5/6/2018

Connor Rowland – P16204361

LECTURER: Dr Simon Coupland

Blaster Design Report

Object Oriented Design and Development with C++

Contents

[Introduction: 1](#_Toc513470085)

[Controls: 1](#_Toc513470086)

[Design decisions: 1](#_Toc513470087)

[Test table: 3](#_Toc513470088)

# Introduction:

I created a 2 player, 2D worms-style, game. You fight another player, taking turns. You can either move, or attack. Your soldier has a health, lives, and a score. The one who has the most score at the end wins!

# Controls:

Left mouse-click to move.

Right mouse-click to throw a bomb.

# Design decisions:

I started by planning what the game would need. Once I had thought about what the game will have I started to extract data from each object, so I could create a hierarchical design. This would allow me to create polymorphic data structures.

I decided to extract the most basic components of an object, into a class called GameObject. This has a (game) position, a size, and a pure virtual method that child classes must implement, it inherits the draw method from sf::drawable. My plan was to have every game object, in the “world”, inherit from game in game.

Another (abstract/interface) class called DynamicObject inherits from GameObject. Every object that can move, will inherit from this class, as it has all the attributes necessary to move an object. This forces any child classes to create an update method, and a draw method, as every dynamic object should be able to move, and be drawn.

I didn’t see a reason to add an abstract base class, for static objects, as they’d mostly have the properties of the GameObject.

I extracted the ability for an object to do pixel perfect collisions by having a sperate class, called PixelPerfect that implemented the necessary attributes, and methods. This allowed any object to become a pixel perfect collision one by inheriting the necessary data.

The Soldier, and Bomb class inherit from the Dynamic Object and Pixel Perfect classes, so they have all the properties necessary to do pixel perfect collision, with the Terrain, and so they can move around in the world, while having gravity applied to them, to slow them, down.

Unlike the Soldier, and Bomb class, the Dynamic Pixel class only inherits from DynamicObject, as they’re a single pixel sized object (rectangle) anyways, which means we’re able to check the position of the dynamic pixel with the terrain pixel position, if its within bounds, to see if its on a solid pixel, or not.

To manage all the dynamic pixels a DynamicPixelManager class was made. It makes creating a single pixel or a cluster of pixels easy, a single method call, then it draws and updates them all. This removes a lot of clutter that would’ve been in the game class, its robust and makes adding additional features to the class a lot easier.

The Terrain class inherits from GameObject, and PixelPerfectObject, as it needs the data and behaviour those classes offer. This class manages the “static” pixels related with the terrain. It has two destroy methods (overloaded), that allows for the destruction of the terrain, and it calculates collision normals, when an object collides with it. One of the methods to destroy terrain takes a polymorphic data type (sf::Shape), so it can destroy any sf::Shape passed in.

I created a Randomiser class, which generates pseudo random integers, floats, or doubles, between two given values. I used the singleton pattern, so there’s only ever a single instance of it, in the program. I did this because I only want to seed it once, at the start, so it has better randomness, rather than recalculating the seed every time I want a random number.

Another class that uses the singleton pattern is the texture manager class. This is because I wanted a central object that manages the textures. Instead of having textures being allocated and loaded in, when needed, and every class storing a texture, the texture manager handles that. The texture manager loads the bulk of textures in at the start, and gives easy access to them, as well as the ability to easily update a texture with an image.

Collision detection is handled by a Collision class. It uses overloaded functors to return whether a collision has occurred, if any new object is added to the game that’s needs collisions with other objects then I just need to create the appropriate (extend) functors in the class.

To deal with collisions a Manifold is created, this is separated from the collision class, as, these are generated on the fly, not every collision generates a manifold, and for abstraction, as it allows extra functionality to be added, when needed.

A user interface class handles the players’ HUDs. It updates and draws them. It handles all the text the player sees!

I created an Audio manager class, to manage the background music, being played, which allows it to easily be switched, and manages a map of sound buffers, which can be used, by getting a sound object. This saves me having to bind the sound buffer to the sound object when I want to play the sound, as the Audio manager does this, when I get a sound(buffer) effect object.

Finally, we have the Game class, which brings all these together. It updates and draws the game components and processes key inputs. Instead of having a lot of data and behaviour in main, I wrapped it up into a nice Game class, so main only must call methods from the game instance, to create the game loop.

# Test table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test number: | Context:  (What’s being tested) | Input: | Expected: | Actual: | Passed: (Y/N) | Comment: |
| 01 | Started the game. | Game. | Game should start. | Game started. | Y | As expected, the game started. |
| 02 | Soldier lands on terrain. | Soldier. | Soldier lands on terrain. | Solder landed on terrain. | Y | As expected, the player lands on the terrain. |
| 03 | Moving the player, in bounds. | Left mouse- click. | Player should move. | Player moved. | Y | As expected, the player moved. |
| 04 | Moving the player, aiming off the left side of the screen. | Left mouse-click. | The player should bounce back on himself. | Player bounced back on himself. | Y | As expected, the player’s x velocity was reversed. |
| 05 | Moving the player, aiming off the Right side of the screen. | Left mouse-click. | The player should bounce back on himself. | Player bounced back on himself. | Y | As expected, the player’s x velocity was reversed. |
| 06 | Moving the player, aiming off the top of the screen. | Left mouse-click. | The player should bounce back on himself. | Player bounced back on himself. | Y | As expected, the player’s y velocity was reversed. |
| 07 | Soldier throws a bomb, in bounds. | Right mouse-click. | The bomb should be thrown. | The bomb was thrown. | Y | As expected, the bomb was thrown. |
| 08 | Soldier throws a bomb, out of bounds. | Right mouse-click. | The bomb should be thrown, and the memory cleaned up, when it goes out of the window. | The bomb was thrown, and the memory was cleaned up, when it went out of bounds. | Y | As expected, the bomb was thrown, and the memory was cleaned up, when the bomb went out of bounds. |
| 09 | Bomb collides with the static terrain. | Right mouse-click. | The bomb should remove a chunk of pixels away from the terrain. | The bomb removed a chunk of pixels from the terrain. | Y | As expected, the bomb removed pixels, at the surrounding area it landed at. |
| 10 | Dynamic pixels are created when a bomb collides with the static terrain. | Bomb, and static terrain. | The bomb’s explosion should create a bunch of dynamic pixels. | The bomb’s explosion created a bunch of dynamic pixels. | Y | As expected, the bomb’s explosion created a bunch of dynamic pixels. |
| 11 | Music plays when game starts | Game. | Music should start. | Music started playing. | Y | As expected, the music played. |
| 12 | Check if music loops. | Game. | Music should keep looping, until the game is over. | Music kept looping, until the game was over. | Y | As expected, the music kept looping. |
| 13 | Bomb hits the terrain the player who threw the bomb should get a score increase. | Bomb and terrain. Right mouse-click. | The player’s score should increase, when the bomb hits the terrain. | The player’s score increased, when the bomb hit the terrain. | Y | As expected, the player’s score increased, by 33.5 points, when the bomb hit the terrain. |
| 14 | Bomb hits an enemy soldier the player who threw the bomb should get a score increase. | Bomb and soldier. Right mouse-click. | The player’s score should increase, when the bomb hits the solider. | The player’s score increased, when the bomb hit the soldier. | Y | As expected, the players score increased, by 155 points, when the bomb hit the soldier. |
| 15 | Moving the player, aiming off the bottom of the screen. | Left mouse-click. | The soldier should die, and the enemy’s score should increase. | The soldier died, and the enemy’s score increased. | Y | As expected, the soldier died, and the enemy’s score increased by 335 points. |
| 16 | Bomb hits and kills an enemy soldier. | Right mouse-click. | The enemy soldier should die, and the player’s score should increase. | The soldier died, and the player’s score increased. | Y | As expected, the soldier died, and the player who killed the enemy gained 335.2 points. |
| 17 | Bomb hits an enemy soldier the soldier’s health should decrease by the amount of damage the bomb does. | Right mouse-click. | The enemy’s soldier’s health should decrease when the (enemy) bomb hits them. | The enemy’s soldier’s health decreased when the bomb hit them. | Y | As expected, the soldier’s health decreased by the damage of the bomb. |
| 18 | When the one of the player’s soldier dies the game should end. | Soldier. | The game should end and send the player to the game over screen. | The game ended | Y | As expected, the game ended, and the player[s] were sent to the game over screen. |
| 19 | Dynamic pixels collide with the terrain. | Dynamic pixel, and terrain. | The dynamic pixel[s] should collide with the terrain and be pushed back up. | The dynamic pixel[s] collided with the terrain and were pushed back. | Y | As expected, the dynamic pixels collided with the terrain and were pushed apart. |
| 20 | Exit game over screen. | Any key/mouse input. | The program/game should close. | The program/game closed. | Y | As expected, when the player pressed nay key the game closed. |