<u>CMP208 Coursework Report</u>

<u>Student Name:</u> Jamie Mulvihill

<u>Student Number:</u> 1703169

<u>The Purpose</u>

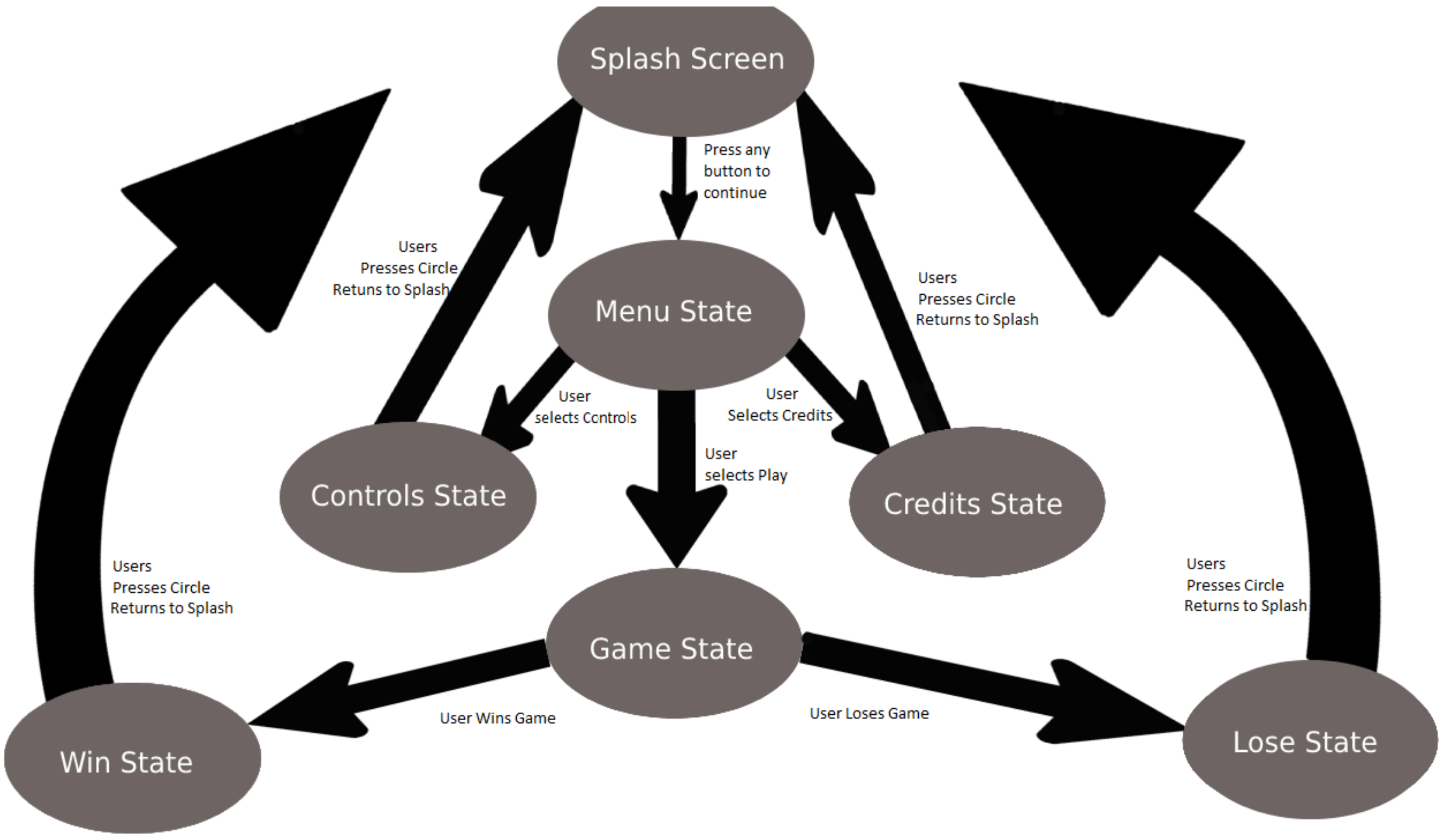
The purpose of my application was to try and recreate a game I had played and enjoyed called "Angry Birds" for the PlayStation Vita using the GEF gaming education framework and the box2d library but also put my own personal spin on the game. The original game sees players launch a bird from a slingshot to hit pigs who have stolen their eggs, in my version of the game the basic idea is the same (launching an animal out of a slingshot), however, now the player launches a pig, as the game follows a group of pigs who are sick of being ridiculed and laughed at for not being able to fly, taking offense to the common derogatory phrase "when pigs can fly". The pigs decide the most effective way to start changing the rest of the world's perception of them is the rid the world any pigs who don't support them and follow these same ideals. Now the player must launch a pig at and destroy all the other pigs in the world. On each level there would be a set number of pigs to destroy but for the purpose of this application prototype there is only one level. The User would then receive a score and star rating based on their performance. The application makes use of the PlayStations Vitas touch screen to read input from the player on what forces to apply. The application also makes use of the box2d library to create physics simulations, this allows the pigs to be launched by applying the relative forces and have gravity acting on the bodies of each pig. The box2d library also allows for the collisions between bodies to be identified and the relevant gameplay to occur based on these collisions. Using the GEF gaming education framework, a number of different sprite animations and sound effects were added to the application to try give the game a more polished feel.

Application Design

During the development process of the application a number of different design choices were made to create an appropriate organisation and structure. These ranged from the implementation of a state machine to the creation of a GameObject super class and the design of a generalized function for sprite animations.

State Machine

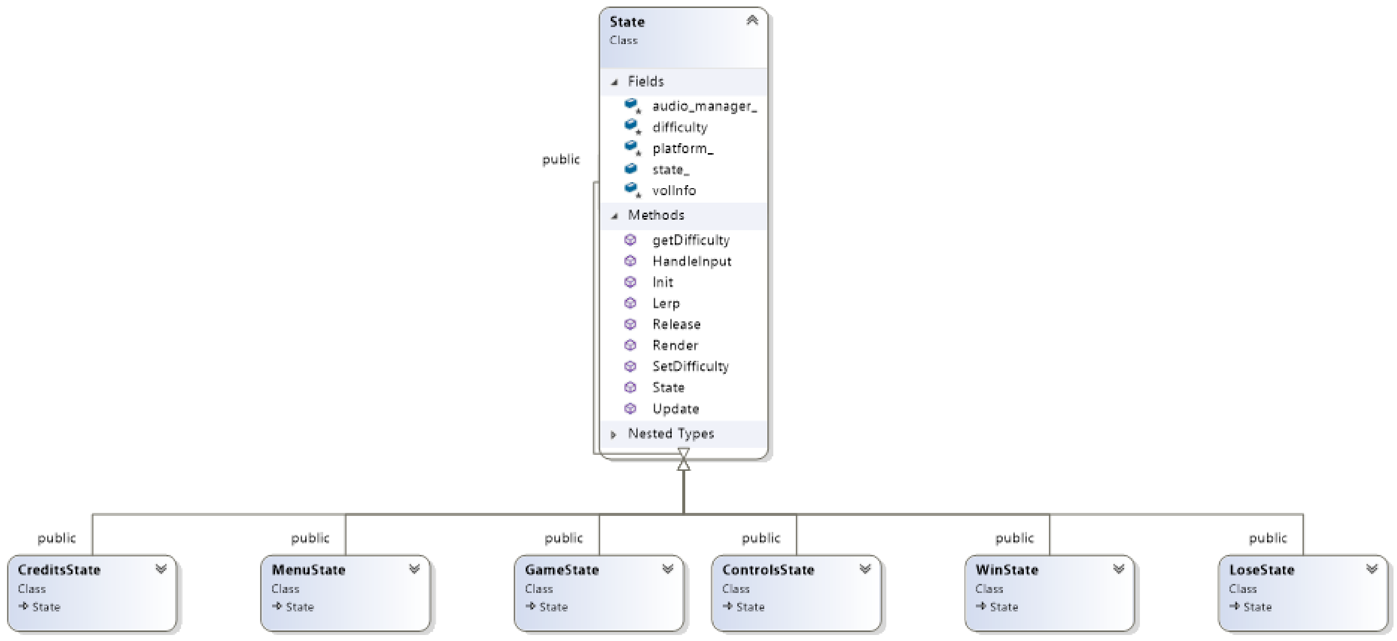
State Machine Diagram


Splash Screen
Press any button to continue
Menu State
Users Presses Circle Retuns to Splash
Users Presses Circle Returns to Splash
User selects Controls
User Selects Credits
User selects Play
Controls State
Credits State
Game State
Users Presses Circle Returns to Splash
Users Presses Circle Returns to Splash
User Wins Game
User Loses Game
Win State
Lose State

The applications design utilised a state machine for switching between the relevant states when needed. Different states were created for the Splash Screen, Menu, Controls, Game State, Credits, Win and Lose state. This meant the application could start in a Splash Screen state and the user could dictate what the application would do next. In the SceneApp class of the Application the Init, Render, Update and Release functions could be called for whichever state the Application was currently in.

To implement the State Machine designed for the application a Super Class called State was created. This State class had a number of Virtual functions that would then be overridden by its child classes to carry out that specific states in-game functionality. These overridden functions were the Init function to initialize all the variables and values used in that state, the Update function to perform whatever each state has to do every frame, the Render function to draw the sprites or 3d meshes for that state and lastly the Release function to clean-up and delete pointers when changing to another state.

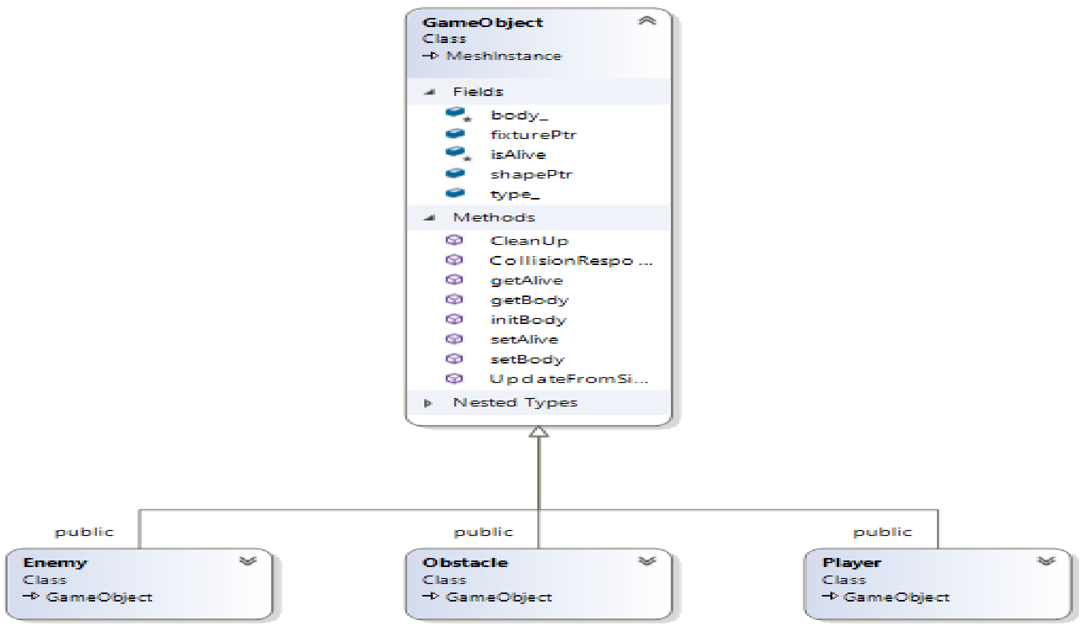
To implement the changing from one state to another the application made use of static functions and a State pointer called currentState. A static function called SetState that had a State pointer as a parameter was created in the SceneApp class, as this function was static it could be called in any other class that had access to SceneApp. The SetState function would call the currentState Release function, then set currentState to the state pointer passed in the function call as a parameter and then call the new currentState Init function.



Game Objects

The application needed to have a number of different game objects. A GameObject class was therefore created. This class would make use of inheritance and composition programming techniques in its design. It would inherit from the Gef MeshInstance class meaning now all instances of the GameObject class were a MeshInstance and could be rendered easily using the Gef 3d Renderer. The GameObject class would also make use of composition to give each instance of GameObject its own Box2d body. The GameObject class could then have a function to initialize the values of the Box2d Body, translate its mesh to the Box2d body and Child classes could then be created from the GameObject class inheriting all properties of the GameObject but would be able to create their own functions and properties that would have no impact on any of the other child of the GameObject class. Classes were made for the Enemy, Obstacle and Player objects to inherit from GameObject.

UML of GameObject Super Class Inheritance



<u>Game Assets</u>

To design the level of the game, a number of 3d assets were created and textured using the blender 3d design software. An Asset class was then created in the application that would convert these 3d assets into meshes that could be used and rendered using the Gef framework. Meshes were made for an instance of the Player, Enemy and Obstacle classes as well as the game background and the slingshot used to launch the pigs.

<u>Sprites</u>

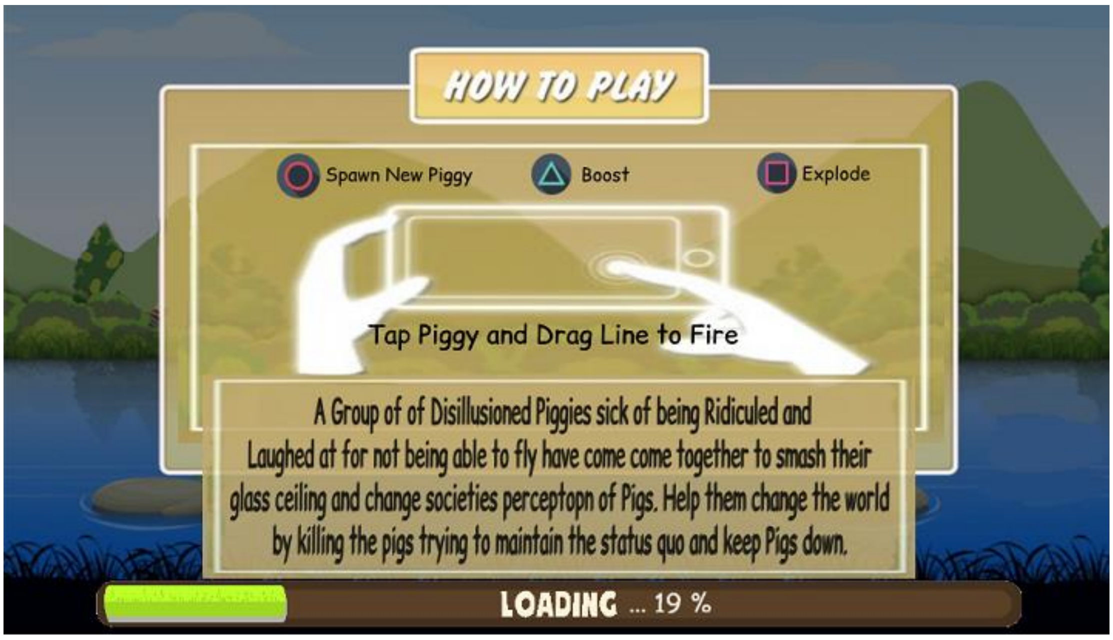
The level also made use of the Sprite Renderer in a number of different ways. To launch the pig from the slingshot, the user had to tap the pig that was loaded in the slingshot and drag their finger back creating a line to show the launch trajectory of the pig, to show this line on screen for the user to see, a texture was created and assigned to a sprite. The sprites position, width and rotation would then be adjusted according to the users touch.

The sprite renderer was also used in the level to display the number of pigs the user had left to launch at the enemy pigs. This was done by creating a GUI class. The GUI class would use an integer value to render the number of pigs left to launch on screen. To give the GUI a bit more life the pigs were programmed to bounce and placed against a wooden back drop alongside the users current score.

<u>Loading Screen</u>

As the applications Game State began to grow, the time spent waiting between the Menu State and the Game State launching increased dramatically. It was decided to add a loading screen to prevent an awkward and boring wait for the user. The initializing of the variables and properties of the Game State were split into three different functions. The progress of these functions was then displayed with a loading bar along with the controls of the game to keep the user engaged whilst the applications Game State loaded.

Screenshot of Loading Screen



A Group of of Disillusioned Piggies sick of being Ridiculed and Laughed at for not being able to fly have come come together to smash their glass ceiling and change societies perceptopn of Pigs. Help them change the world by killing the pigs trying to maintain the status quo and keep Pigs down.

Animations

As the application needed animations it was decided that the best way of achieving the desired result was to create an array of Sprites, give each sprite a frame of the animation and then to play back animation in game the application would loop through the array in render function to render the animation on screen. Each frame of the animation would have to be named with common stem string and number value representing its place in the order of the animation i.e. “Frame01, Frame02, Frame03…etc.”, This naming design would allow for easy looping through the frames and creating textures accordingly. As the application would have use of multiple animations, a function was created that took the needed parameters to generalise the problem, this made it easier to make multiple new animations.

Pseudocode and Code snippet for animation function

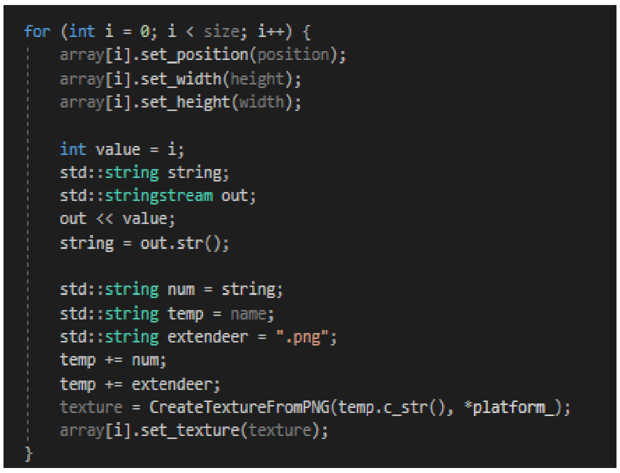
```
Get the total number of frames in the animation
Create an array of Sprites the size of the total number of frames

loop through the number of frames

    for each element of the array set a postion, height and width

    convert the loops iteration vlue to a string
    get the common stem string for each frame of the animation
    create a string for the file extension
    add these strings together to make one total string

    create a texture from the total strings
    set the texture of the array element to the created texture
```



```
for (int i = 0; i < size; i++) {
    array[i].set_position(position);
    array[i].set_width(height);
    array[i].set_height(width);

    int value = i;
    std::string string;
    std::stringstream out;
    out << value;
    string = out.str();

    std::string num = string;
    std::string temp = name;
    std::string extendeer = ".png";
    temp += num;
    temp += extendeer;
    texture = CreateTextureFromPNG(temp.c_str(), *platform_);
    array[i].set_texture(texture);
}
```

Sound

To try and increase the user's experience whilst playing the Game, music and sound effects were added. While the music and sound effects do increase the experience during the Splash Screen and Menu State, it is in the Game State that the impact of the sounds effects was really felt by the user. By adding a sound effect to the user selecting a pig to launch, a sound for the launch itself and a sound of the pig squealing as it is flying across the world, the application felt much more engaging; further sounds were then added for the different gameplay mechanics. When the user hits the explosion button, an explosion sound effect will play, when the user hits the boost button, a fart sound effect will play. If the pig collided with and destroyed an obstacle, a wood smashing sound effect would play and if an enemy was collided with a score increase ding sound effect would play as the score increased.

Credits

To create the application a number of different assets were used, this varied from sound and textures to software and libraries, therefore it was decided to add a Credits State. In this state, a sprite with the information about the individual assets was created and rendered to the screen using the sprite renderer. These sprites would then scroll up the screen until they reached a max height value and would then disappear from the screen.

Scoring & Difficulty

To give the game a more goal oriented gameplay, a score system was added. The scoring was based on the collisions between the User pigs and the enemy pigs. When a User launched pig collided with an enemy pig, the enemy pig would be destroyed and the score would be increased by 1,000. When all the enemy pigs on a level were the destroyed the level would be completed.

To add a varying degree of difficulty to the game, it was decided that three different choices would be added. When the player selected Play from the Menu State, three new button selections would render to the screen. These new button choices would be Easy, Regular and Hard. The User selected difficulty would then be stored using a static enum variable. When the Game State was then initialized, a static function called GetDifficulty would be called that would return the stored user selected difficulty. The GUI and gameplay could then be initialized accordingly. On the easy difficulty the user would have 5 pigs to launch and complete the level. When the level is completed the users score would be increased by 10,000 for each pig left to launch. On the Regular difficulty the user would have 4 pigs to launch and get 15,000 for each pig left unlaunched. On the Hard difficulty the User would have 3 pigs to launch and get 20,000 for each pig left.

In the Win State, values were declared for One Star, Two Star and Three Star performances by the User. If the user scored higher than 3,000, they received a One Star rating, if user scored higher than 23,000 a two star rating was given and if the user scored over 43,000 a three star rating was given. The users score would be rendered to the screen and the values would be lerped until the final score was reached. As the score increased the Star rating sprite would change based on the users score and sound effects would play for each star change.

Techniques

Gameplay

The main gameplay functionality of the application came from reading the users touch input. Each new user touch position is checked against the position of a bounding circle. If the touch position is within the circle, the pig has been selected and the user can now try to launch it, if it is not, the user can pan the camera to change what they are looking at and get a better view of the targets. To launch the pig, the user must draw a line by dragging their finger along the screen, a vector is created based on the user's touch's starting position and the users touch current position. A b2vec2 was then created called force, the x value of force is equal to the distance from the touch's start x position to the touch's released x position and the y value of force was set to distance from the touch's start y position to the touch's released y position.

Once the pig was launched all the User could do now was wait and see what the outcome of the launch would be. To increase the interactivity with the User, further gameplay features were added. These features were boost and explosion mechanics. By pressing the triangle button after launching a pig, the force being applied to the pig would be modified, the sprite animation for the boost gameplay would play and the boost sound effect would be played. The force would be multiplied positively in the x axis and negatively in the y axis. This gave the pig a more purposeful trajectory and potentially causing more damage in game as well as giving the user more control of the outcome of each launch. The Explosion mechanic was implemented similarly. By pressing the square button after a pig was launched, the pig would stop at its current position, the explosion sprite animation would play, a sound effect would play and its hit box would increase in size. This was done by incrementing the radius of the pig's box2d body. As the pig's body would increase in size and come into contact with more obstacles, the Obstacles collision response function would check if the pig had exploded and apply a force based on that obstacles position relative to the pig's position.

Collisions

As most of the gameplay triggered from collisions, a generalized function was created that would handle the collisions between any two GameObject bodies. The function would check the B2worlds contact list and create an int value for the number of bodies that were touching. The function would then loop a number of times equal to the amount of values in the contact list. Each iteration of the loop would check what the body in the contact list was colliding with, cast those colliding bodies to GameObject pointers and lastly call each bodies Collison Response function for the appropriate gameplay to occur.

To stop different GameObjects from colliding with each other the application made use of a technique called Collision Filtering. This was implemented by setting values when creating a GameObjects body, the values could then be used to essentially state what each GameObject was and what each GameObject collided with. When the bodies were created, a fixture was given to each body, these fixtures can store values for categoryBits and maskBits. By assigning these values to each GameObject different collisions could occur and some GameObjects would not collide at all.

New Player

As the applications gameplay centred on launching a pig out of a slingshot across a world, where the pig could explode or would smash into other pigs and obstacles, the application needed to spawn a new Player pig for the game to continue whenever each Player pig died. This was done by creating a vector of Player pointers and a Player pointer called activePlayer. When the Game State is first initialized a Player created and added to the vector. The activePlayer is then set to be the last element in the vector using the Back() function of the vector. Now when the Player was killed it could be popped from the vector and deleted whilst a new instance of Player could be created and added to the vector. The activePlayer pointer could then be assigned to this new instance of player and the game could continue onwards until the maximum number of players is reached based of the difficulty setting selected.

File Reading

As the application would need a number of instances of both the Obstacle and Enemy classes, a manager class was made for both, i.e. EnemyManager class and ObstacleManager class. The EnemyManager class has a list of Enemy objects. To populate the list a function was created called InitEnemies. In this function a loop was created to iterate through a number of positions stored in a vector of b2Vec2s and assign an enemy to each position. To get these positions a function was created to read values from a .txt file, the function would read each element of the file and return it as a char pointer. A separate function was then created to convert the char pointer to a string and loop through each element of the string. The elements of the string would then be converted into floats using their ASCII values and stored in a vector. Finally this vector would then be looped through converting every two floats into a b2Vec2 and storing it in the vector for positions.

Deleting

Each instance of the Enemy class contained an Enemy List iterator. This meant that when the list was populated by the EnemyManager each Enemy's iterator could be assigned to the current list positions iterator. This ability to assign an iterator meant that when a Player collided with an Enemy, that specific Enemy could be deleted instantly instead of looping through the list and checking if any of the Enemy objects had been collided with, drastically improving the performance of the application. As this same process was repeated for the Obstacles and ObstacleManager, the performance of the application was improved further and meant that any size level could be created as this solution for deletion would scale along with the new level.

Multi-Threading

As the Game State class grew and more and more Assets were added, the Game States initializing time was also increasing. To speed up the process of initializing all the necessary Assets, a technique called Multi-Threading was implemented. Using Multi-Threading the work could be split up and spread across the PlayStation Vitas four CPU cores using a simple Fork-Join Design Pattern. Each of the four cores would be a given a set amount of work and when all the cores were finished they would join back together with the Main Thread and the Application could continue on. As the Main thread no longer had to sequential iterate through each of the individual Asset and initialize it, the total time spent initializing was decreased dramatically.

Lerping

After adding the GUI elements of the Game State to show the number of pigs the user had left to launch, it was decided that to make the GUI pigs stand out and catch the eye of the user, it would be better for the GUI pigs to bounce up and down giving the impression they were excited to be launched. To achieve this a simple mathematical technique known as Lerping (Linear Interpolation) was utilised. A Lerping function was created that would take three floats as parameters. The first two would be the start and end point of the calculation, whilst the third would be the time it would take to get from the start to the end point. This Lerping function could then be used to change the

GUI pigs Y position creating a bouncing effect. After this successful implementation of the Lerping function, it was then decided that the Lerping function would be used again to change the GUI pig's height and width. This would create a simple stretch and squeeze illusion that added to the bouncing effect.

Data Oriented Design

Data Oriented design is a method used to optimize a programs performance by making efficient use of the CPU Cache. To achieve this, further emphasis is put on the way data is stored instead of on Object Orientation. When a piece of memory is accessed the CPU gathers the data of the adjacent memory addresses and if the next piece of memory that is needed has already been gathered the overall performance of the program will be improved as the CPU does not have to return to main memory/RAM to retrieve the next piece. This is known as a cache hit. The more Cache hits a program has the further the performance will be improved. If the next piece of memory that needs to be accessed has not been gathered, the CPU must go back, access the RAM and find the newly required memory address. This is known as a Cache miss and is an expensive waste of time to the overall performance of a program. By minimizing the use of pointers and sacrificing some of the features of Object Oriented programming the number of Cache hits can be increased, the number of Cache hits can be decreased and the overall performance can be dramatically improved.

To ensure the maximum amount of Cache hits, data needs to make use of contiguous memory, i.e. consecutive memory addresses. In order to do this the use of Arrays as a collection data structure is best as it is stored on the stack, whereas, a vector or a list makes use of dynamically allocated memory and is stored on the heap. By having a vector or list of pointers, when one element is accessed, the next elements address could be anywhere else in main memory and will result in the CPU spending valuable time searching for the next element. This time spent searching then increases as the size of the vector or list increases.

Following this logic there is a number of changes that could be implemented to the Application to achieve a more Data Oriented Design.

Manager Classes

Both the EnemyManager class and the ObstacleManager class made use of a list of pointers to store each object. Each manger class has a Render and Update function that loops through the list. The Render function checks whether each element is alive and if so to renders it on screen. The Update function checks whether each element is alive and if it is applies physics to its body. As the elements of the list are pointers each time the loop iterates there is a Cache miss occurring and the CPU will stall and wait for the address and data for the next element to be returned. If an Array had been used for the storing each element and the Enemy and Obstacle classes took an int value in their constructers, the index position in the array could have been used when creating each Enemy or Obstacle and would allow for each element to be accessed individually by using its index number. Through making use of an Array the elements would be stored in contiguous memory and the process of looping through each element would be much faster as each elements address would result in a cache hit for the next element. Whilst this design gives a performance boost for the Update and Render functions, it would be less efficient as a method of deletion than the currently implemented one that makes use of list iterators. As the Player class returns the iterator of the

Enemy or Obstacle that needs to be deleted and the deletion from the list can then be done without any looping or need for adjusting the organisation of the list.

Debris

When an Obstacle is destroyed a vector of GameObjects pointers is populated. These GameObjects act as debris to signify that the Obstacle has been broken. As each piece of debris has a body for physics to act on, a function to update their position must be called every frame and a function must be called to check whether each piece must be deleted based on its position. As each piece of debris is a small size in memory and only makes use of two function calls that are updated every frame, it is an ideal candidate for the Data Oriented Design method. A simple switch from vector of GameObjects pointers to an array of GameObjects would mean a cache hit when looping through the debris particles and as GameObject class is size in memory size the CPU may only have to make one trip to main memory/RAM to get access to the whole array.

As the Application designed was quite small and only had one prototype level, these were the only glaring examples found that could make significant use of Data Oriented Design. As more features could be added to the application or in larger projects in general the number of areas this design can be utilised is quite high. Examples of these could be Ai, Animations and particle systems. Particle systems in general seem to be the ideal candidate for a Data Oriented design. By making use of further techniques used in Data Oriented design such as Hot/Cold splitting, where the data within a structure is stored in a way that groups the elements together based on a flag, the performance can be improved further still. An example of this in the Application designed would be to arrange the Array of debris objects with all the alive debris at the start and any dead one at the end. This would increase the speed of looping through the Array by utilising branch prediction.

User Guide

Splash Screen

The application launches in the Splash Screen state where the user can press any button and the application will change to the Menu State.

Menu State

The User must use the Directional Pad to Navigate the Menu and Press the Cross or “X” button to make a selection.

If the Users selects Controls or Credits, the circle or “O” button must be pressed to return out of the relevant state.

If the User Selects Play, they must now select a difficulty to launch the Game State. Using the same controls as the previous selection.

Game State

Once in the Game State, The User must tap on the pig located in the slingshot and drag their finger along the screen, when the User releases their touch the pig will be launched.

Once the pig has been launched the user can press the triangle button for a boost or the square button to explode. If the user is unhappy with their launch or wants to spawn a new pig the circle or "O" button must be pressed.

Win/Lose State

If the User has reached the end of the game and is in the Win or Lose state, the circle or "O" button must be pressed to return to the start of the application. The User does not have to wait for their score to be calculated before returning to the start of the application.

Conclusion

After completing the coursework for this module, I feel like my appreciation for the work that goes into producing a full working game has been increased dramatically. To complete the coursework for this module I had to familiarize myself with a Game design framework called Gef. As the main features of Gef were covered in lectures, the task of learning a whole Game design framework for the first time was not as daunting as it could have been. This module has given me the confidence, base knowledge and understanding of what a game engine is built on to be able to look at other Game Engines in the future and be able to design and create my own Games and Game mechanics.

This coursework afforded me the opportunity to produce a game for the PlayStation Vita console. This was a very exciting experience as it allowed me to implement gameplay features I had never been able to do before. With the PlayStation Vita having the ability to read input from a touch screen I was able to create a game that utilised this feature in its gameplay. This again has given me the confidence to try and integrate touch input in any Games I make going forward. I also learned about Polish and Game feel by completing this modules coursework. Sound and Audio was one of the best examples of this. By adding audio to the game an extra dimension of interactivity was added that gave the overall game a much more complete feeling. As one of the coursework specifications was to make use of the Box2d library, I was able to integrate physics simulations into my Game. This meant more complex gameplay could be implemented with ease as the library would have its own set of functions to carry out the work and necessary behaviours in game.

In undertaking this coursework I learned a number of different techniques to improve the performance of my games. I learned how to implement a method of deletion that made use of list iterators to access the specific object for deletion as opposed to looping through a whole collection of objects every time something needs to be deleted, a method of Multi-threading with a Fork-Join design pattern was used to reduce loading times and by following some of the Data Oriented Design techniques to utilise cache memory more efficiently the performance my future games could be improved future.

My goal for this coursework was to create a game similar to one that I myself had played and enjoyed for countless hours but also put my own personal flavour on it, whilst also learning how to use a gaming framework and implement code for a specific games console. On reflection of the completed coursework I feel like I have managed to achieve this goal.