Technical Design Document

Product name: Jumperman
Team name: Only4

Author:
Bryan Koh Yan Wei

Reviewers:
Seet Min Yi (Testing Lead)
Wee Boon (Design Lead)
Dallas Lau Cheong Kin (Producer)

Last Revision Date:
14/2/2021

Table of Contents

1.Problem statements

1.1 Dilemma for physics

When developing this project, our team will encounter a multitude of real world physics problems such as gravity and acceleration. But since the team is inexperienced at game developing, the team feels that accurate simulation is something that is out of grasp. With that said, the plan for physics in Jumperman will be to try as close as possible to imitate the physics of the real world.

1.2 Audio integration

Alpha Engine does not offer support for audio integration. As a result, our team will be required to source for external libraries to fill the shoes. Currently, our team is looking to use FMod to fill this gap..

1.3 Collapsing is a core feature of Jumperman

One of the core features of Jumperman is the interaction between the player, enemies and the tiles. As a refresher; when the player kills an enemy, the tile that the enemy was located on will start to collapse after a delay. When a tile starts to collapse, it triggers any adjacent tiles to start collapsing too. To achieve this, the engine will need to check if the adjacent tiles should be collapsed when the collapsing event triggers.

1.4 Need for a dynamic collision system

Collision is a core feature of platformer genres and Jumperman is no exception. The core features in Jumperman will include player movements, events with enemies and collapsing tiles, all which relies on features and triggers from collision data. To accurately simulate these features. Hence, to achieve this purpose the team will need to implement a collision system that is capable of separating between the different possible outcomes when two objects collide.

2. Context

2.1 Glossary:

Physics: A branch of science that deals with matter and energy and their interactions.

FMod: A proprietary sound effects engine and authoring tool for video games and applications developed by Firelight Technologies.

AlphaEngine: A C++ external library provided by DigiPen Institute of Technology which consists of certain basic functionalities such as Graphics rendering and Math amongst others provided to the students to aid game development.

Level Editor: A game development tool used to design levels, maps, virtual worlds for a video game.

Object-oriented programming(OOP):OOP is a programming paradigm based on the concept of "objects", which can contain data and code. A key feature of OOP is the ability for an object's own procedures to directly access and modify its own data members.

Pseudocode: An intermediary language between human readable language and machine code which explains an algorithm or another system. Often used as a medium for human beings to explain to one another about the workings of a system while encapsulating the programming aspect.

Functions (programming): A reusable set of instructions written in code to perform a specific task.

Class(programming): A user defined data type.

Application Programming Interface(API): A computing interface that defines interactions between different software mediums.. E.g. AlphaEngine for Graphics, Fmod for Audio.

Collisions: An event when the shapes of two rigid bodies are intersecting at a common position.

Axis Aligned Bounding Box(AABB): A collision detection methodology based on the principle of rectangular collision shape aligned to the base axes of the environment. E.g: in 2D, alignments to the x and y axis.

2.2 Proposed solution

Our proposed solution to the problems we mentioned earlier is to try and use everything at our disposal, (e.g: AlphaEngine Math collision detection) and try to solve problems with a trivial approach. Rather than potentially failing at implementing complicated solutions and coming short.

Our team understands that our approach may not offer the most technically complicated solutions, the team looks to develop a game that is simple and fun rather than a simulation that is technically complicated. Given the lack of experience in both coding and game development, this is the approach the team agrees is the best fit for our position.

The team’s vision for this project is to make a game that is wholesome and enjoyable.

3. Goals

3.1 Physics Implementation goals

As mentioned earlier, our team feels that accurate physics is not something that is within the team's grasp. For Jumperman, realistic and accurate physics will be a non-goal. But since some degree of physics will still be required in order to develop jumperman, the goal for physics in Jumperman is to have a system that imitates real world physics. As an example, there will be "gravity" introduced in jumperman. The application of the "gravity" in Jumperman however, will not accurately simulate real world physics. As for accurate physics simulation, our team feels that is a goal that is best reserved for the future.

3.2 Audio Integration goals

Our team believes that audio assets are quintessential to any game. Hence, this will be a goal that the team is looking to score. Specifically, there are two major goals for audio integration. The first type being integrating a function which can play an audio asset once for uses such as sound effects upon killing enemies. And the second being a function which will play an audio asset on a loop for a scene for purposes such as background music. Non goals for auditory integration will be high levels of integration such as sound grouping systems.

3.3 Collapsing system goals

The goal for collapsing systems is to have a manager which, while working together with the collision manager, provides accurate data to the system on which tile to collapse. Since Jumperman will feature several different types of tiles, the manager should be dynamic enough to discern if a tile type is collapsible. And should it be collapsible, apply the collapsing event on the tile whenever appropriate.

3.4 Collision system goals

In Jumperman, there are many different possibilities for collisions. From the player colliding with platforms which prevents gravity from applying, to the player dying to enemies on collision. There are many different scenarios that will require an accurate collision check to determine which scenario to play out. The goal for Jumperman's collision system is to provide accurate collision checks to correctly trigger events. For this project, our team intends to use basic shapes such as circles and rectangles as much as possible as the collisions between these shapes are much easier to handle.

A future goal for the collision system is to have support for more robust shapes such as a polygon which will require the use of the Separating Axis Theorem (SAT).

4. Solutions

4.1 Solution for Physics Implementation

4.11 Context:

As mentioned earlier, accurate physics will be out of Jumperman's current scope. But some degree of physics is still required for the production of Jumperman. The team's solution for this problem is to apply a constant velocity instead of a realistic dynamic velocity which is applied relative to the projectile maximum height, gravitational strength and the force of the projectile.

4.12 Proposed solution:

Bringing back the example of gravity from earlier, instead of obeying the law of acceleration, the team will instead apply constant acceleration of a certain value and apply it towards the player whenever the player's feet are not in contact with any surface.

Pseudo Code:

Example 1: Acceleration due to gravity

```
const float gravity_strength = 10.0f

Game Update Loop{
        IF Player is alive{
                IF Player feet is not in contact with any surface{
                        Apply gravity_strength to player's y coordinates
                } // end if
        } // end if
} // end while
```

Hence forth, anything that is physics related will be integrated into Jumperman with a similar perspective in mind.

4.1.3 Physics solution dependencies:

- **NIL**

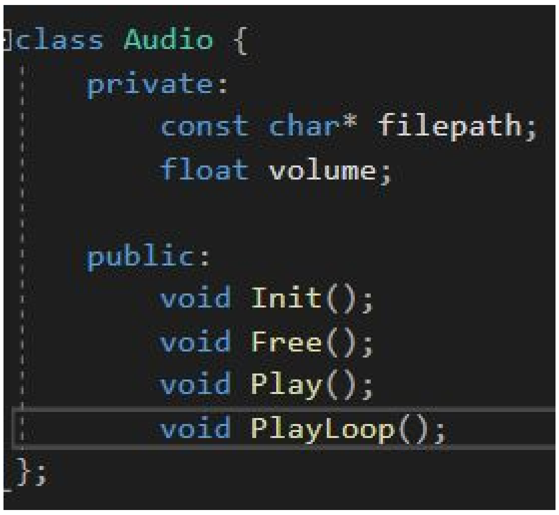
4.2 Solution for Audio Integration

4.2.1 Context: Why FMod?

Two major reasons. Firstly, there is a well documented API guide to use FMod in a C++ environment on the FMod webpage. And secondly, it is a API which the Teaching Assistants(TAs) are familiar with. Hence, should we require their support, it is likely that they will be able to offer us a helping hand.

4.2.2 Proposed solution

The team's plan is to author functions in a C++ environment using FMod's library which will handle the initialization, loading, playing, pausing, looping and freeing of audio assets. The motivation behind this is to have a modular system which can integrate many different audio assets during the development of the Jumperman.



```
class Audio {
    private:
        const char* filepath;
        float volume;

    public:
        void Init();
        void Free();
        void Play();
        void PlayLoop();
};
```

Example of an Audio Class definition

Init will perform the necessary loading of the audio assets.
Free will perform any cleanup that may be required.
The play function will play the asset once until the end of its lifespan.
PlayLoop will play an asset indefinitely.

4.2.3 Audio Integration solution dependencies

- **FMod**

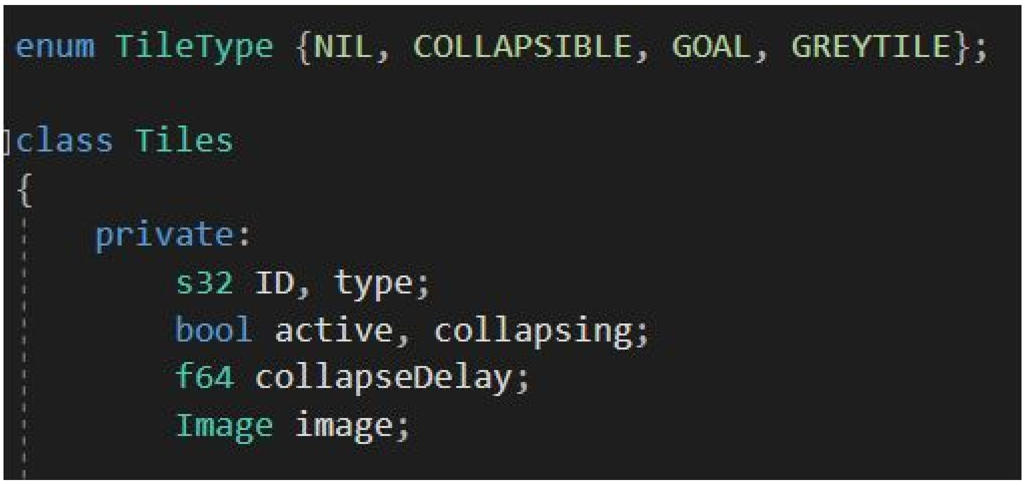
4.3 Solution for collapsing

4.3.1 Context:

In Jumperman, there are various different tiles that the team intends to integrate into the game. Each of the tiles have different properties, and not all the tiles are designed to be collapsible. Hence, we need a system in place that is flexible enough to identify the type of the tile it is checking, and whether it should be part of a collapsing chain effect.

4.3.2 Proposed solution:

The team intends to create a tile class which will encapsulate all the data needed to perform said operations.



```
enum TileType {NIL, COLLAPSIBLE, GOAL, GREYTILE};

class Tiles
{
    private:
        s32 ID, type;
        bool active, collapsing;
        f64 collapseDelay;
        Image image;
```

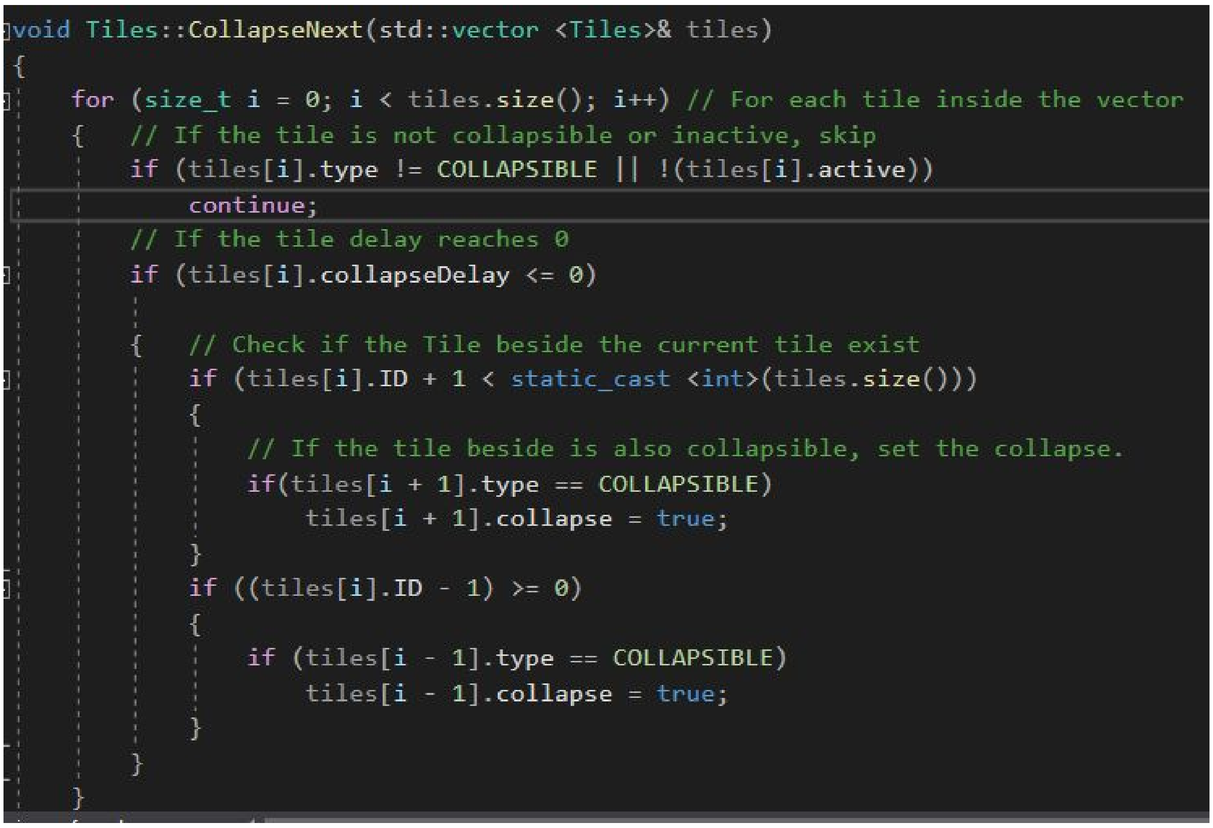
Example of Tile class definition

Purpose of data:

1. Type will represent which enum type the tile belongs to.
2. ID will represent the index of the tile inside a vector.
3. Active will represent if the tile is “dead” “alive”.
4. Collapsing is a boolean which indicates if a tile is currently collapsing.
5. collapseDelay is a float which determines the delay before the tile collapses.
6. Image is a class which contains the mesh, texture and the position of the tile.

The motivation behind defining a class for the Tiles is so that the paradigm of object oriented programming can be applied to each tile in the game.

Each Tile on the level will be added into a vector container and from there, a function will be called using that vector and check if the collapsing effect should be applied to the tile adjacent to the ground zero tile.



```cpp
void Tiles::CollapseNext(std::vector <Tiles>& tiles)
{
    for (size_t i = 0; i < tiles.size(); i++) // For each tile inside the vector
    {   // If the tile is not collapsible or inactive, skip
        if (tiles[i].type != COLLAPSIBLE || !(tiles[i].active))
            continue;
        // If the tile delay reaches 0
        if (tiles[i].collapseDelay <= 0)

        {   // Check if the Tile beside the current tile exist
            if (tiles[i].ID + 1 < static_cast <int>(tiles.size()))
            {
                // If the tile beside is also collapsible, set the collapse.
                if(tiles[i + 1].type == COLLAPSIBLE)
                    tiles[i + 1].collapse = true;
            }
            if ((tiles[i].ID - 1) >= 0)
            {
                if (tiles[i - 1].type == COLLAPSIBLE)
                    tiles[i - 1].collapse = true;
            }
        }
    }
```

<u>Example of a collapsing system</u>

<u>4.3.3 Collapsing solution dependencies</u>

- **AlphaEngine**

4.4 Solution for Collision

4.4.1 Context

In Jumperman, the game objects rely on collisions to trigger certain interactions with platform tiles and enemies during gameplay. Collision checks detects if the player is standing on the tiles and triggers specific functions depending on the types of tiles the player is standing on. The game also uses collision checks to detect where the player makes contact with the enemies to determine if the player defeats the enemies or dies to the enemies. In this context, collision is necessary to implement the core mechanics of Jumperman.

Thankfully, Alpha Engine has built-in math libraries which provide basic collision checks such as AABB collisions and circle-to-circle collision checks. Jumperman will rely on these library functions to perform collision checks for the various requirements its features need.

However, the problem of correctly differentiating the appropriate events based on the collision still stands. Simply deploying an AABB collision between the player and the enemy wouldn't suffice.

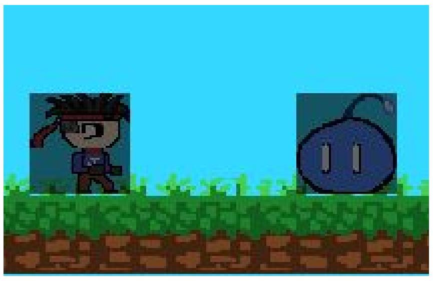
4.4.2 Proposed Solution

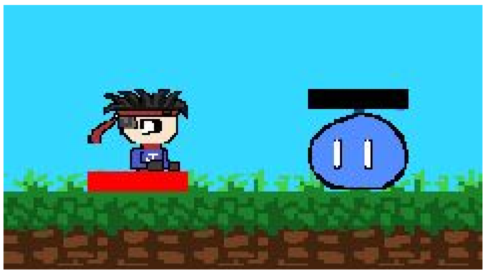
Our team's proposed solution to this problem is to have two sets of collision data specifically for the player and the enemy. The second collision data for the player would be a bounding box around the player's feet. Similarly, for the enemy, we will deploy a second bounding box around the tip of the enemy sprite.

When the system detects a collision between a player and an enemy, the system would first run a check on the second collision data between the player and the enemy. If the result of the between the second collision is true the system will set the enemies' flag to false and exit out of the function.

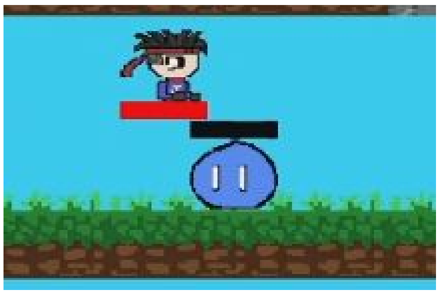
If however, the system does not detect a collision between the second collision data, the system will conclude that the player is the one who died to the enemy, and thus lose a live count.

Player colliding with enemy example:

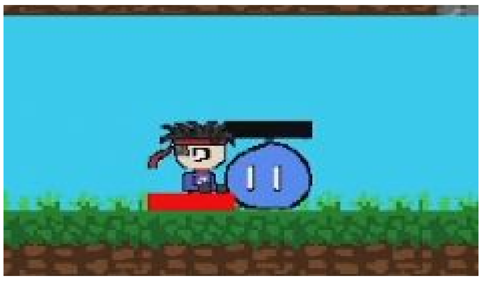

Collision data 1


Collision data 2

Case 1: Collision data 1 and 2 are ***both*** true:


Example of enemy dying to the player

Case 2: ***Only*** Collision data 1 is true:


Example of player dying to the enemy

4.4.3 Collision solution dependencies:

- Alpha Engine

5. Dev Plan

5.1 Task List

#	Task	Dependent on	Estimated Time taken
1	Movement: Player + enemy	Alpha Engine	4
2	Create platform tiles	Alpha Engine	6
3	Collision detection: player + enemy	Alpha Engine, Player and Enemy AABB data	4
4	Collision detection: player + tile	Alpha Engine, Player and Tile AABB data	4
5	Animation: player + enemy + collapsible tile	Pixil	4
6	Gamestate Manager	State engine	8
7	Binary Map	Saving and loading levels from text file	8
8	Binary Collision	Saving and loading levels from text file	8
9	Level Editor: Level 1	Saving and loading levels from text file	8
10	UI: Menu	Photoshop and Alpha Engine	2
11	UI: Gameover	Photoshop and Alpha Engine	2
12	Death Animation: player + enemy	Pixil	2
11	Sound system: death + jump + music background	Fmod	12
12	Level Editor: Level 2	Saving and loading levels from text file	2
13	Level Editor: Level 3	Saving and loading levels from text file	4
14	Level Editor: Level 4	Saving and loading levels from text file	4

Table 1. Task List

Systems like Level Editor, Binary Map Collision and Fmod would require more time to be implemented since we have never touched on these features before. Therefore, the estimated time for these tasks are longer as more time would be spent on learning and researching these materials.

5.2 Roll-out plan

Phase #	Week	Goals/Milestones	Comments
1	1 - 2	Sprite design Basic movement Rendering	Pixel art design for tiles, player and enemies Player movement, enemy movement, rendering of assets Rendering of assets using Alpha Engine
2	3 - 4	Basic collision system Collapsing system Pitch demo	Collision between enemy, tiles and player. Tiles collapse upon enemy death. Demo showcasing core features.
3	5 - 6	Documents Game State Manager	SRS, TDD, PPD
4	7	Audio integration Level editor Animation	Author functions using FMod external library Building of level through text file reading Player and enemy sprite animation
5	8	Level design Engine proof	Building of levels 1 to 3 Proof of feasibility.
6	9	Sound composing Sound integration Menu UI design	Background music Sound effects (Player death, enemy death, jumping) Buttons, background
7	10	Gameplay UI Design Alpha presentation User test demo	Player lives, current stage, time elapsed. Slides, core feature showcase Creation of a few levels for user tests on the following week.
8	11	Tutorial level design User testing High score system	Interactable tutorial level Execute user test Reading and writing to text file to save scores
9	12	Pause menu design Testing report submission	Review result of user test
10	13	Polishing, Submission	Hard deadline

Table 2. Roll out plan

5.3 Risks

5.3.1 Physics risk:
Other than the lack of a “wow” factor, there is no foreseeable risk. From a technical perspective, the purpose of going with this approach is to enable the team to focus more on developing a game that is more enjoyable, rather than possibly sacrificing gameplay at the cost of accurate simulations.

5.3.2 Audio Integration Risk:
As of writing this documentation, no one on the team has started looking at the FMod API. Hence there might be many restrictions and challenges that are still waiting to be discovered. This portion of the document will be updated at a later time when the team gets a clearer picture of what FMod is and isn’t capable of.

5.3.3 Collapsing system risk:
The current design of the collapsing system relies on the assumption that each tile is stored inside the vector inside a sequential order. Should that not be the case, the system might fail to work to expectations.

Making a more dynamic system is ofcourse, the more ideal solution. But because the team is mostly made of people who are relatively new to coding and game development, we decided to settle for a more trivial approach. Hence, this could potentially pose a risk of a system which is not as modular.

5.3.4 Collision risk:
In the event that the function of collision is not working properly, systems such as detecting a player’s or enemy’s death might be affected as well.

1. AABB collision checks are constrained to bounding boxes, so it poses difficulty to objects that use complicated meshes and sprites.
2. For physics and collisions to be working properly in game, the bounding box data for every instantiated object have to be accurate. Inaccuracies in bounding box data across multiple game objects will result in janky gameplay experience.

6. Success measurement

6.1 Success for Physics Implementation

Since Jumperman does not intend on simulating real world physics. The success measurement for physics in Jumperman is to have a physics system which can imitate laws of physics such as gravity throughout the game.

6.2 Success for Audio Integration

A success for audio integration will be the ability to use FMod to create functions in a C++ environment which can play audio assets both for a single time to use for sound effects, and on a loop for background music to use throughout the game. Anything more such as having different audio playing concurrently and creation of sound grouping will be considered a bonus. However, given that the API of FMod is still unknown to Only4 at the time of authoring this documentation, this goal might have to be readjusted depending on the learning curve experienced by the team. This portion will have to be updated at a later time to provide a more accurate success measurement.

6.3 Success for Collapsing system

Being one of the unique selling points of Jumperman, the collapsing effect is another feature which will strongly dictate the overall enjoyment of the product. As a means to that end, the collapsing system will have to work in tandem with the collision manager. When the collision manager detects that the enemy on a tile is killed, it will trigger a countdown timer which, when depleted, will cause the tile to collapse. Only4 intends to alert the player of the countdown starting by implementing a shaking effect on the tile. In order to make sure that tiles which are not supposed to collapse will remain unaffected, the system will have to accurately check if the tile adjacent is of the “collapsible” type. If it is not, the tile will remain unaffected. A success for this feature will be to have a system that manages all the tiles on the level, and only apply the collapsing effect, whenever a tile is supposed to collapse.

6.4 Success for Collisions

Since collisions between entities are what forms the core mechanics in Jumperman, the enjoyment of the product is heavily dependent on the collision. A perfect example can be seen from the interaction between the player and an enemy. When that happens, there are two possible outcomes, the first outcome would be the player dying to the enemy. The next being the enemy dying to the player. A success for the collision system will be to have a collision system that is able to accurately differentiate which of the two previously mentioned outcomes is the correct event happening based on the given collision data and with that information, executing the correct result.

7. Closing

7.1 References:

Physics library | science. (n.d.). Retrieved February 13, 2021, from https://www.khanacademy.org/science/physics

Computer programming - functions. (n.d.). Retrieved February 13, 2021, from https://www.tutorialspoint.com/computer_programming/computer_programming_functions.htm

What is Pseudocode? Definition of Pseudocode, Pseudocode Meaning. (n.d.). Retrieved February 13, 2021, from https://economictimes.indiatimes.com/definition/pseudocode

Flame, S. (n.d.). Level editor. Retrieved February 13, 2021, from https://digitalhorizons.org.uk/job-role/level-editor/#:~:text=Level%20Editor,the%20buildings%2C%20objects%20and%20landscape.

Api. (2021, February 07). Retrieved February 13, 2021, from https://en.wikipedia.org/wiki/API

Fmod. (2020, December 26). Retrieved February 13, 2021, from https://en.wikipedia.org/wiki/FMOD

Collision detection. (n.d.). Retrieved February 13, 2021, from https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection

Object-oriented programming. (2021, February 03). Retrieved February 13, 2021, from https://en.wikipedia.org/wiki/Object-oriented_programming