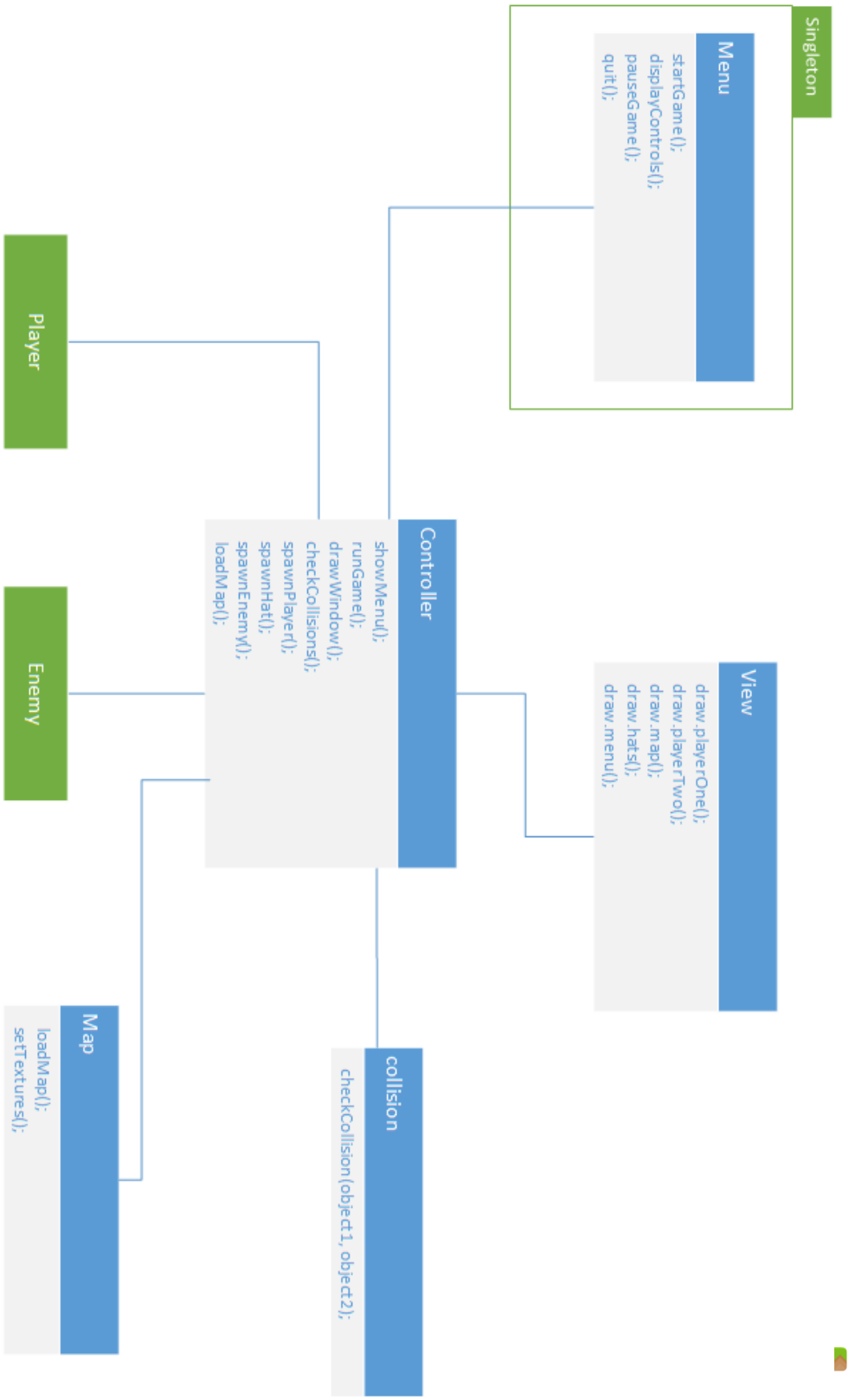
Hats will utilise the factory pattern to create a series of AI enemies of different types. There will be 3 different enemy types that will be randomly cycled through, making the game more interesting and challenging for the player. With roughly two enemies per level there will be around 9 different options for each level, made possible by using the factory pattern.

7.0 – UML Diagrams

7.1 – Simple Activity Diagram



7.2 – Class Diagram



7.3 – Class Justification

View – The view class is used to draw all of the game objects to the screen. The main game objects are passed in from the controller class and the view class simply invokes the draw.Window(object) function contained within the SFML library.

Controller – The controller class is responsible for running the main game loop. It will contain the main function for the program and will make calls to the other classes to create the objects in the game. It also passes these objects to the view class to be drawn to the screen.

Menu – The menu class will be the first screen introduced to the player when the game is started. Only one instance of the menu will exist and will be accessible at all times hence its state as a singleton.

Collision – Collision will be used to test two objects for collisions and will return a value based on whether a collision has occurred or not.

Map – Map holds all of the information about the map as a series of arrays. These arrays can be passed to the controller then on to view to be drawn to the screen.

Enemy – The enemy packet is a series of classes in the factory design pattern used to spawn a series of enemies on to the screen, These objects are controlled by the controller.

Player – The player packet contains the classes to create two player characters and one instance of the inventory class to hold any hats the players have picked up.

N.B. Note a full expanded class diagram can be found in the appendix.

8.0 – Game Specific Features

8.1 - Collision Detection

Collision detection is an important part of any platform game. It allows the creator to code obstacles, levels, enemies, collectables and anything else a players character might run into on their way through the level. Collision detection has allowed me to create simple levels built from a series of square tiles and place various check points along the level for the player to trigger. This is implemented by creating a bounding box around the players character and on each draw() checking to see if the box has intersected with another bounding box; be that another character, a level tile or an enemy. Collision detection is done at every draw stage rather than on player movement as it has to be done for any other moveable object that may have changed position in the last frame.

Collision detection will be implemented in my game by having a class that receives a sprite as an object, checks the bounding box of that object against other objects and map tiles in the same x-coordinate, then returns a value based on the object it has collided with. This value can then be used to deduct health from the object, stop the object from moving in a certain direction, add the collided item to the objects inventory etc. Limiting the collision detection to other sprites and maps tiles around the same area in the x-axis means it should not take very long to check for collisions.

When calculating the bounding boxes for collision detection it is important to give priority to the player. A player may become aggravated if a bounding box is larger than an object and a collision is triggered before there is contact appearing on the screen. Similarly if a bounding box is too small and the player is trying to trigger an event, the target area may be too difficult to hit. When assigning bounding boxes it is important to consider these two attributes to keep the player entertained.

8.2 - Artificial Intelligence

Alongside collision detection Hats contains a number of very simple AI enemies that move back and forward on the level until the player is within range. Whenever a player gets close enough to be noticed by the AI, the enemy then progresses towards the player until it is either killed or the player is out of range. This can be achieved simply by comparing the player's position to the position of the enemy. If the player's position subtracted from the enemy's position is between 1 and 25, the enemy will move left until it collides with the player. Likewise if the difference in positions is between -1 and -25 the enemy will move right until it collides with the player. Allowing the enemy to track the player means the enemy can move at a slower pace, so the player can outrun them, but also keeps them competitive throughout the game.

Although very simple, this addition of slight AI will make the game more challenging and therefore more rewarding to the player.

8.3 – Notable Additions

Puzzle Design

Puzzles are a main feature in the game and it is important to design them correctly in order to keep the player entertained. If the puzzles are too easy the player may become bored whereas if they are too difficult the player may be put off from continuing the game. Finding this balance is a difficult component in level design.
When trying to design the puzzles for Hats I felt the early puzzles in the game can be used to teach the player the basic mechanics of the game and controls need to move the characters Later in the game once the player has mastered the controls and has an understanding of the game the puzzles can start to increase in difficulty. Most of the puzzles in the game are going to rely on the collection of Hats spread throughout the game and how the player choses to use them to advance his or her character. As a character can only wear one hat at a given point it is possible to design puzzles that require the player to equip a number of hats in a certain order to complete the challenge.

The puzzles also serve as a form of player direction, showing the player the way to the main objective. Although this is not necessary in a platform game, due to the gameplay being very linear, the location of a puzzle can be used to direct a player to the location of another hat.

Multiplayer

Hats will allow the player to control two characters in the game window, moving only one at a time. This leaves the option of introducing multiplayer into the game by allowing a second player to control the second character on the screen. By allowing a player to play alongside their friend increases the playability of the game. The process of coding the interactions should be straightforward due to the controls for the second player mimicking that of the first character.

9.0 – Time Management Plan

When designing and building a full game in such a short time frame, time management becomes very important. It allows the programmer to split the tasks into smaller, more manageable chunks and plan for each of them. By splitting the tasks into these chucks it means the programmer to assign an importance level to each part and manage how much time will be spent on each part.

My time management plan (gantt chart can be found in the appendix) reflects the importance of each task and the order in which they need to be completed.

9.1 - Contingency Plan

Looking at the gantt chart for the project it is easy to see all tasks should be completed before the deadline, but in any project there are likely to be complications. It is important that a contingency plan should be laid out in the early stage of the project should time run out before the project can be finished.

Firstly it is important to decide which tasks are the most important and should be completed first, and which tasks could become optional if time runs out. I believe the most important part of the project is to have a level drawn to the screen with a playable character that can detect collisions with the map tiles.

Once the main features of the game are decided upon, the rest of the tasks can be put in order and worked through sequentially. I feel this is reflected in my gantt chart. Alongside prioritising game features I have included 4 days at the end of the project (after playtesting and time for fixing bugs) should any of the other tasks run over their allotted time.

10.0 - Bibliography

Digital Trends, 19/04/2013, Recent results give Microsoft a compelling defence for the always-online Next Xbox, http://www.digitaltrends.com/gaming/xbox-360-sales-and-xbox-live-membership-numbers-give-microsoft-a-compelling-defense-for-the-always-online-next-xbox/, 14/10/13

Euro Gamer, 22/11/2011, Making an XBLA Game: The Inside Story, http://www.eurogamer.net/articles/2011-11-21-making-an-xbla-game-the-inside-story-article, 14/10/13

Freeman, Elisabeth, **Freeman**, Eric et al, 2004, Head First Design Patterns, O'Reilly Media, USA

Gamasutra, Unknown, Building Buzz for Indie Games, http://www.gamasutra.com/view/feature/132506/building_buzz_for_indie_games.php, 15/10/13

PC Gamer, 3/11/2012, The Indies' Guide to Game Making, http://www.pcgamer.com/2012/11/03/the-indies-guide-to-game-making/, 15/10/13

Source Making, Unknown, Design Patterns, http://sourcemaking.com/design_patterns, 14/10/13

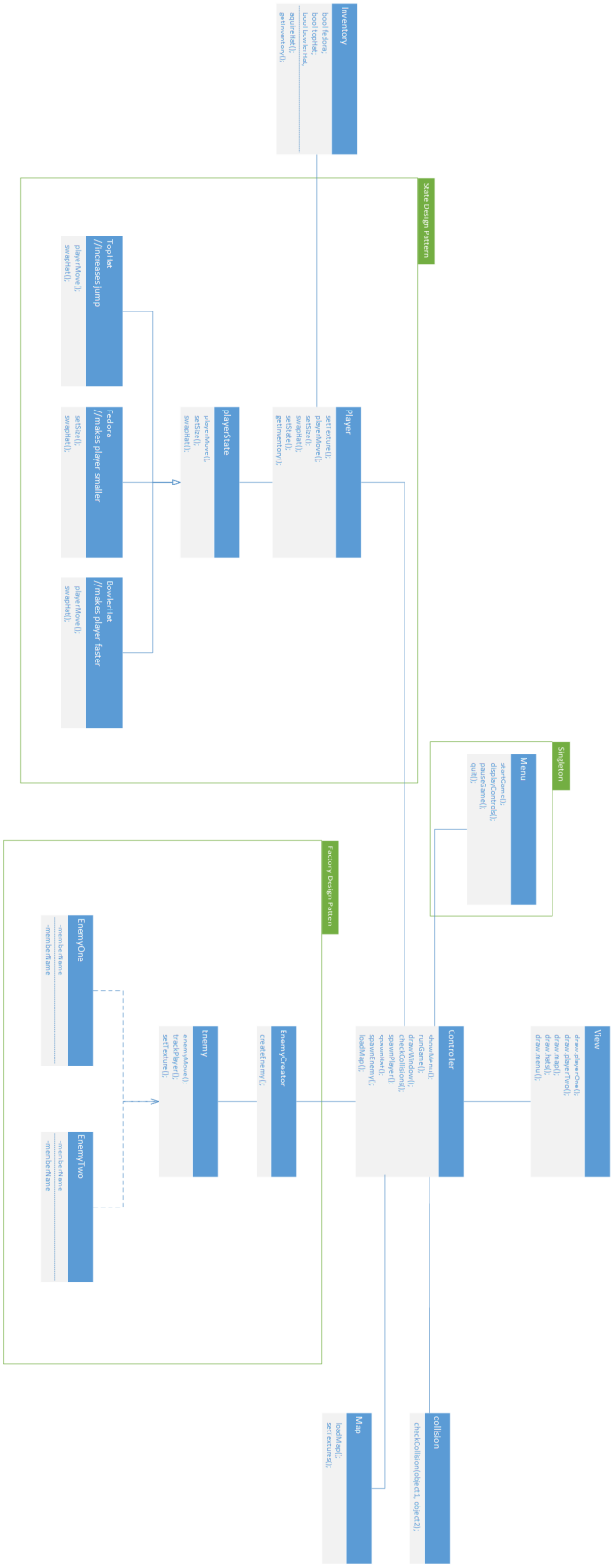
Thompson, J. 2012, The Computer Game Design Course, Thames & Hudson, London

Special Thanks to

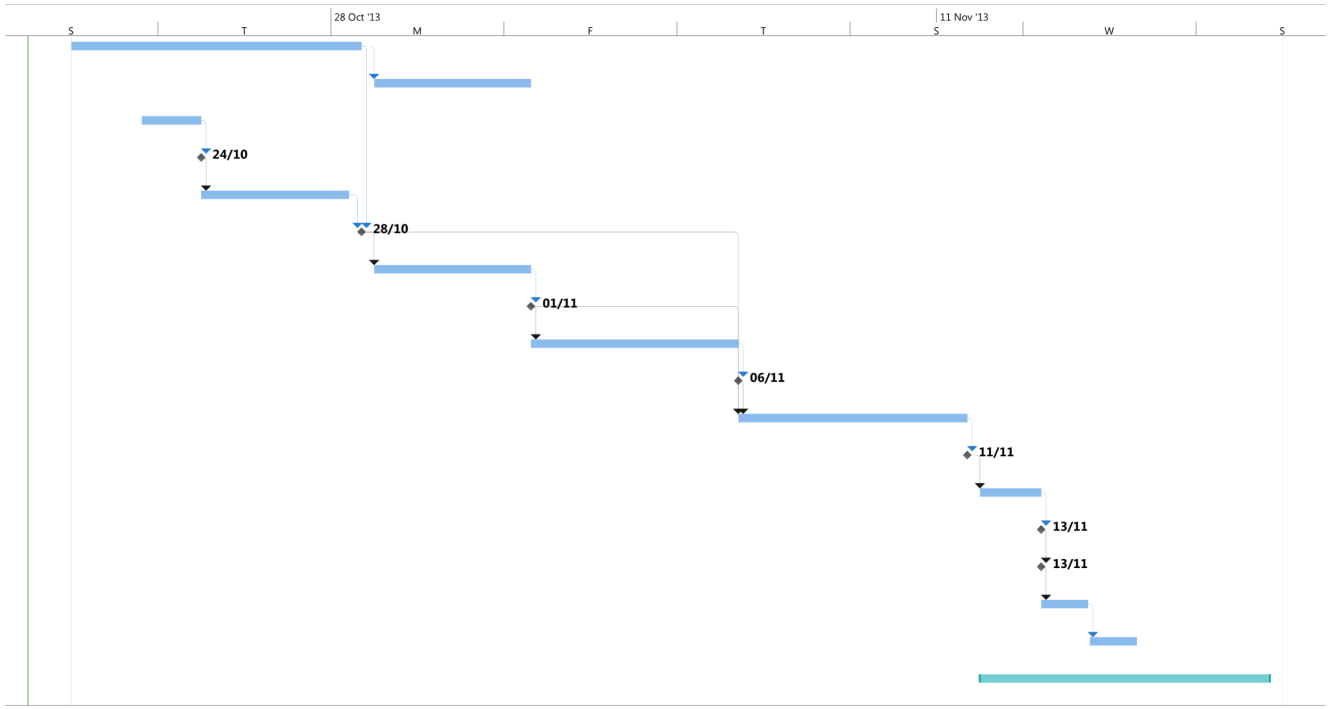
The Famitsu Generator:
http://www.famitsu.com/freegame/tool/chibi/index2.html

Kenney Vleugels :
http://www.kenney.nl

11.0 – Appendix

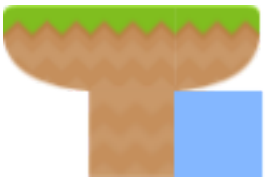


ID	Task Mode	Task Name	Duration	Start
1		Complete Artwork	4 days	Tue 22/10/13
2		Design Levels	3 days	Tue 29/10/13
3		Template Classes	1 day	Wed 23/10/13
4		Test Class interactions	0 days	Thu 24/10/13
5		Complete Map Class	1 day	Fri 25/10/13
6		Test Map Implementation	0 days	Mon 28/10/13
7		Complete Player Class	3 days	Tue 29/10/13
8		Test Player Class	0 days	Fri 01/11/13
9		Complete Enemy Class	2 days	Fri 01/11/13
10		Test Enemy Class	0 days	Wed 06/11/13
11		Complete Collision Detection	3 days	Wed 06/11/13
12		Test Collision Detection	0 days	Mon 11/11/13
13		Implement Hats	1 day	Tue 12/11/13
14		Check Puzzles	0 days	Wed 13/11/13
15		Check Level Validation	0 days	Wed 13/11/13
16		Playtest Game	1 day	Wed 13/11/13
17		Fix Bugs	1 day	Thu 14/11/13
18		Complete Game Design Report	4 days	Tue 12/11/13


28 Oct '13
11 Nov '13
S
T
M
F
T
S
W
S
24/10
28/10
01/11
06/11
11/11
13/11
13/11

Player 2 Sprite

Player 1 sprite



Potential Level Tiles