Technical Design Document

for

Hookshot

Version 1.0

Prepared by: Tan Wei Wen / Tan Egi / Lau Yong Hui / Liu Ke

Team: Hook

02th February 2021

Table of Contents

Revision History

Name	Date	Reason For Changes	Version

1.1 Introduction

1.2 Overview

‘Hookshot’ is a pixel art game that includes adventure, cave exploration, and a 2D side scroller. The game will revolve around the player, Mr. Hook, and his dinosaur buddy, Rex. The main objective of the game is to get the sacred artifact lying deep in the cave. However, in order to get through the different game levels, the player will be using our *unique selling point*, a ‘Hookshot’ mechanic. This main feature allows our player to either hook onto walls or cause damage to enemies.

2. Problems

2.1 Complex Hookshot Mechanic

As the main feature, the player will use it consistently in the gameplay loop, whether they are crossing obstacles, fighting an enemy, or traveling the level. This mechanic could very well determine if the game is fun or a failure depending on how well it is implemented.

We plan to simulate real-life physics by having it conserve the character momentum when swinging. However for the rope, to make it simpler, it will be static, not reacting to the environment,

The Hookshot can be broken down into two main steps, first attaching onto an unmovable object followed by using it as the pivot for swinging. However writing this in code is much more complex and will not suffice, the two steps will need to be further broken down into smaller steps to account for all the nuance in the physics. This will be further discussed as to how in the solutions

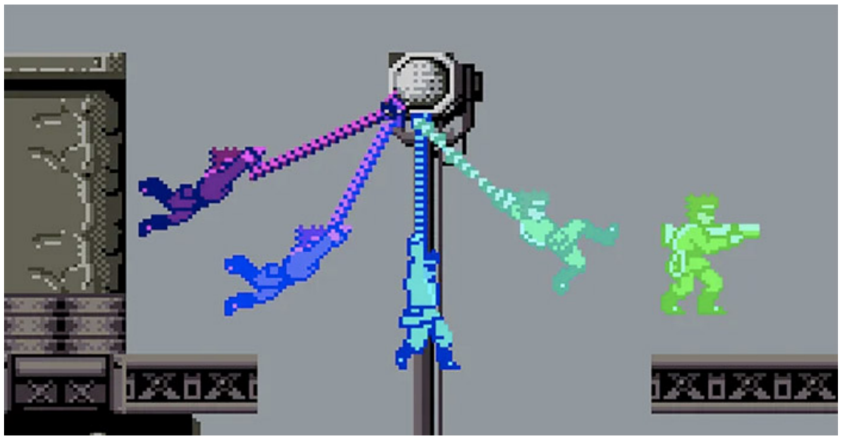


Image by FarPlanet (2020)

2.2 Enemy AI

One of the problems we will face will be developing the enemy ai. The reason for this is that compared to our previous project and assignments, The AI of our enemy is more intricate, and separated into 3 classes, each with its own behavior pattern distinct from the other. Because of this, we may face some difficulty due to the lack of experience.

By researching how to create efficient AI and studying enemy AI from other platformer games, it can give us an idea of how to program our ai. Another solution is to break down each enemy behavior into a flow chart and slowly adjust the code to fit our vision.

2.3 Animation Implementation

We have limited experience and knowledge in this area. Hence, our character animation may have a few major drawbacks. Firstly, our character needs to be able to do more than just idling in the same spot such as walking and jumping. Secondly, we need to establish these animations which requires a different frame rate depending on the character's action.

Therefore, in order to produce a smooth animation, this is a rather crucial process we need to understand and get used to.

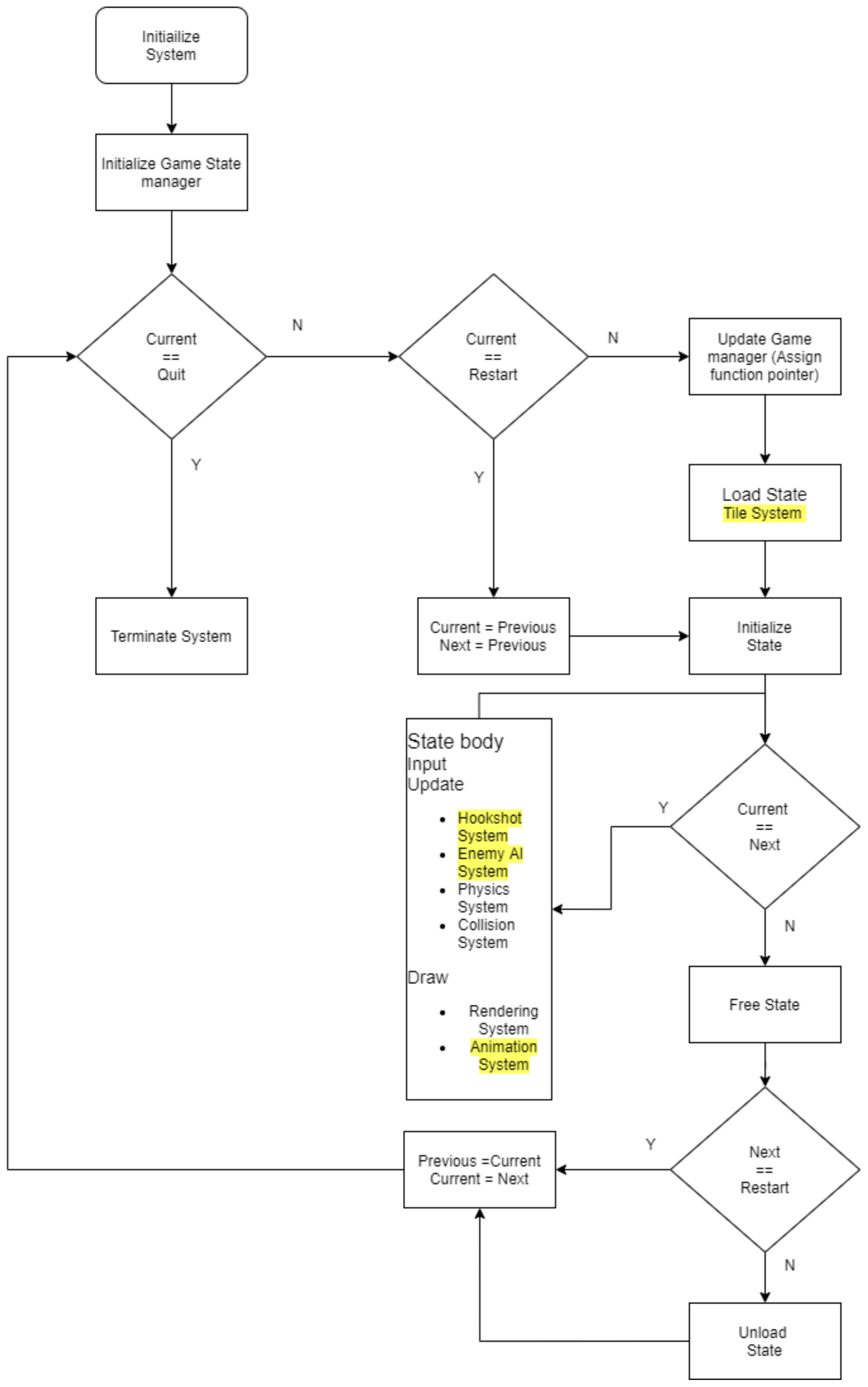
2.4 Level Generation

Hookshot, as a level-based game, will need individual levels, each with its own obstacles and terrain. Without a proper level generator or map editor, it will be extremely tedious and time-consuming to plot out where each object is positioned on the map, taking into account the width and the height of each individual object.

Therefore, rather than doing the calculation ourselves, it is best to leave it to the computer to do it for us where there is less room for error. Modern game engines usually have a map editor that allows for a drag and drop system. However, due to time constraints, we will be opting for a simpler editor that uses text-files to represent the level generation.

3. Integration of Solutions into the Engine

3.1 Engine Flow



Highlighted in yellow are the solutions that will be integrated into the engine

3.2 Engine System

- **Main:** Controls the flow of the engine, depending on the current/next state. States included are QUIT, RESTART, and LEVELs.
- **Game State Manager:** Updates the function pointers to the current game state. There are 6 function pointers.
  - Load()
  - Initialize()
  - Update
  - Draw()
  - Free()
  - Unload()
- **Load System:** Loads any data from files if any, for the current game state. Assigns any memory if required. Will not be called when restarting.
- **Initialize System:** Variables used for the current game state will be initialized in this function. Will be called again when restarting.
- **Free System:** Resets any values used or free objects that are no longer used. Will be called again when restarting.
- **Unload System:** Frees any memory used, especially for assets. This is to prevent memory leaks. Will not be called when restarting.
- **Physics System:** Motion of rigid bodies modeled using newton mechanics such as gravity, velocity, and acceleration, action, and reaction.
- **Collision System:** This system will be using axis-aligned bounding boxes or shapes to detect collision (rectangle to rectangle, rectangle to circle, or circle to circle).
- **Rendering System:** Adds colors and textures in the sprite and renders on the 2D screen as pixels, arranged in a 2D grid, which holds RGBA color.

- **[Solution] Hookshot System:** Contain the implementation for the Hookshot mechanic. From the firing mechanic to calculating the physics for the swing arc motion. Refer to 5.2 for more details.
- **[Solution] Animation System:** Contains all required animation and frame control functions for the game. This system creates the animation movement to portray walking, jumping, and other actions. Refer to 5.3 for details
- **[Solution] Enemy Ai System - Yong Hui:** The enemy AI controls how enemies move around, attack, and respond to certain events(such as players attacking or death).
- **[Solution] Tile System**: This system will help to separate the screen into multiple squares, each with its own index. It will also provide the conversion from the index into the global position and vice versa. It will also contain the function for loading the level data from a text file. Refer to 5.5 for more details.

4. Summary of Goals

4.1 Goals

- Overall Game goals
  - Basic working game engine
  - 3 balanced working levels (Not too hard or easy)
  - A fun and workable ‘Hookshot’ mechanic
- User Experience Goals
  - A useful tutorial level that guides player thoroughly
  - Users are able to tell where to go or how to clear the game

4.2 Non Goals

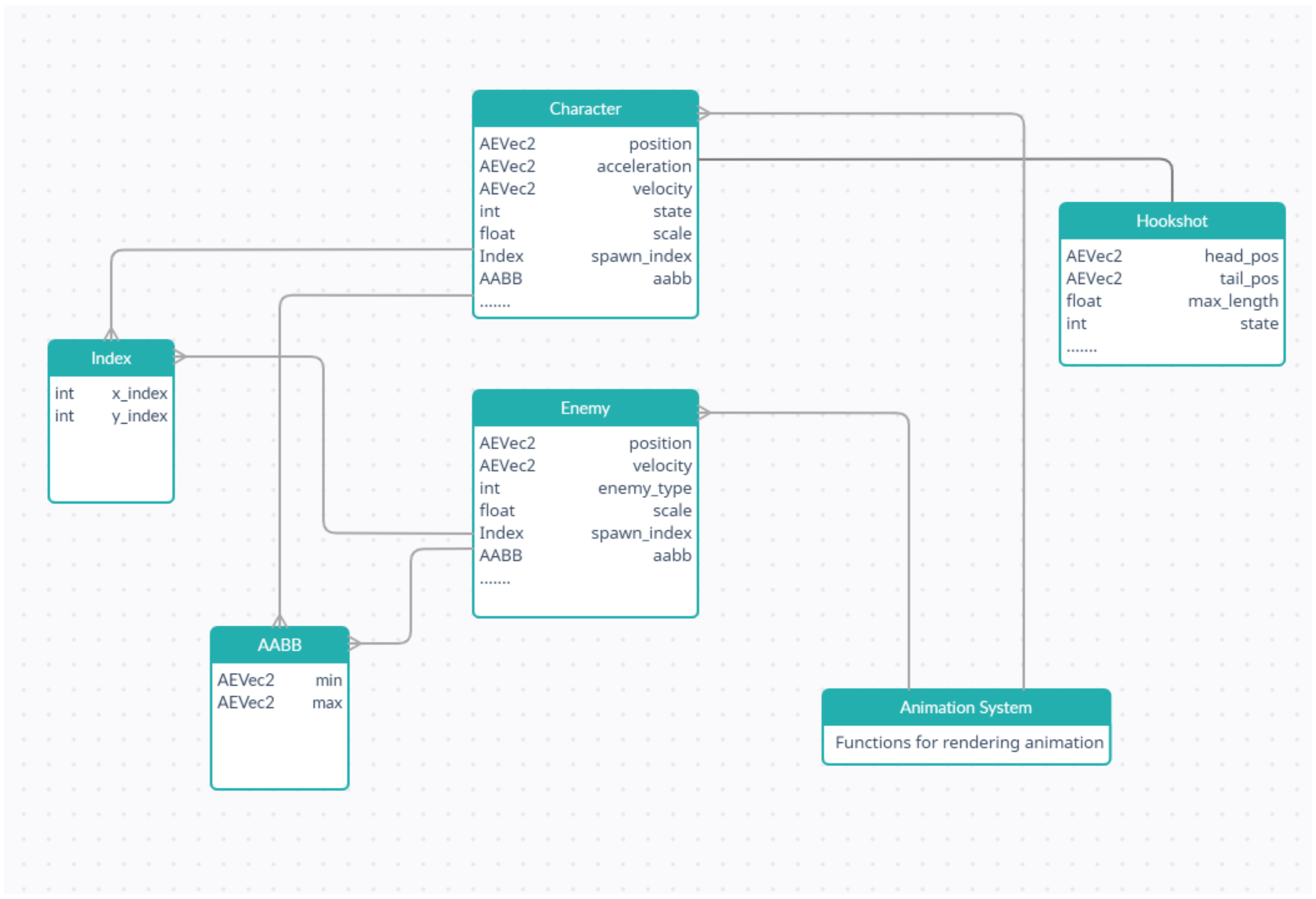
For this game project, we will not be supporting OpenGL as we currently do not have sufficient knowledge and experience in using it.

4.3 Future Goals

- More game features
  - Puzzles platforming - This is to increase level difficulties and make it more challenging for players
  - Time attack mode - A mode for players to challenge themselves.
  - Players will be able to find hidden areas in the different levels to upgrade their ‘Hookshot’ to swing further or have a higher damage output.
- OpenGL support

5. Solutions

5.1 Class Diagram



5.2 Solution to Complex Hookshot Mechanic

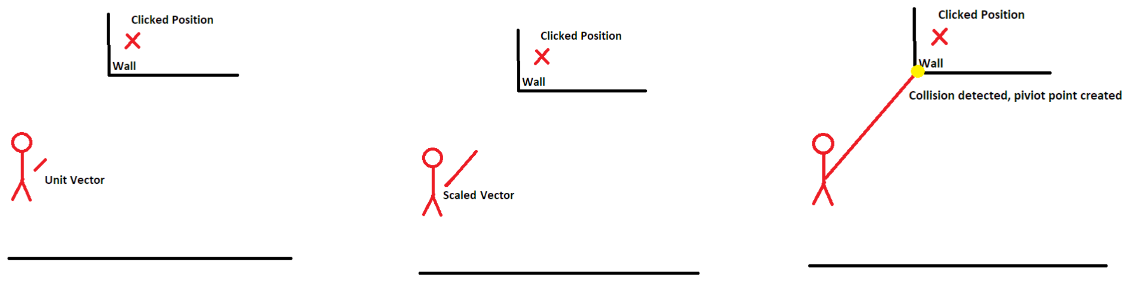
5.2.1 Explanation

As mentioned before, we will need to break down the Hookshot mechanic into smaller steps to see how we can implement it in our game.

Starting with the aiming of the Hookshot, it is required for the position selected to within range before the Hookshot can be fired. We can deduce this by using the two points, the character position and the mouse cursor position to find the distance using the Pythagorean theorem and comparing it with the set range limit.

Once the condition is fulfilled and the fire button is triggered. We will need to simulate casting a ray by getting the tether angle and scaling it with the Hookshot velocity. Once it collides with an object (if there are any, else it will use the open space) the pivot point will be created. If the button however is released before reaching its chosen position, the Hookshot will be canceled.

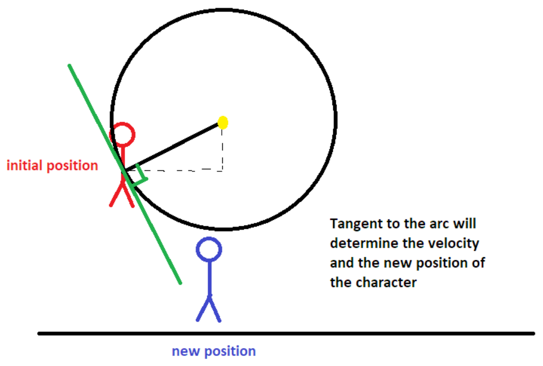
The tether angle can be represented as a unit vector relative to the character position. We will need to subtract the position of the pivot with the character position and divide it by the magnitude.

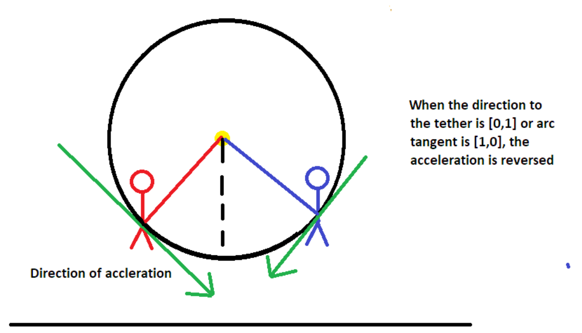


To calculate the physics we first need to get the tangent of the arc by taking the orthogonal of the previously calculated firing angle. This will give us an initial angle unit vector.

When first tethering, we will want to preserve the momentum by finding the magnitude of its current velocity and transforming it into a new direction. This can be done by scaling the angle unit vector by the magnitude

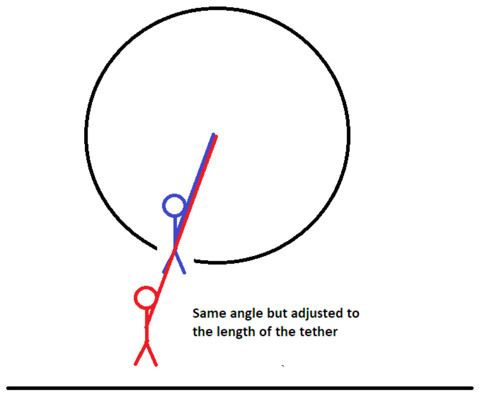
As for the acceleration, it will be fixed and based on the angle unit vector that is recalculated each frame. Only when the character is in the center of the arc will the acceleration be then inverted.


initial position
Tangent to the arc will determine the velocity and the new position of the character
new position


When the direction to the tether is [0,1] or arc tangent is [1,0], the acceleration is reversed
Direction of accleration

In this case, if the velocity is so high that the character goes beyond the tethered length we will need to adjust the character so that it lies on the arc again. This is illustrated with an algorithm by Jamie Fristrom (2013).

```
if( amITethered )
{
    if (testPosition - tetherPoint ).Length() > tetherLength )
    {
        // we're past the end of our rope
        // pull the avatar back in.
        testPosition = (testPosition - tetherPoint).Normalized() * tetherLength;
    }
}
```



The algorithm illustrated.

This will be repeated each time starting from the recalculation of the angle unit vector for the acceleration to the adjusting of the character based on the velocity and the tether length. Only when the firing button is released, will the loop stop, the acceleration set to zero, and the player no longer restricted by the arc.

5.2.2 Pseudo Code

```
GET mouse cursor position
GET character position
CALCULATE aim distance using character position and mouse cursor position

IF (aim distance < set range limit AND right button is triggered)
    SET Hookshot object flag to active
    ASSIGN selected position = mouse cursor position
    SET head position of hook to character position
    SET tail position of hook to character position
    SET current hook length to zero
    SET is state to firing t

IF (state is firing)
    GET character position
    ASSIGN tail position = character position
    CALCULATE tether angle using selected position and character position
    INCREASE current length
    CALCULATE new position of head  …. tail position + (current length * tether angle.)

    If (position collided with another object)
        SET pivot point to collision point
        SET state to first tethered

    IF (current hook length > max hook length)
        SET pivot point to current head position
        SET is state to first tethered

IF (state is first tethered or tethered)
    CALCULATE tether angle using pivot point and character position
    CALCULATE arc tangent by taking the orthogonal of the tether angle.

    If (state is first tethered)
        CALCULATE the magnitude of the current velocity
        SET character velocity ….. magnitude * orthogonal angle
        SET state to tethered.

    SET character acceleration …. acceleration * orthogonal angle
    CALCULATE new position using acceleration and velocity.

    APPLY Jamie Fristrom algorithm, if the new position is outside the arc and pull it back

IF (right button is released)
    SET Hookshot object flag to not active.
    SET state to not active.
```

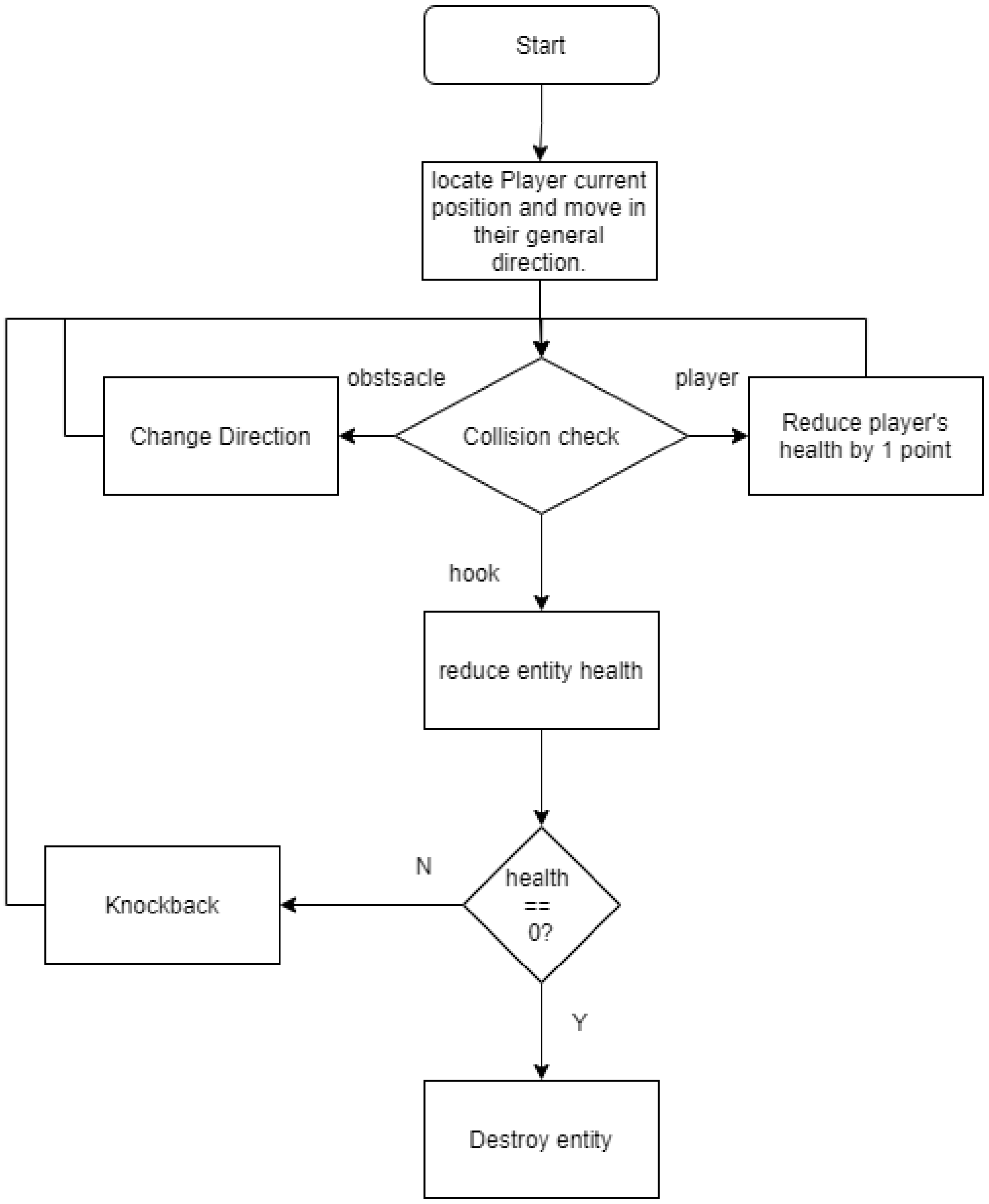
5.2.3 Structure Table

Hookshot	
Type	**Variable Name**
bool	flag
int	hook_state
AEVec2	head_pos
AEVec2	cen_pos
AEVec2	tail_pos
float	max_length
float	curr_length
AEVec2	pivot_pos
AEVec2	arc_tan
AEVec2	pivot_angle

5.3 Solution to Enemy AI

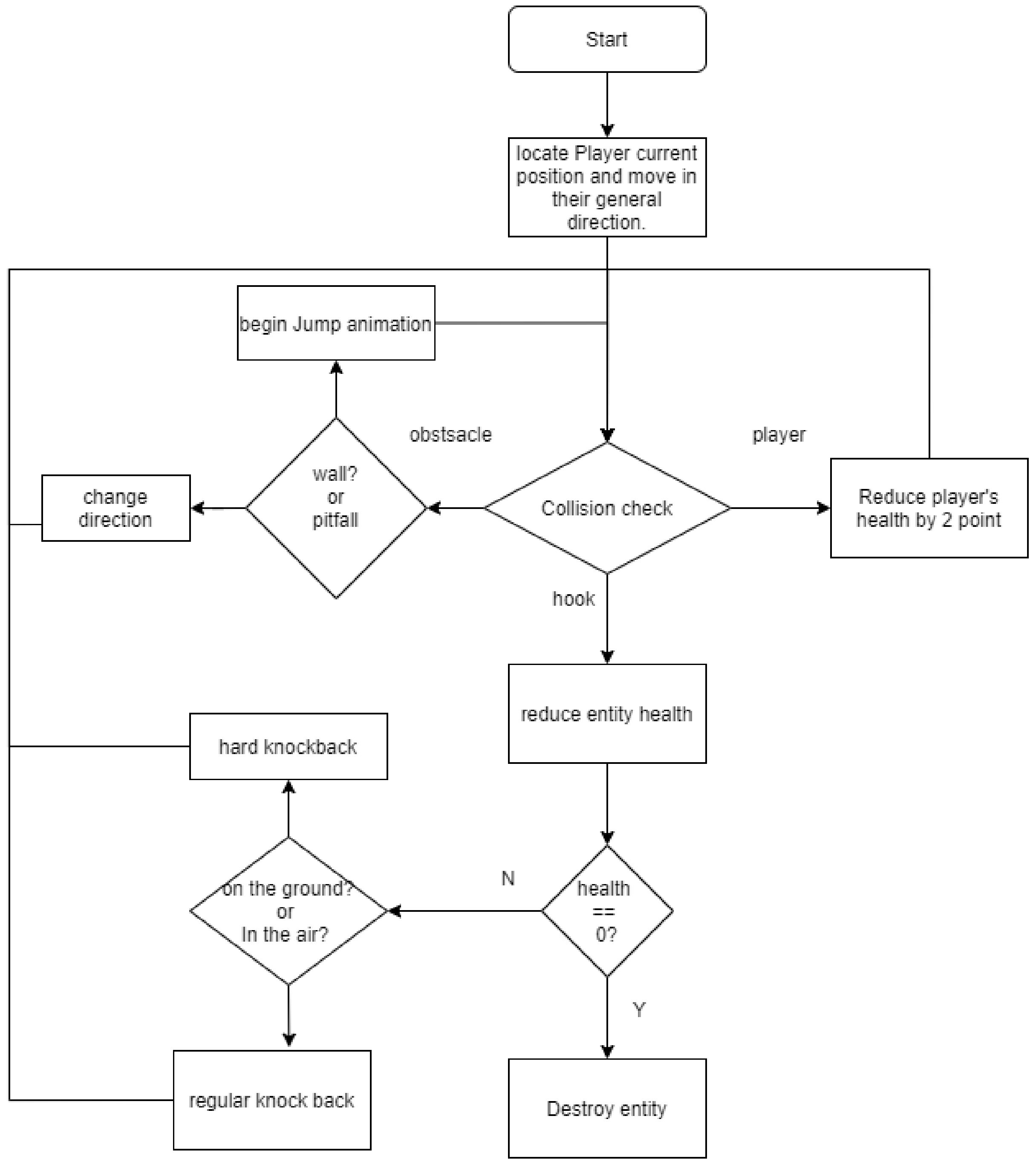
After doing some research, We have come up with some possible solutions for the way our AI works. Most of them are rough pseudo-codes with high chances of changes during the lifespan of our game development.

5.3.1 Skitter AI



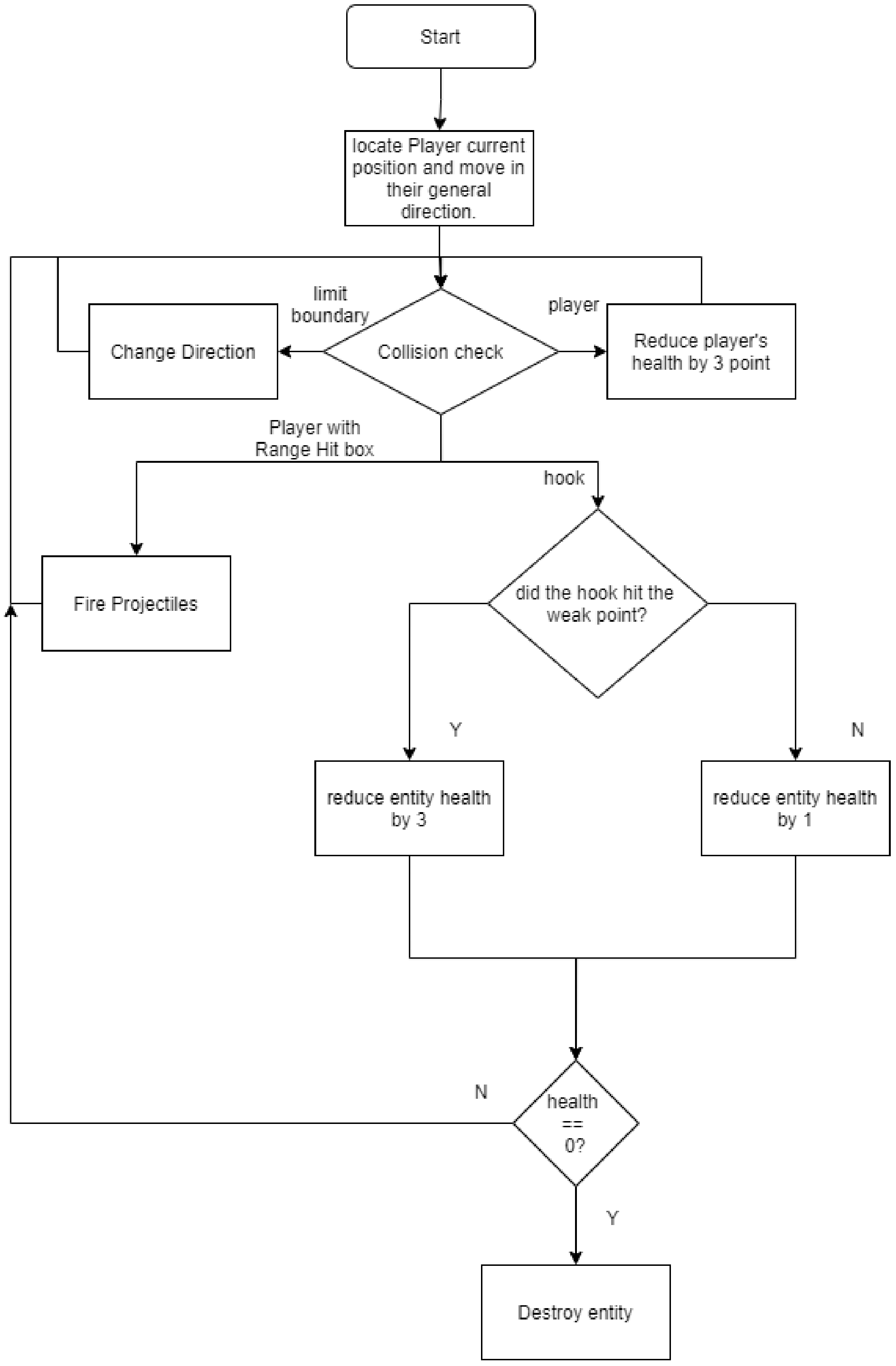
The skitter AI is the simplest and easiest among the 3 the AI. The reason behind this decision is because the Skitter is the most basic and common enemy in the game, and it is meant to ease the player into the combat system. Their movement is simply to move in the player's general direction and only change their direction when they encounter an obstacle like a cliff or a wall.

5.3.2 Hopper AI



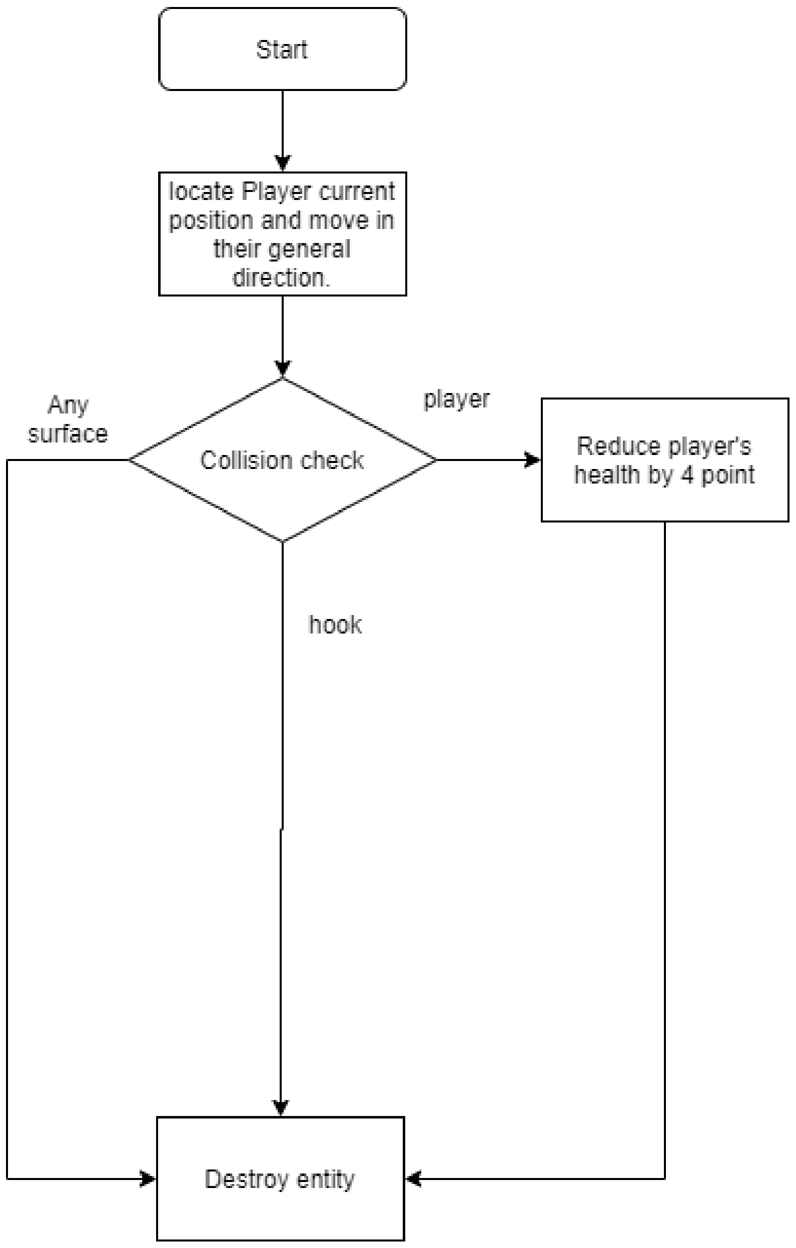
The Hopper is the second class of enemy the player will encounter. As the name suggests this monster moves by performing small hops. When it encounters a ledge, it will perform a leap to cross pitfall, unlike the Skitters. When this monster is hit while still in the air it will receive a higher knockback allowing the player to potentially defeat the enemy by simply knocking them into the pitfall while they are still airborne.

5.3.3 Titan AI


Start
locate Player current position and move in their general direction.
limit boundary
Change Direction
Collision check
player
Reduce player's health by 3 point
Player with Range Hit box
hook
Fire Projectiles
did the hook hit the weak point?
Y
N
reduce entity health by 3
reduce entity health by 1
N
health == 0?
Y
Destroy entity

The Titan AI is by far the Most Intricate. If the player is within range The Titan will fire off a project that will fly in the player's general direction. The titan will also have a weak point on its back which causes the entity to take more damage than usual. Its movement is also restricted and will be paced back and forth in its spawned location.

5.3.4 Projectile


Start
locate Player current position and move in their general direction.
Collision check
Any surface
player
Reduce player's health by 4 point
hook
Destroy entity

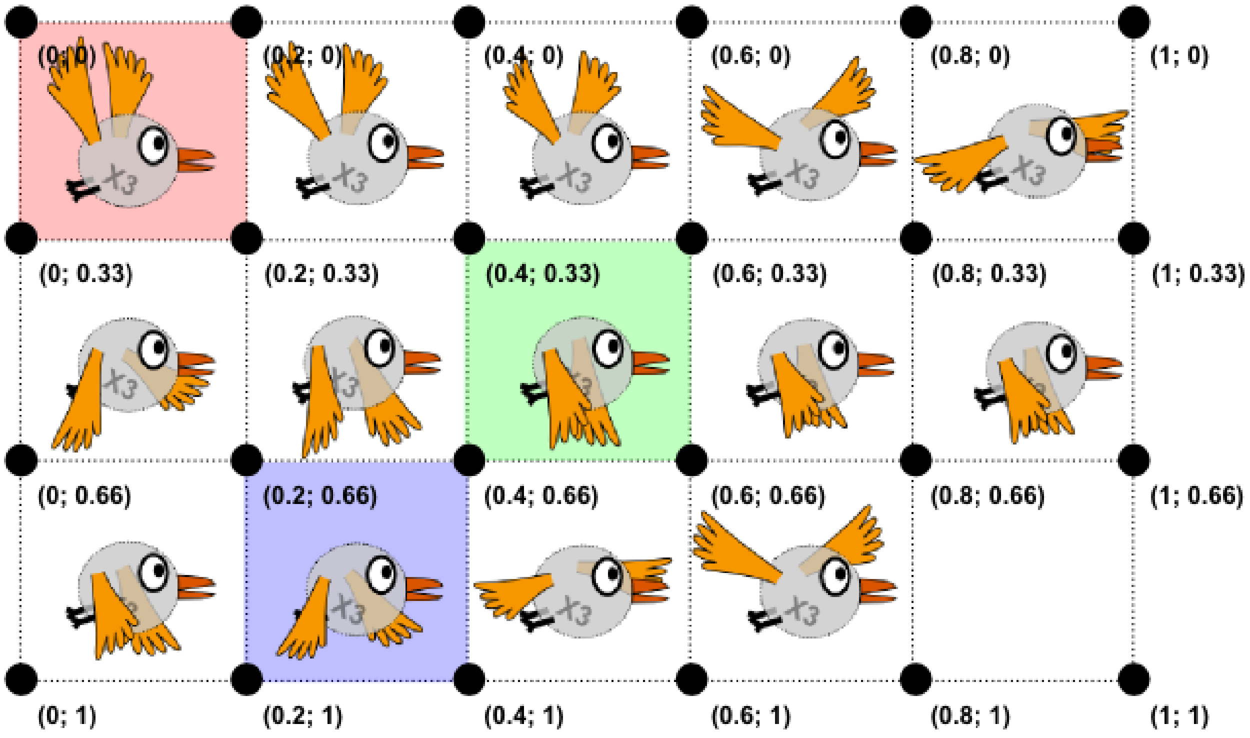
The projectile will fly directly towards the player's last position when it first spawns. The player can either try to dodge the projectile or destroy it with their hook.

5.3.5 Structure Table

Enemy	
Type	**Variable Name**
bool	flag
int	enemy_type [4 different types]
int	health
int	damage
float	scale
float	gravity
AABB	aabb
AEVec2	velocity
AEVec2	position
int	spawn_index
bool	knock_back

5.4 Solution to Animation Implementation

There are 2 things needed, creating a sprite sheet for the object or creating the UV values. A sprite sheet that contains images of several smaller images (sprites). Combining the small images into one big image improves the game performance. UV mapping coordinates are needed for each corner of the sprite. By interpolating between these to get a set of UV coordinates for each pixel.



Sprite sheet with UV mapping coordinates. Image from Raki (2014)

There will be a few items to be added in.

- Creating objects:
  1. Begin by creating the object using AEGfxMeshStart() function. This is to start building up the mesh for the shape
  2. Secondly draw the shapes using multiple squares, rectangles and triangles using AEGfxQUAD, AEGfxTri function(mainly for the map or background).

  These 2 functions for drawing will be one of the most crucial for our game. It needs to simulate the characters and the monsters

- Implement control for characters:

  For implementing controls, it begins by first initiating the start of the loop.

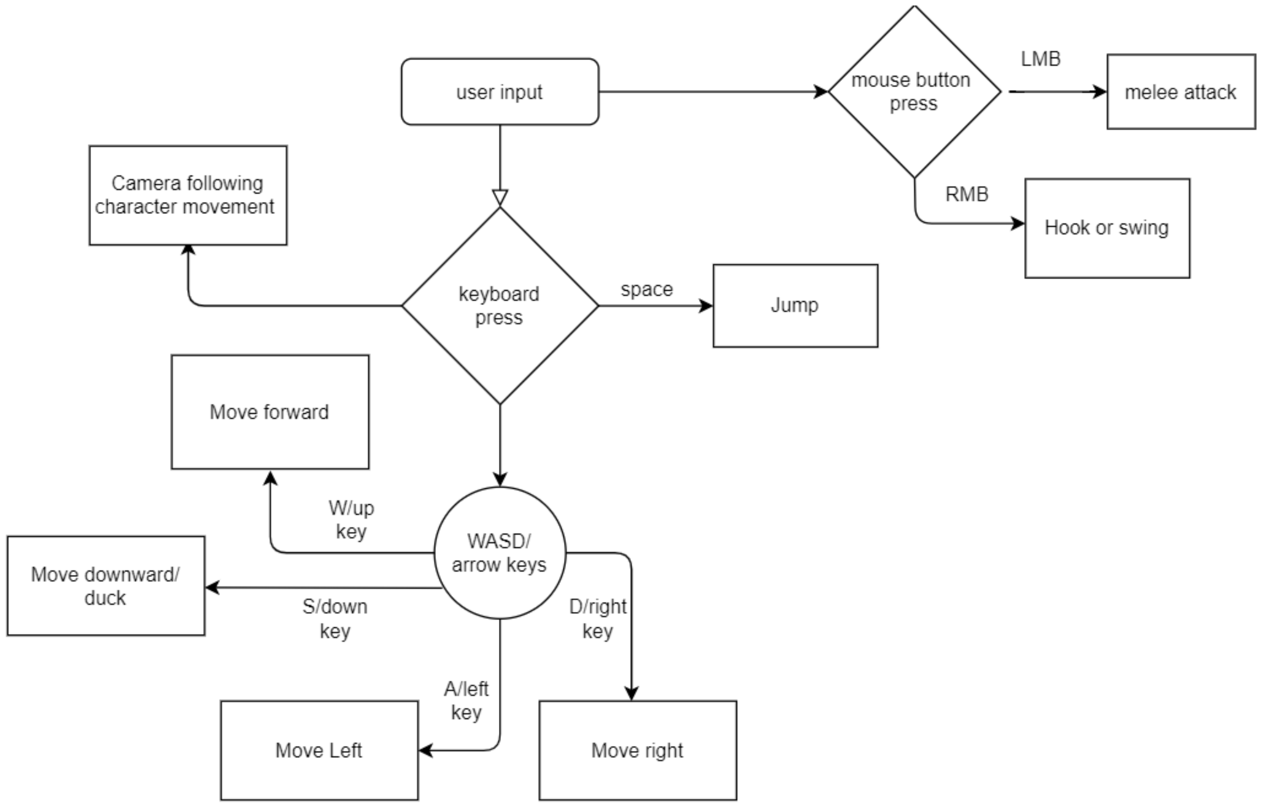
  1. Use the AESysFrameStart function to begin by informing the system that the loop has started.
  2. Next by using AEInputupdate function to handle inputs.
     Throughout the loop there will be several functions being used for various inputs.

- Blending mode in the game loop update:

  AEInputCheckCurr(), this function checks the current input key or value. There are the 4 main movement keys up, down , left and right. There will be more such as throw and duck.

  Various objects have different keys such as main character and hand device.

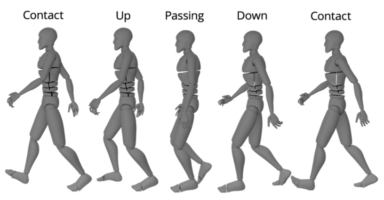
  AEGfxGetCamPosition()/camera movement: Getting camera movements, ensuring the camera moves along with the character.



Actions	**Keys**
up/jump	w button/up arrow
down/duck	s button/down arrow
move left	a button/left arrow
move right	d button/right arrow
melee attack	left mouse key
swinging or hooking	right mouse key

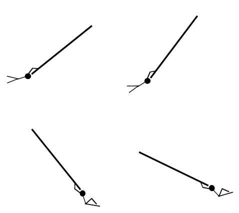
Drawing Objects Functions	
AEGfxSetRenderMode(AE_GFX_RM_COLOR)	set the type of colour
AEGfxPosition()	set the position for the object
AEGfxtexure()	set the texture of the object
AEGfxMeshDraw()	drawing out the mesh of the objects using mesh lines
AEGfxSetTransparency()	setting transparency of the objects
AEGfxTexture * AEGfxTextureLoad()	function to load texture from file.
AEGfxTextureSet()	function to set the texture for rendering and changing of uv coordinates. This function will be used widely especially for loading the sprites at different time frames and at different positions. Basic movements such as running and jumping have an estimation of 5 to 7 image sprites for each movement. Hooking and swinging will have an estimation of 7 to 10 images sprites.

WALKING:



RUNNING:

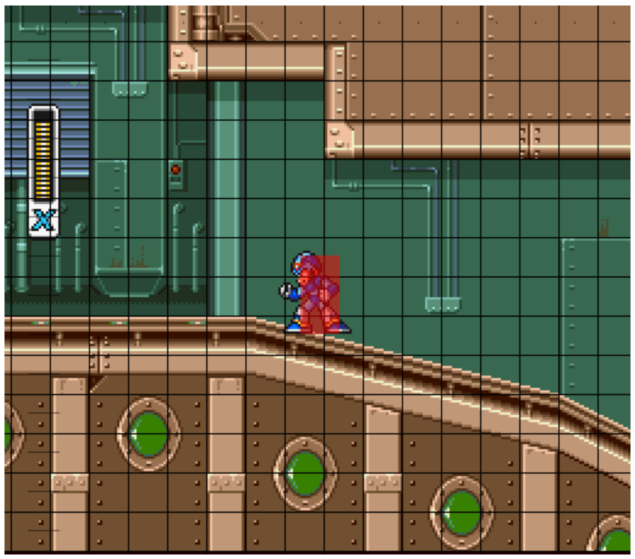
SWINGING:



5.5 Solution to Level Generation

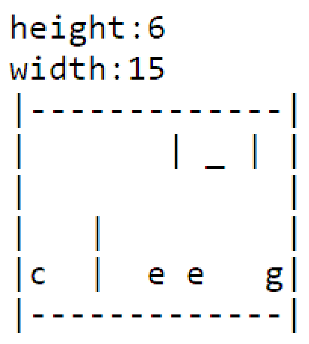
5.5.1 Explanation

As mentioned before, due to time constraints, we will be opting for a map editor that uses text-files to represent the level generation. First, the map will be first divided into multiple tiles to standardize the width and the height of each static object. By doing so each tile will have its own index instead of a global position. Moving objects like the character and enemies will need to have free-movement, hence they will only use the tile for their initial spawning location.


Mega Man X Tile System. Image by Rodrigo Monteiro (2013)

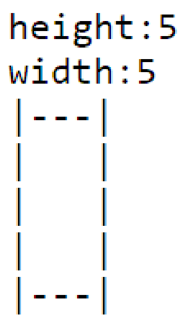
Next, each tile will have to be represented in a text-file with the matching object. It should be done with a one to one scale with a single character representing each tile. A unique ASCII key will be used to recognise the type of object placed in the level. This however means that the total type of objects is limited to 256, the number of ASCII characters. Another limitation is that the text file characters are not square, resulting in a distorted view.

The height and width will also be stated at the top for the maximum index of the level.



```
height:6
width:15
|-------------|
|       | _ | |
|             |
|   |         |
|c  |  e e   g|
|-------------|
```

Sample level in txt file



```
height:5
width:5
|---|
|   |
|   |
|   |
|---|
```

Distorted View of a Square

Lastly is the conversion from the text file to the tile system. This process will first begin in the load system of the engine. It will follow the pseudo code as stated below:

5.5.2 Pseudo-Code

```
OPEN text file
READ first line.
ASSIGN max height
READ second line
ASSIGN max width

SET current y index to (max height - 1)
SET current x index to 0

WHILE (current y index > -1)

    READ line *(should be starting from the third line)

    WHILE (current x index < max width)
        READ character

        SWITCH condition based on the character, ASCII.
            *Depending on character.
            SET flag active to true;
            ASSIGN starting index position to current y index and current x index

        GOTO next character
        INCREASE current x index
    end of loop

    GOTO next line
    SET current x index to 0
    DECREASE y index
```

end of loop

5.5.3 Structure Table

Index	
Type	Variable Name
size_t	y_index
size_t	x_index

6. Development Plan

6.1 Work Estimates

Epic	Description	Estimated Time
Engine building	Rendering system, load system & basic map editor, physics system, 'Hookshot' mechanic, collision system, camera system, input system, user interface, enemy AI.	20 days
The building of game Level	Crafting, designing, and building game levels using the game engine.	5 days
Polish of game	Game levels are polish based on user testing results.	7 days

6.2 Roll-out Plan

The basic engine building will happen from week 3 to the end of week 6. Afterwards, on week 7, we will be polishing our engine and 'Hookshot' mechanic. When the game engine is done, it will be used to build the game levels from week 8 to the end of week 9. Then, from week 10 to 13, we will be carrying out user testing and polishing of our game.

🟧	-	Everyone
🟥	-	Lau Yong Hui
🟦	-	Liu Ke
🟪	-	Wei Wen Tan
🟩	-	Tan Egi
⬛	-	<TBC>

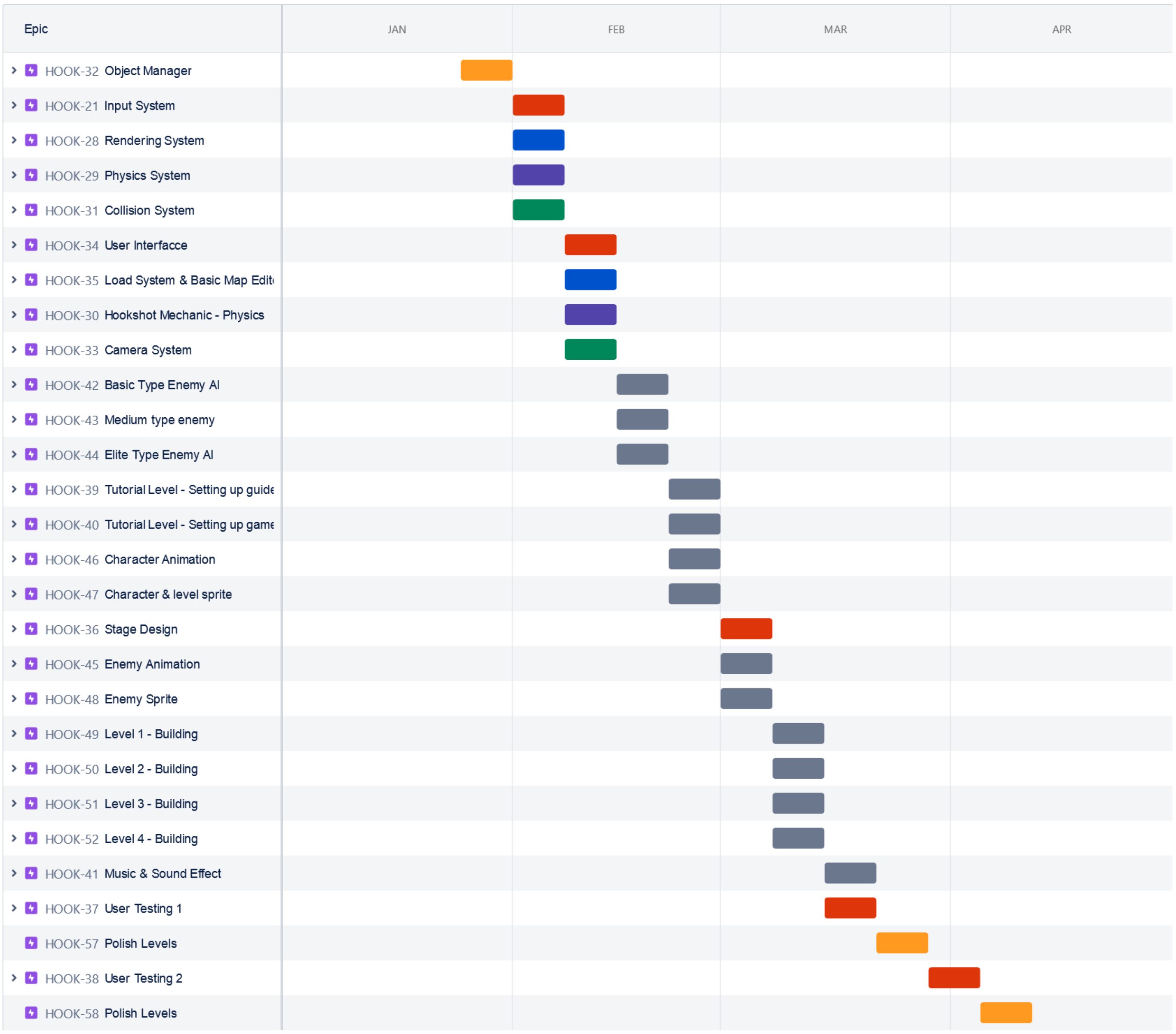


Image of Gantt bar chart for Hookshot

6.3 Risk Management

Risk	Description	Mitigation
Potentially over-reliant on ‘Hookshot’ mechanic	The game is built around this mechanic as it is our unique selling point to make the game stand out. However, if it is not well implemented, the game may end up being monotonous.	We will be relying heavily on user testing and player feedback to improve the ‘Hookshot’ mechanic.
High technical knowledge is required	This game requires lots of physics and math calculation for the ‘Hookshot’ mechanic as our knowledge is still currently lacking, our engine may end up taking a long time to complete.	Each feature is given a week to complete. Hence, before implementing, we will be doing all the necessary research to find different coding methods first.

7. Measuring Success

We will be relying heavily on user testing to measure the success of the overall game. For example, based on the player’s feedback, we will be able to tell if the overall game or user experience goals are achieved. To explain this further, refer below for a list of questions linked to our project goals.

- Overall Game Experience Goals
  - Are there any bugs?
  - What is your first impression of the ‘Hookshot’ mechanic?
  - Is the game too easy or too hard?
- User Experience Goals
  - To what extent is our tutorial level useful?
  - Do you know what to do?
  - Are there any confusing parts about the game?
  - What can the ‘Hookshot’ mechanic do?

Therefore, by using this list, we will be able to determine if our game is a success or not.

As for measuring the success of the game engine, we will be taking into account how long it takes to build a basic game level. For example, if level generation, animation, and enemy AI are established properly, the building of a game level would at most take a day to complete. However, if we were to go back to the base engine to make major changes, it means that the game engine was not established properly. Hence, affecting and slowing down the game level building process. Therefore the duration of how long a game level is built determines how successful the game engine is.

8. References

Rodrigo, M. (2013, April 05) *The Guide to Implementing 2D Platformers*. Retrieved online from https://www.gamedev.net/articles/programming/general-and-gameplay-programming/the-guide-to-implementing-2d-platformers-r2936/

Fristrom, J. (2013, June 05) *Swinging Physics for Player Movement (As Seen in Spider-Man 2 and Energy Hook)*. Retrieved online from https://gamedevelopment.tutsplus.com/tutorials/swinging-physics-for-player-movement-as-seen-in-spider-man-2-and-energy-hook--gamedev-8782

FarPlanet. (2020, 9 October) *Grappling Hooks: The Integral and Defining Feature of Gaming.* Retrieved online from https://the-artifice.com/grappling-hooks/

Raki. (2014, 20 September). *Generating UV coordinates for texture atlas*. Retrieved online from https://stackoverflow.com/questions/25945930/generating-uv-coordinates-for-texture-atlas