# “Beetle run”

# Technical Design Document

# JACK MILLER

# P11268161

# Game Summary

Beelteerun is a platformer that will run on a PC as an executable. The object of the game is to guide the famous adventurer through a level to collect the magic jewel and reach the exit. There will be a beetle enemy.

# Development Overview

The development will take place using the programming language C++ and will be developed in Visual Studio 2010. The hardware that the game will run on is typical PC hardware, and will run on windows 7 and 8.

The main software that will be used during development will be Visual Studio 2010 and Fruity loops to make some music/sound files.

The game will use the SFML library for drawing sprites and handling audio. It will also create a render window for the game to be displayed.

# Game Mechanics

The main game mechanics involve movement, collisions, an enemy that patrols and chases the player and a level design where the player has to reach a certain point, but will have to fulfil certain requirements to move onto the next level, which for this game, is by collecting the magic jewel.

The main control loop drawing all of the sprites; it gets the input from the player and then updates the screen to redraw the sprites in their new updated positions depending on the game state and user input. There will be a few different states in the game, including actual gameplay, a high score screen, and a splash screen could be implemented by using a state.

# Audio

Audio will be created by a program called fruity loops. Audio and sound effects will also be found using online resources that hold royalty and copyright free sounds. Audio will be implemented using SFML. There will be a main melody which can be described as a ‘background’ sound, and several sound effects used for overall atmosphere. They will also be used to show something to the player, such as when they collect a ‘gem’, a sound cue can indicate to the player that this has indeed been collected. These audio files will be loaded on compile time, and then ran at certain times, depending on what event has just occurred.

# Graphics

The graphics will all be 2D in the game. To load sprites into the game, the SFML/GRAPHICS library will be used. This will handle making an image object, loading the picture of a sprite into the image, and then assigning the sprite the image that’s just been loaded in. These sprites will be assigned to certain objects. For example, for the main player object, the main player sprite will be loaded with that object. Images will be in .PNG format as this allows transparency behind them. Graphics will be animated by having a sprite sheet. The effect of animation is that as a sprite moves, the next image will draw on screen, and then the next will take its place. The graphic will essentially cycle through the images on the sprite screen in order to give the impression of animation and “movement”. Graphics will also be used as a way to do collisions, as collision boxes can be based around the size and position of different graphics.

# Artificial Intelligence

The main piece of artificial intelligence that this game will have is the enemy that the player faces. The beetle will chase the player when it is facing towards the player. The speed of the beetle will increase by three times its original speed. There isn’t a lot of AI in this game that needs to be implemented, but implementing this simple piece of AI may take quite a few methods and variables to work out when the player is in front of the beetle. This could work by telling the beetle that the player is on the same platform as it is. A bounding box that extends outwards from the front of the beetle could be implemented.

# Physics

There will be a few physics in this game but mainly to do with gravity acting on the player. When the player jumps there will be a downwards force acting upon him. There will also be an upwards force acting against the player while he is on a platform to stop him from sinking through it. Gravity should only be activated when the player is in the air.

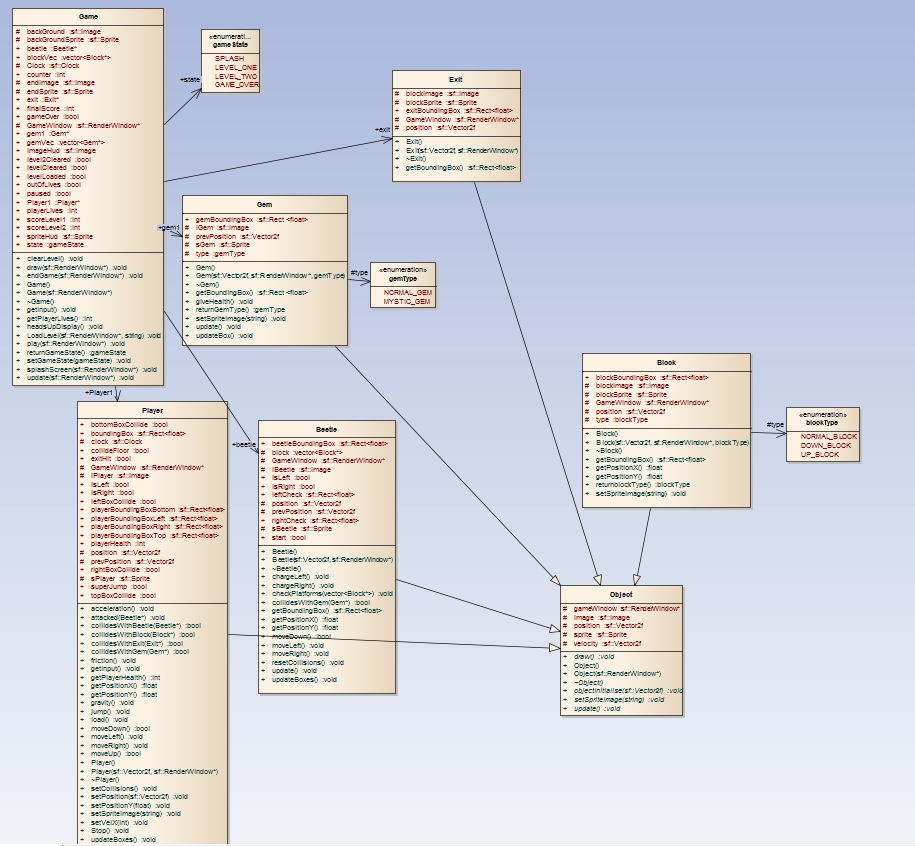
# Game Objects

There will be several game objects in this development. There will be an Object class which holds an object that won’t be instantiated. This will be in charge of making the objects initialise and draw them for less code to be used. The player object will handle the logic of the player such as health and movement. There will be an enemy object which will be called beetle. This will be pretty similar to the player object except its movement will be handled by its own logic, as well determine collisions with the player and the gem. There will be a collectable object that will create the base logic for the collectables in the game, being the gem and the magic jewel. The gem and magic jewel will inherit their properties from the collectable object but have separate logic to determine how they act when they collide with other objects. SFML features a vector class which can be used to hold the positions of objects in the game world.

# User Interface

The user interface will have several elements to it with different purposes. The high score interface will show the player the high scores and allow them to enter their name if their score has beaten any of the top players. As the game starts, a little tutorial explaining the game will appear explaining the key game mechanics and the buttons. When the user is playing the game, there will be an indicator of the player’s health in clear view, as well as an indicator whether or not the player has picked up the magic jewel. A sprite will appear to indicate this.

# UML DIAGRAM IS IN BEELTERUN FOLDER NAME D ‘BEETLERUN UML’



# Summary of Classes and Header Files.

### Game Class:

The Game class holds the main block of code that ties everything together. The game objects role is to keep running an update of the screen every processor tick. It has a draw function which calls all of the other draw functions from other objects. It receives input so that the player’s actions are reproduced on screen. The game class has several functions to keep the game updating and running, which end up going in the function named ‘play()’. This will be called by main and will keep running through this until the game is over or the user quits.

(Code Design)

void draw(sf::RenderWindow\* window); //This initialises and draws sprites

void getinput();//Handles input from the player

void update(sf::RenderWindow\* window);//This handles updating the game

void play(sf::RenderWindow\* window);//should be called in the game loop.

void LoadLevel(sf::RenderWindow\* window, string filePath);

The game object will also include a level loader which creates objects in various places by reading in a text document containing unique characters depending on what needs drawing.

The level loader is in the game class itself and is one of the first things to be called when the game is launched. The game is separated into different states in the game place, which will call different levels and modify behaviour based on what state the game is in.

The game class also holds all the objects for use in the game. Some things will be stored in vectors such as blocks because they will be multiple objects of the same type.

### Player Class:

The player class holds the player object which the player will control. This player class will have many functions and methods. It will also hold important variables such as the player’s health. The player class will have several movement functions that get called when the player uses the movement keys. The player class also sets up its own bounding box’s. The player object will have 5 bounding box’s. One for itself and 4 around the object to detect what direction the collision is coming from, and the player will react accordingly. The player class inherits from the object class to use its draw sprite and initialise.

(Code Design/Methods planned)

void load();

void Stop();

void setVelX(int velX);

void setPositionY(float posY);

void getInput();

void moveLeft();

void moveRight();

void friction();

void acceleration();

void gravity();

void updatePlayer();

### Block Class:

The block class inherits from the object class to use its draw sprite and initialise functions. The block class holds the bounding box’s for each block, so that the player can collide with them. Each block constructed should be able to easily change its position when it’s drawn due to the constructor enabling the position to be set when the block is created. The block class will also have an ENUM to set the different types of blocks up with ease.

(Code Design)

enum blockType {NORMAL\_BLOCK, DOWN\_BLOCK, UP\_BLOCK};

### Beetle Class:

The beetle class will also inherit from the object class to use its draw sprite and initialise functions.  
The beetle will handle a lot of its own logic and control itself during the course of the game. It will have similar methods and functions to the player such as movement but the game will control this itself. The beetle will also have its own behaviours like when it will chase the player if the player is in front of the beetle on the same platform.

(Code Design)

void moveLeft();

void moveRight();

bool moveDown();

void update();

void checkPlatforms(vector<Block\*>block);

### Gem Class:

The gem class will also inherit from the object class to use its draw sprite and initialise functions.  
The gem class is in charge of creating gem objects around the level to aid the player in progressing through the game. The gems will give the player more health when he collides with them, and therefore more time. There will also be an ENUM type for the gem type. Since the player needs to utilise the mystic gem in the game, it will be a variation of the normal gem, as they have similar attributes and methods. The gem will have its own bounding box which will be used by the player object and the beetle object. The player will need to be able to collide with the gem to progress.

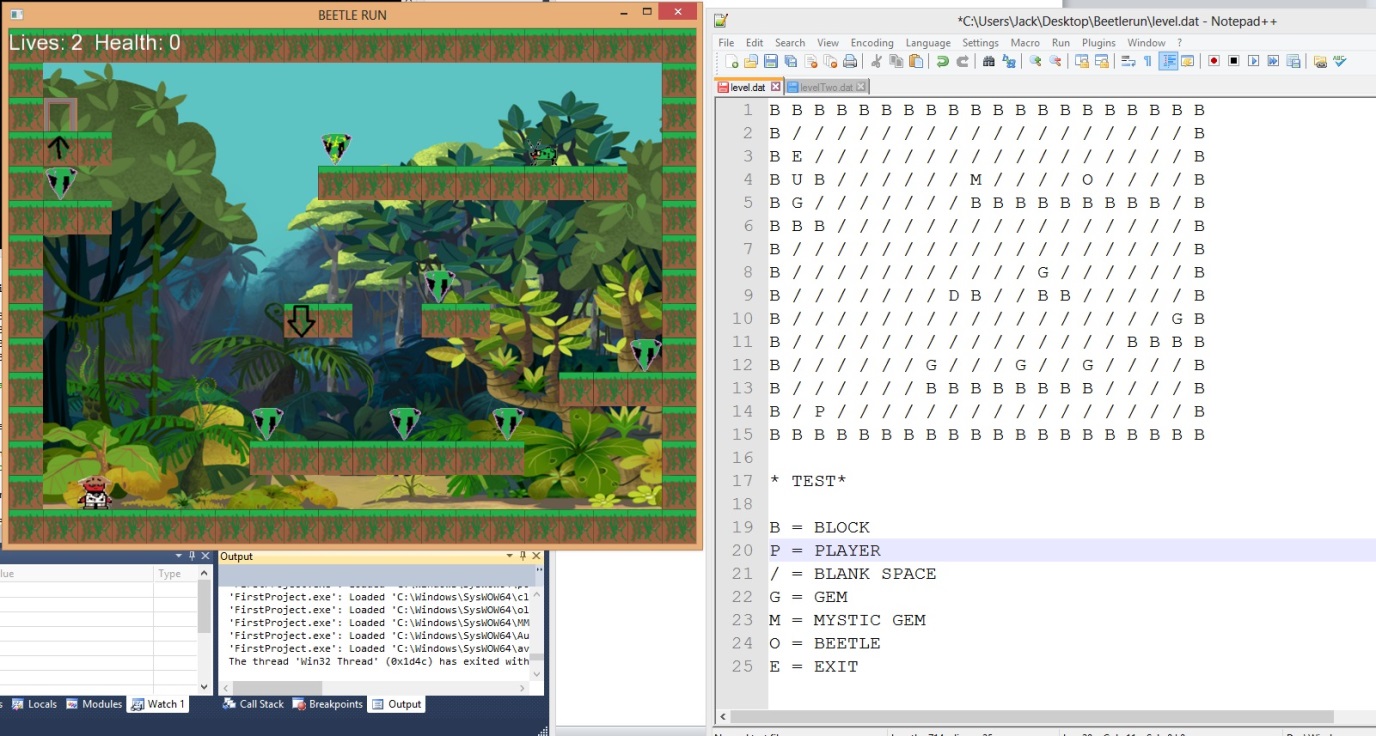
void giveHealth(); - An example of the type of method to give the player health.

void giveHealth()  
{  
 Player1->health += 10;  
}

# Test Log.

Most of the tests with these code aren’t very mathematical, and are mainly visual based testing, although some numbers had to be testing to make sure that things were adding up correctly.

1. Creating a level using a text file.



This test involved testing whether or not my level loader was working correctly.

*Expected output to match text file.   
  
Actual result was successful*

1. Player Collisions working correctly



***Collision tests were successful***

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected result** | **Actual Result** |
| Testing the players score at end of level. | Players score to be health at end of level | **Players score success** |
| Lives test | Life’s should count down when defeated | **Success** |
| Health Test | Health values should count down every second by two. | **Success** |
| Switching state | State should switch at end of level. | **Success** |
| Player Jumps | Player jumps upwards | **Success** |
| Gravity | The player falls when he jumps off a platform | **Success** |
| Beetle kills player | When player gets hit by beetle, life lost. | **Success** |
| Beetle hits gem | Beetle should turn around | **Sucess** |