CMP208 – Games Programming and System Architectures

Coursework Report

Student Name: Bridget Annabel Casey

Student Number: 1802644

Git Username: BridgetACasey

Contents

[Introduction 2](#_Toc69991309)

[User Guide 2](#_Toc69991310)

[Project Setup & Navigation 2](#_Toc69991311)

[Gameplay 3](#_Toc69991312)

[Application Design 3](#_Toc69991313)

[Audio 4](#_Toc69991314)

[User Input 4](#_Toc69991315)

[Game Objects 4](#_Toc69991316)

[Game States 5](#_Toc69991317)

[User Interface 5](#_Toc69991318)

[Utility 5](#_Toc69991319)

[Techniques Used 6](#_Toc69991320)

[Data Oriented Design 6](#_Toc69991321)

[Reflection 6](#_Toc69991322)

[References 6](#_Toc69991323)

[Research 6](#_Toc69991324)

[Assets 7](#_Toc69991325)

# Introduction

The application featured in this coursework is a prototype of a platform-based computer game titled ‘Trek’. It was developed in C++ 17 and compiled in Microsoft’s Visual Studio 2019. The software does not feature animations, but it does make use of 3D geometry for character and object models. It also exclusively utilises the ‘Box2D’ physics engine for collision and movement events. The result is a perspective that is often referred to as 2.5D.

As a product, the game would be suitable for audiences of any age but would likely appeal most to fans of retro platforming games.

In Trek, the user takes control of a brightly coloured blob-like character, and they are tasked with navigating a treacherous mountain landscape by jumping across platforms made of snow and stone. They must also avoid pitfalls such as exposed molten lava, and blocks of ice that shatter upon contact. These hazards appear in increasing frequencies as they progress through the prototype level, to give the impression of rising difficulty. The player does not have health, and so upon falling through the world or into lava, they must restart from the beginning of the level. The player racks up a score by collecting golden coins along their journey, with the goal of eventually reaching a campfire at the end.

The current version of Trek has one level to showcase its core features and mechanics.

# User Guide

The submitted ZIP folder should contain everything required to build, run, and test the application fully. The prebuilt executable was compiled in Visual Studio’s Release mode, and thus will not have debugging information.

## Project Setup & Navigation

Included below is an overview of the folder structure and a brief guide on how to set up and run the application.

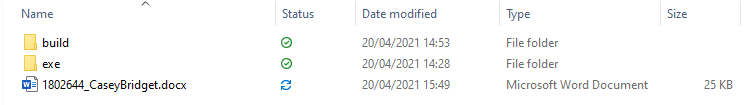


Figure : The top directory, containing folders for the source files and the executable application. This is also where the gef\_abertay and box2d repositories should be placed.

The ‘exe’ folder contains an executable file of the application (titled ‘assignment.exe’) for playing and testing purposes, along with the compiled libraries for GEF Abertay and Box2D. To minimise the size of the ZIP file submission, this folder will not contain all the resources required to run the executable first time.

The ‘build’ folder is divided into two more folders – assignment and media.

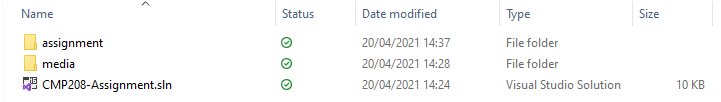


Figure : The assignment directory, containing all source files including assets and code.

The ‘assignment’ folder contains the Visual Studio 2019 project files for the assignment itself and for Box2D, which was created to allow easier viewing of the Box2D source code from within the Visual Studio IDE. This folder also contains all header and C++ files, inside a folder labelled ‘src’.

The ‘media’ folder is, as the name implies, the folder that contains all the assets required to successfully build the application, such as models and textures. In order to run the executable, the contents of this folder must be copied and pasted into the ‘exe’ folder.

It should be noted that in order to load the project into Visual Studio, the repositories for Box2D and GEF Abertay must be placed in the topmost directory – the same folder as this document is located, as this is where the Visual Studio projects will be searching for them by default. Neither Box2D’s nor GEF Abertay’s source code have been modified in the development of this application, so installing the latest versions of each should suffice. The Visual Studio Solution file is also provided within the ‘build’ folder to make the process of setting up easier.

## Gameplay

As for gameplay controls, they are relatively simple and can be compared to those of other platform games.



Figure : Diagram of gameplay controls.

The application is equipped to handle input from Sony controllers as well as keyboard and mouse. The user can navigate menus and click on buttons using either MOUSE and LEFT MOUSE BUTTON, LEFT ANALOG STICK, or by pressing UP and DOWN on the d-pad. The player is moved with LEFT ANALOG STICK or the A and D keys. Jumping is achieved with either the CROSS button or SPACE BAR, and sprinting is performed with the CIRCLE button or LEFT SHIFT. From within the level, the game can be paused with the OPTIONS button or the ESCAPE key.

Gameplay testing was completed using a standard keyboard and mouse setup, as well as a Sony PlayStation 4 DualShock controller.

# Application Design

Efforts were made to remain consistent with the design of the GEF Abertay framework in terms of class structure. This included adopting the use of ‘create’ functions to instantiate new objects and protecting the constructor, rather than leaving it public. Additionally, like the framework, each header file is encapsulated in a pre-processor definition check for compatibility with earlier C++ versions.

The use of enumerators over standard integers to retrieve elements of an array or map is also a consistent theme. This was implemented to allow for easier readability, as well as permit these lists of items to be expanded without much difficulty.

The combined use of two frameworks – GEF Abertay and Box2D – meant there was little need to create many more tools or manager classes before development on the game’s core mechanics could begin. However, the application has still benefited from the use of wrapper classes to encapsulate functionality from these two frameworks.

The application’s core functions can be divided into six main categories – audio, user input, game object generation, game state management, user interface, and general utility. The source files are also separated into their corresponding folders in the same manner.

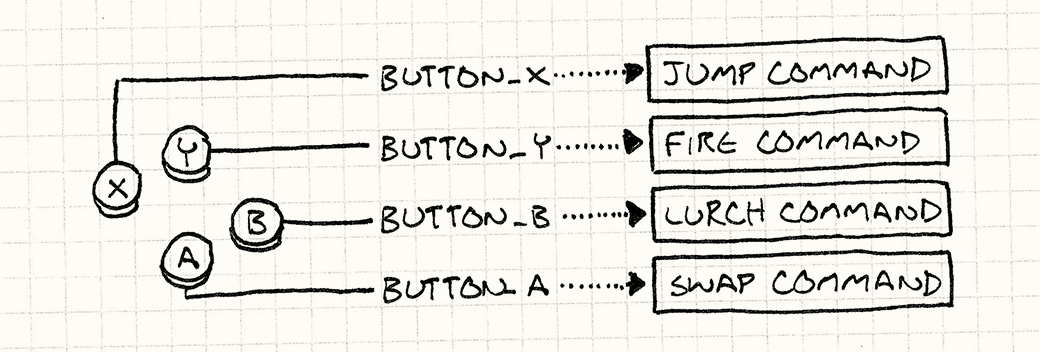
## Audio

The GameAudio class has few original features and serves primarily as a wrapper class to encompass functionality from the Audio3D, AudioEmitter, and GEF Abertay’s AudioManager classes. It stores scalar values for master, music, and sound effect volumes, as well as setter functions for each. Not all volume information, such as master volume, can be easily be retrieved from the AudioManager class. Thus, it was decided that storing these values in a wrapper class was the most sensible workaround. When called, these setter functions assign a given value, then pass them to AudioManager or loop over the available sound effects and adjust their volumes accordingly.

## User Input

Much like GameAudio, GameInput is a wrapper that draws from GEF Abertay’s InputManager and condenses its most used functions into a handful of methods specific to the game. However, it also introduces a system for mapping buttons to in-game actions.

The game input system is based on the Command pattern, with some minor tweaks.

Figure : An illustrated example of the Command pattern being used to map user input to an Xbox controller, from the book Game Programming Patterns by Robert Nystrom (2011).

GameInput stores a collection of Key objects, which therein store a pointer to an InputCommand object. As the application supports both Sony controller and standard keyboard input, associating these Key objects with one specific button as depicted in the diagram above did not seem appropriate. Instead, the Key objects are named after their given commands, and return the corresponding button or key pressed depending on which is the active input device.

To check which is the active input device, GameInput also contains Mouse and Controller structs that store information about the positions and states of keys, buttons, or analogue sticks – the latter struct of which also stores a Boolean variable called ‘active’. If Sony controller input is detected, ‘active’ is flagged as true and it is assumed the controller is the current input device. Then, if keyboard or mouse input is detected, the reverse happens.

Input is structured in this manner as the way in which some gameplay mechanics are executed will be different depending on which device is used, such as selecting a button in a menu.

## Game Objects

All game objects that appear in the level scene inherit from the base GameObject class. This class contains essential attributes shared by all game objects, such as world position, scale, and rotation. It also contains an instance of CollisionTag, an enumerator class which is used to determine what type of game object it is assigned to. This attribute is utilised in the ‘onCollisionBeginWith’ and ‘onCollisionEndWith’ virtual functions, which child classes use to determine what happens to themselves when they collide with another game object of the specified type.

The collision system makes heavy use of call back functions from the world in the Box2D engine. The CollisionListener class, which inherits from Box2D’s b2ContactListener, is responsible for determining what happens when two bodies come into contact. It does not determine how these bodies overlap, which is handled by other parts of Box2D, but calls the appropriate ‘onCollision’ function from GameObject after a valid collision has occurred.

So long as CollisionListener is set as the active world contact listener in the Box2D engine, all other collision-related code can be neatly encapsulated in a game object’s overridden ‘onCollision’ functions. This system negates the need to loop over all objects in the scene when checking for collisions and allows a game object to easily determine what happens to itself without ever having to know anything about the other object it is colliding with.

Below is a sample of pseudocode to help illustrate how this process works.

Pseudocode – Collision Listener

1. **IF** collision has begun
   1. **SET** bodyA to FixtureA -> get contact body
   2. **SET** body B to FixtureB -> get contact body
   3. **CAST** bodyA -> get user data to GameObject
   4. **CAST** bodyB -> get user data to GameObject
   5. **SET** objectA to bodyA -> get user data
   6. **SET** objectB to bodyB -> get user data
      1. **IF** objectA is **VALID**
         1. **IF** objectB is **VALID**
         2. **CALL FUNCTION** objectA –> onCollisionBeginWith
            1. **PASS** objectB -> get collision tag

**IF** collision tag = objectB -> get collision tag

**DO** some operation on objectA

**ENDIF**

* + - * 1. **CALL FUNCTION** objectB –> onCollisionBeginWith

**PASS** objectA -> get collision tag

**IF** collision tag = objectA -> get collision tag

**DO** some operation on objectB

**ENDIF**

* + - 1. **ENDIF**
    1. **ENDIF**

1. **ENDIF**
2. **IF** collision has ended
   1. **SET** bodyA to FixtureA -> get contact body
   2. **SET** body B to FixtureB -> get contact body
   3. **CAST** bodyA -> get user data to GameObject
   4. **CAST** bodyB -> get user data to GameObject
   5. **SET** objectA to bodyA -> get user data
   6. **SET** objectB to bodyB -> get user data
      1. **IF** objectA is **VALID**
         1. **IF** objectB is **VALID**
         2. **CALL FUNCTION** objectA –> onCollisionEndWith
            1. **PASS** objectB -> get collision tag

**IF** collision tag = objectB -> get collision tag

**DO** some operation on objectA

**ENDIF**

* + - * 1. **CALL FUNCTION** objectB –> onCollisionEndWith

**PASS** objectA -> get collision tag

**IF** collision tag = objectA -> get collision tag

**DO** some operation on objectB

**ENDIF**

* + - 1. **ENDIF**
    1. **ENDIF**
  1. **ENDIF**

1. **ENDIF**

## Game States

The state machine implementation is heavily inspired by the State software design pattern.

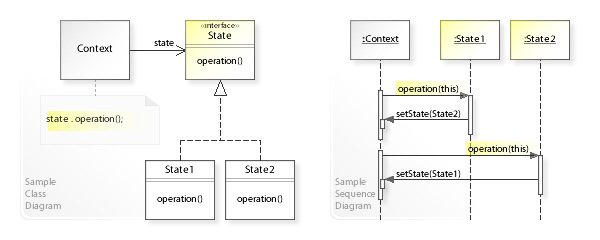


Figure : An example of a UML class diagram (left) and a sequence diagram (right) of the State pattern in action. Taken from w3sdesign.com.

All concrete states inherit from the State class, which is purely virtual and stores a pointer to the Context class. State also contains the ‘onEnter’ and ‘onExit’ functions, which are primarily populated with commands to instantiate or delete a state’s contained objects, serving a similar purpose to a constructor or destructor. This ensures that each state is responsible for managing its own attributes upon transition in or out. The State header file also contains includes for all the other essential classes, such as MeshManager, to avoid duplication of includes in child classes.

Context is similar in concept to a game state manager, and stores instances of objects needed by every concrete state, such as GameInput and GameAudio. Context is primarily responsible for instantiating and storing State objects, additionally storing a pointer to the active State object. State instances are stored in a standard map and ordered by a StateLabel – an enumerator class containing a collection of State names.

Context also handles transitions between states in a similar manner as depicted in the diagram above. During this transition process, the previous state’s ‘onExit’ function is called, the new state is set as the active one, a reference to Context is passed to it, and then its ‘onEnter’ function is called. This ‘setActiveState’ function can be accessed from within any State sub-class, meaning any state can switch the active instance to a different one without ever having to know anything about the new state.

## User Interface

The application’s user interface utilises two components – Button objects and Slider objects, based on the GEF Abertay Sprite class.

The Button class checks primarily for three things – if it is being hovered over, clicked, or held down. Each of these functions returns a Boolean value that changes depending on the button state. The ‘isHovering’ function determines if the mouse coordinates are overlapping with the Button object’s dimensions, and it also changes the Button’s texture depending on if this returns true. The ‘isClicked’ and ‘isHeld’ functions similarly check if the left mouse button has been pressed, with the exception that the latter is true by default if the Sony controller is the active input device. This is for ease of use later in the Slider class.

Slider inherits from Button. ‘Slider’ technically refers to the interactable button component of a slider, but the class stores an additional two sprite objects for the upper and lower portions of the background. In addition to being selected, a Slider object is permitted to move between two given coordinates along the x-axis. The lower background’s dimensions are set to the minimum and maximum x-axis coordinates, whereas the upper background’s size and position scale relative to the slider button’s position. This gives the illusion that the background is ‘filling in’ as the slider button shifts along the x-axis.

Button objects are used in every menu state, but Slider objects are exclusively used in SettingsState to control the application’s master volume, music volume, sound effect volume, player movement speed, and player jump force.

## Utility

The application’s utilities section is made up of classes that did not cleanly fit into any other category, but still provide necessary functions. Its most notable are the texture and mesh manager classes, where scene assets are instantiated and stored. This is done in such a way that all textures and meshes are loaded in once and stored in their respective manager classes – a design based around the Flyweight software design pattern.

A game object that needs to access a specific texture would store a pointer to a texture object, and upon that game object’s initialisation in a state, it will be told to point to the appropriate instance of Texture that has been retrieved from TextureManager.

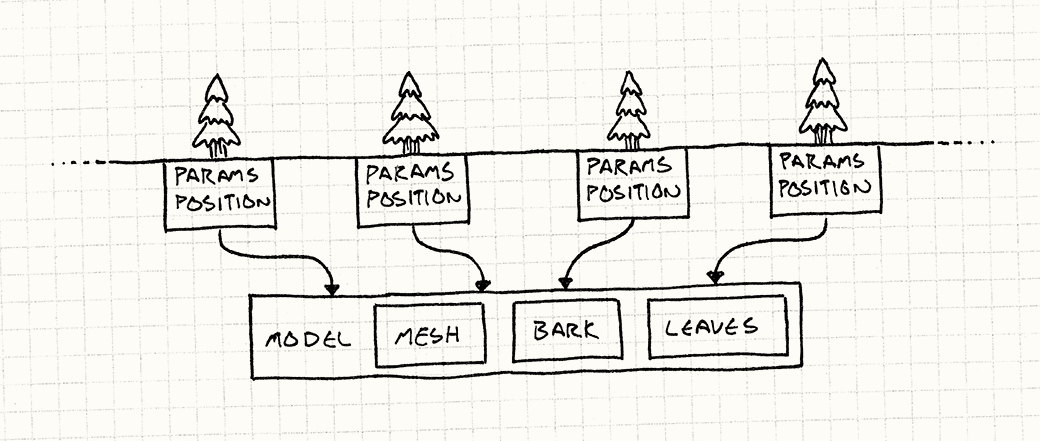


Figure : An illustration of the Flyweight pattern being used to separate a tree object from its mesh and other components. From the book Game Programming Patterns by Robert Nystrom (2011).

The MeshManager class is designed in an identical form to the TextureManager, the only difference being it is responsible for loading in SCN files instead of PNGs.

Lastly is the ParallaxBackground class, which is responsible for storing the layers of a parallax image and updating their positions on the screen. This class stores a series of Layer objects in a vector, each representing one layer of the parallax image. It also stores an identical copy of this vector.

The Layer struct contains a depth value to determine its order of appearance on the screen, a velocity value to determine its scrolling speed, and a Sprite object. The ‘updateBackgrounds’ function iterates over all Layers and shifts the Sprite objects along the x-axis, relative to their depth and velocity. When a layer or its corresponding copy reaches the edge of the screen, its position is reset to its original starting point, and resumes scrolling. This gives the illusion that the background is infinite without ever needing more than two instances of a particular image, thus saving space in memory.

# Techniques Used

Early in the design process, it was decided that researching standard software design patterns and common game programming patterns would be the most sensible approach to creating an efficient application. These concepts are long established, thoroughly tested, and well documented, and were thus likely better bases to work from than attempting to invent completely original systems.

As mentioned previously, the application primarily utilises three of these patterns – the Command pattern, the Flyweight pattern, and the State pattern.

The Command pattern was adapted for user input. The implementation of this pattern ensured commands and their corresponding keys were neatly separated, meaning they were easy to expand upon if more commands or keys wished to be added. Additionally, as input was not bound to a specific game object class, such as Player, a command could be assigned to any game object. This made the code much more flexible in the event an input command was executed on a different object, such as an enemy or second player character.

The Flyweight pattern was utilised in both the TextureManager and MeshManager classes. A straightforward way to handle texture or mesh management, and the initial method used earlier in development, was to simply store an instance of a texture or mesh within the Game Object class. However, this led to unnecessary duplication of data if there were multiple instances of the same game object.

A better system was then implemented, which would ensure there was only ever one instance of the same type of data at runtime. Therefore, potentially hundreds of objects could all be utilising the same instance of a particular game asset, thus minimising memory usage.

The State pattern was used as the basis for the application’s state machine. The design of this pattern, having a Context class and several concrete State classes that are in constant communication, is easy to work with and build upon.

There is only ever one instance of Context or each concrete State it manages. Having only one instance of Context in the application at runtime also ensures two things - that the active state is always interacting with the same instance of Context, and that unnecessary duplication of objects is avoided. For example, rather than storing an instance of TextureManager in each state that requires one, a state can access the same singular instance of TextureManager via its reference to the Context class.

Additionally, a core theme during development was ensuring as many distinct features of the program were decoupled as possible. No specific decoupling patterns were used, but an effort was made to implement this concept, nonetheless. This can be demonstrated in both the state machine and the collision listener. Both implementations ensure that states or game objects do not need access to one another for communication to happen between them, which is done through Context or CollisionListener respectively.

Decoupling could be further expanded by implementation of a Component system, such as splitting the physics-related functions of GameObject into its own class and creating a BodyComponent object, for example. Implementation of this system was considered during development; however, it was decided to be an unnecessary amount of abstraction for a relatively small application.

# Data Oriented Design

Despite the attempts to create a program that was as memory efficient as possible, there are still several areas that could be improved upon.

Throughout the program are several uses of arrays, vectors, or maps of custom objects. Most of these store pointers to objects, rather than instances of objects. These occurrences can be found in TextureManager, MeshManager, SettingsState, and more. Additionally, such as with the GameObject class, many of these objects have attributes which frequently need to be accessed by other classes. When these arrays are iterated over, it can result in hopping through memory several times in search of the address of each element, and therefore multiple CPU cache misses.

This slows the program down unnecessarily and could have been avoided by using contiguous arrays to store objects, instead. Additionally, declaring these arrays on the heap and storing objects within them instead of storing pointers to objects would have generated more cache hits, as the data is more tightly packed in memory.

Within the LevelState class is a standard vector of type GameObject pointer, called ‘map’, which stores a collection of all game objects in the level. Although this makes the process of generating the map and iterating over its elements much easier, this kind of heavy reliance on polymorphism can hinder performance. No instance of the GameObject base class is ever stored here, meaning sub-classes of GameObject that likely have varying sizes are all being jammed into the one container, resulting in padding around the smaller elements. This runs the risk of creating cache misses, as these objects may not be very close together in memory. A way around this would be avoiding the use of polymorphism altogether and storing these objects contiguously, in the manner mentioned in the paragraph above. This method would have ensured better memory locality and increased the likelihood of cache hits.

Despite being easy to use and read, the state machine implementation also has flaws in memory management. State is a purely virtual class, meaning all key functions of its sub-classes are overriding another function. These include the ‘update’ and ‘render’ functions, which are called every frame by Context. This can create even more cache misses as there is no way to predict which memory address the instruction lies in, and this kind of indirection can result in jumping to an address outside of the current cache. A resolution to this would be making these key functions non-virtual, and simply having separate definitions for them in each sub-class of State.

Perhaps the easiest optimisation would be in how getter and setter functions are written in each class. Presently, these functions are written like any other and declared in the header file before being defined in the source file. However, this can result in significant overhead time, as such small and simple functions will take a relatively short time to execute compared to the time to push onto and pop off the stack. A remedy for this problem would be marking each of these functions as ‘inline’ and defining them all in their respective header files, thereby avoiding overhead.

Finally, the application could have benefited from the use of multithreading on the CPU. Presently, all processes are run on the main thread, but some of these processes could be run concurrently without issue. The most sensible portion of code to implement this on would have been game audio. During testing, it was noted that audio samples sometimes stuttered or were cut short if played in a state with many objects. If all audio were played on a separate thread, it is less likely this would occur due to it being the only process that thread was responsible for.

# Reflection

The level of research and planning put in prior to formal development proved to be essential later, especially in reference to software design patterns. Without this, it is likely time would have been wasted attempting to arrive at the same conclusion or even implement solutions that were less effective. This permitted greater focus on developing gameplay mechanics and polishing code.

However, some initial features were still cut due to time constraints. The controls menu was originally meant to feature a key bindings section, which would allow the user to dynamically change key bindings at runtime. The current input system would have served as a basis to allow this; however, the feature was still scrapped, in part due to insufficient knowledge and commitments to other projects. More focused research into this area would have likely prevented this from happening, as well as better balancing of this project with others.

Getting to grips with frameworks – both Box2D and GEF Abertay – was also an unforeseen challenge that took longer than anticipated. While not devastating to development, it did force some reconsideration of program structure as new information was learned, such as how volume is handled in the GameAudio wrapper class.

Finally, more consideration of memory and cache efficiency when designing the program would have proved rewarding. Better data locality would have led to a more stable application overall, and this can be put down to an inferior understanding of memory management at the start of development compared to the end of development.

Overall, the final product is an accurate reflection of the initial design and the development process was a success, despite some challenges along the way.

# References

Below is a collection of all resources used in the development of the application and the writing of this report.

## Research

Nystrom, B. (2011) *Command*. Available at: https://gameprogrammingpatterns.com/command.html (Accessed 5th April 2021).

Nystrom, B. (2011) *Data Locality*. Available at: http://gameprogrammingpatterns.com/data-locality.html (Accessed 20th April 2021).

Nystrom, B. (2011) *Flyweight*. Available at: https://gameprogrammingpatterns.com/flyweight.html (Accessed 5th April 2021).

W3S Design (2017) *The State Design Pattern – Structure and Collaboration*. Available at: http://w3sdesign.com/?gr=b08&ugr=struct (Accessed 20th April 2021).

## Assets

Abertay University (2021) *Box collected* [Sound effect file]. (Accessed 7th April 2021).

Arks (2020) *PS4 Buttons*. Available at: https://arks.itch.io/ps4-buttons (Accessed 7th April 2021).

BlobFortress (2017) *Lowpoly gold coin*. Available at: https://sketchfab.com/3d-models/lowpoly-gold-coin-34794c00e9d140f6b86e930fef18b009 (Accessed 7th April 2021).

DL Sounds (no date) *Blazer Rail 2*. Available at: https://www.dl-sounds.com/royalty-free/blazer-rail-2/ (Accessed 7th April 2021).

DL Sounds (no date) *Blazer Rail*. Available at: https://www.dl-sounds.com/royalty-free/blazer-rail/ (Accessed 7th April 2021).

Karat, A. (2019) *Low Poly 3D Block - Devils Work.shop*. Available at: https://devilsworkshop.itch.io/essential-low-poly-isometric-3d-block-and-hex-pack (Accessed 7th April 2021).

Koenig, M. (no date) *Computer Error Alert*. Available at: http://soundbible.com/1540-Computer-Error-Alert.html (Accessed 7th April 2021).

Lil Cthulhu (2021) *Pixel Art Cave Background*. Available at: https://lil-cthulhu.itch.io/pixel-art-cave-background (Accessed 7th April 2021).

Mixkit (no date) *Player recharging in video game*. Available at: https://mixkit.co/free-sound-effects/player/ (Accessed 7th April 2021).

Mixkit (no date*) Video game retro click*. Available at: https://mixkit.co/free-sound-effects/click/ (Accessed 7th April 2021).

Mojang (2016*) PC Computer - Minecraft Java Edition - Sound Effects*. Available at: https://www.sounds-resource.com/pc\_computer/minecraft/sound/6210/ (Accessed 18th April 2021).

Nectanebo (2020) *Menu Buttons*. Available at: https://nectanebo.itch.io/menu-buttons (Accessed 7th April 2021).

Nitti, V. (2018) *Glacial Mountains: Parallax Background*. Available at: https://vnitti.itch.io/glacial-mountains-parallax-background (Accessed 7th April 2021).

Yeahyea, A. (2020) *Low Poly Campfire*. Available at: https://sketchfab.com/3d-models/low-poly-campfire-07e1f44b9b7c4492a0f3abb08b86e243 (Accessed 9th April 2021).