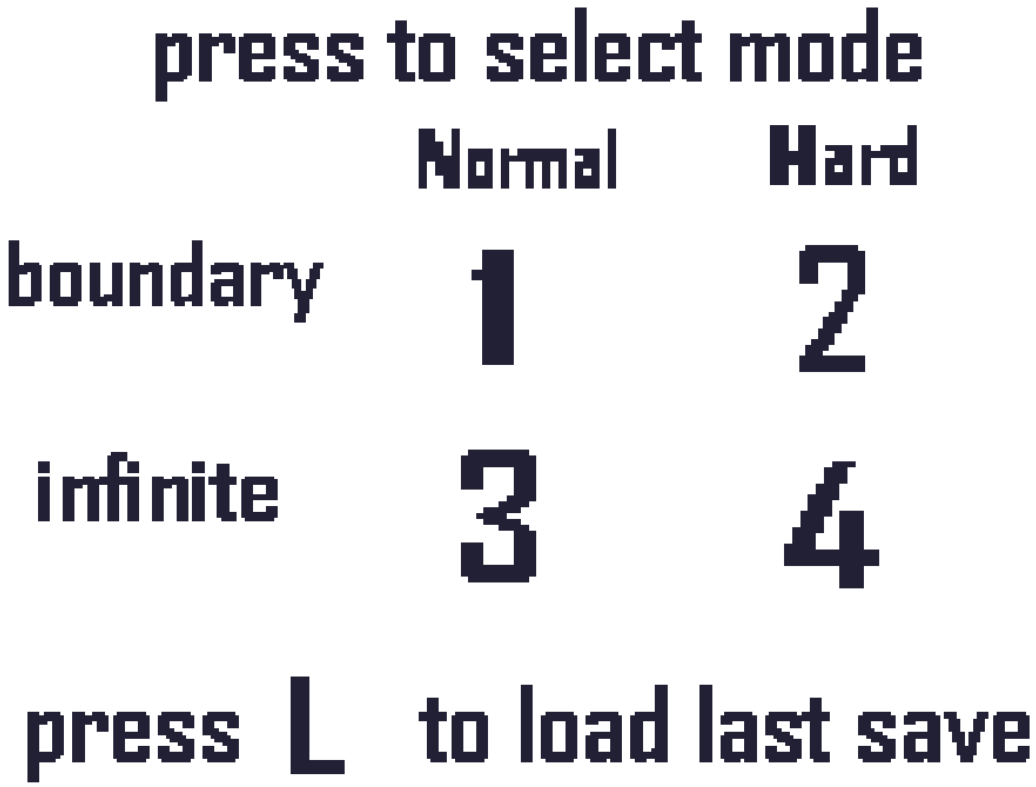
Programming and Fundamental Algorithms Report

1. Introduction

1.1How to play

1.1.1Menu

The game uses buttons to control all operations, starting with the UI interface when you first enter the game.：



You can press the number form 1 to 4 to choose the mode you want.

Number 1:Map with the fixed boundary which is 42x42, maximum number of enemies is 20.

Number 2:Map with the fixed boundary which is 60x60 , maximum number of enemies is 50.

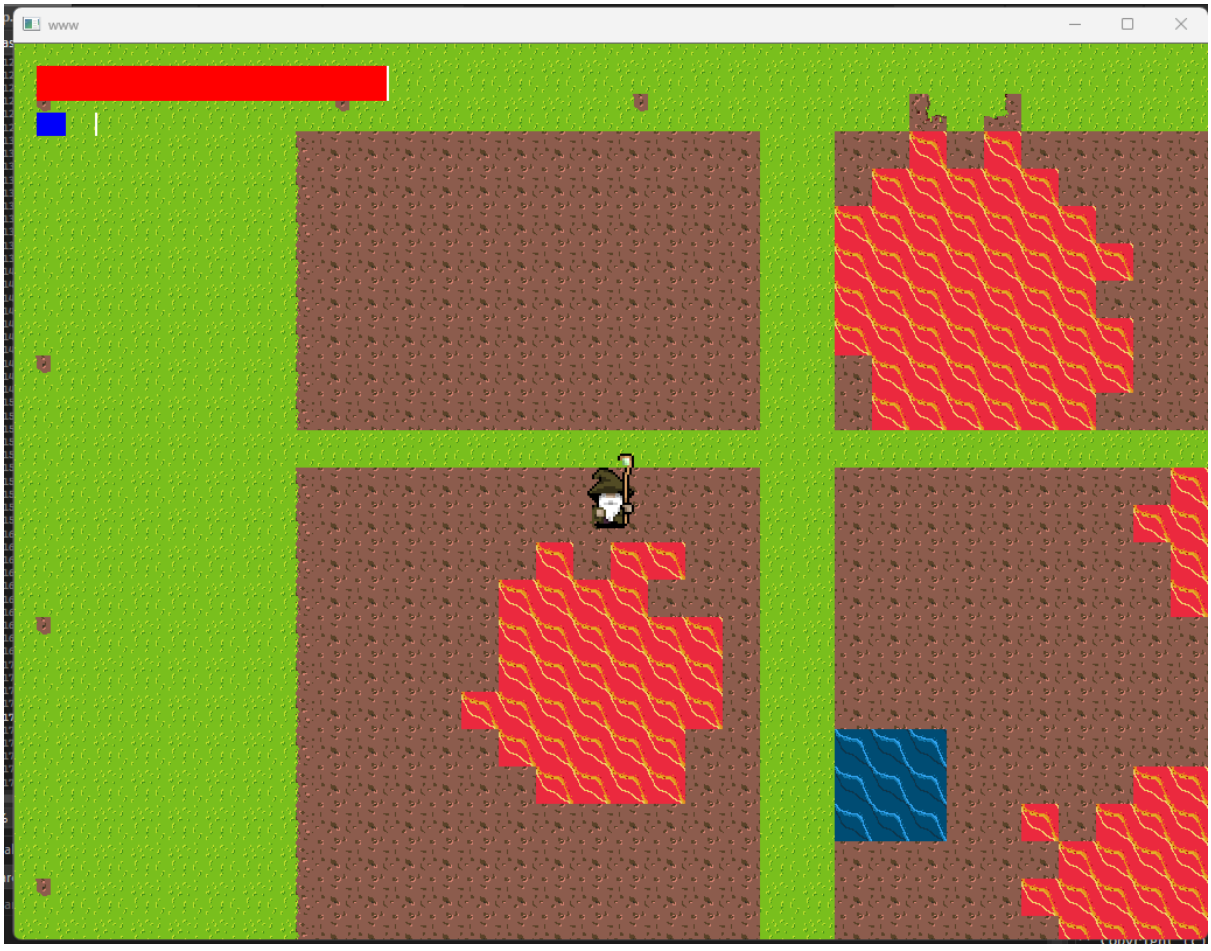
Number 3:Map has no boundary but use the same map as number 1, maximum number of enemies is 20.

Number 4: Map has no boundary but use the same map as number 2, maximum number of enemies is 50.

Press ‘L’ :Load the last save ,it will load the mode ,map ,game time ,hero status and so on ,which is all the same with last time you press the save button(press ‘M’).You can only load the save at the menu.

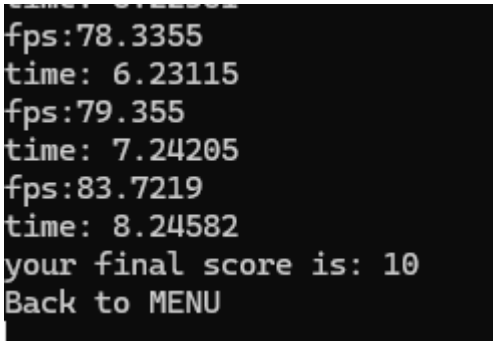
1.1.2 Game running

After you choose a mode and press the button ,you will start to play the game ,here is the screenshot wen you start the game:



The red strip at the top left corner is your health bar. The bule one is your aoe attack cooldown bar(when it runs to the threshold, then you can use it).

The fps, score and time will print automatically at the terminal, when you exit the game, if will shows your final score.



You can press the button to control your hero:

Press 'W': Move up.

Press 'S': Move down.

Press 'A' : Move left.

Press 'D' :Move right.

Press ‘J’ :Use a special area of effect attack, it will find the enemy within the fixed range who has the highest health, it will attack 5 enemies.

Press ‘esc’: Pause the game, you can press it again to continue the game.

Press ‘M’: This will save your game status to the save file,!!!you can only use it when you pause the game, if not press ‘esc’ to pause the game, nothing will happened when you press the ‘M’.

Press ‘Q’: You can press ‘Q’ at the Menu or when you paused the game, then you will exit the program or exit the game.

When the score exceeds the 100, an upgrade tile will be create at a random place of the map, when hero pick it up, it will increase your attack speed and the amount your AOE can attack.

Upgrade pictrue:  After pick up, the terminal shows : 

When the game’s time runs out, it will automatically pause, the score will print at the terminal, you can press ‘esc’ now to continue the game till your hero is die. When your hero is dead, game will paused, you can press ‘Q’ to exit.

1.1.3 Hero

Hero has five appearances when it move to the four directrions and die.

Left: 

Right: 

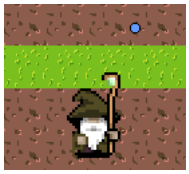
Up: 

Down: 

Die: 

Hero has two types of attack, One type is automatic attack, another one is AOE attack.

Automatic attack:



Hero will launch the blue projectiles (●) to attack the enemy, it is linear attack that targets closest enemy.

AOE attack:

After you press the 'J', the lighting will attack the enemies and make a huge damage to the enemies.

1.1.4 Enemies

There are 4 types of enemies:

Enemy type	Name	Health	Speed	Attack
	Slime	medium	medium	medium
	Bug	low	fast	low
	FlySpookmoth	medium	static	medium
	Pebblin	high	slow	high

The FlySpookmoth enemy can't move, but will launch the red projectiles to the hero. The projectiles will follow the hero, but as the time runs, it will disappear.

The rest types of the enemy will move to the hero, when it touches the hero, the hero will get hurt from them.

1.1.4 Map

There are four types of map tile:

Grass: Road: Water: Magma:




Hero can move safely when it at the Grass, Road and Water tiles. But when it touch the Magma tile, it will get hurt and be rebound form the Magma.

1.2 The technologies

1.2.1 Camera

There is a virtual camera that follows the protagonist, with the protagonist in the center. When the camera touches the boundary, the protagonist will no longer be in the center, but will move to its real position relative to the boundary.

1.2.2 Hero

The hero has different appearance when it at different states.

Hero launch a linear attack that targets closest enemy – the attack runs automatically all the time.

A special area of effect attack that targets the top 5 max health enemy – triggerred by press 'J'.

The projectiles from hero will be destroy when it makes contact with the enemy.

When hero contact to the Magma tile, it will be rebound from the magma, this would also appear with enemy contact other enemy.

1.2.3 Enemy

Enemies will always move to the hero's location, by calculate the distance from the hero to itself.

The enemy will rebound other enemies to avoid their positions overlapping.

In the infinite Map, enemy will judge the position to find the nearest way to the hero.

1.2.4 Map

A tile-based method for displaying the background.

Data driven level loading, by the txt file.

The program will automatically adapt to changes in map size and other parameters during displaying.

The infinite map has to draw the 9 times of the map at different directions,but only draw the pixel which is in the canvas.

1.2.5 Save file

Save file will store all the states detail of each state, each enemy and projectile.

The different unit will find their own date by the keyword.

2. Technique Details

2.1 Class structure

This section, by explaining the implementation and functionality of the classes, will be of great help when discussing the implemented technologies later, especially in the part about managing enemy and bullet states. This section will not show complete information about the class, but only the elements it contains to illustrate the class's purpose.

Unit

First of all is the “Unit”” class, this is a basic class for every object which needs a hitbox, such as: hero, enemy, bullet and tiles which hero can’t pass.

This class only has a get function to retrieve its elements.

```
//The class of every unit which has a hitbox, include the main charactor and
enemies
class Unit
{
protected:
	float locate_x, locate_y;  //Top left position
	unsigned int hitbox;	// hitbox radius
	float hitbox_center_x;	// hitbox center position
	float hitbox_center_y;
	Unit_Type unit_type;	//unit type

public:
	//Get the unit's top left X position
	inline float get_x() { return locate_x; }

	//Get the unit's top left Y position
	inline float get_y() {	return locate_y; }

	//Get the unit's hitbox radius
	inline unsigned int get_hitbox() { return hitbox; }

	//Get the unit's hitbox center X position
	inline float get_hitbox_x() { return hitbox_center_x; }

	//Get the unit's hitbox center Y position
	inline float get_hitbox_y() { return hitbox_center_y; }

	//Get the inmage's width
	virtual unsigned int get_width() = 0;

	//Get the image's height
	virtual unsigned int get_height() = 0;


	~Unit() {}
};
```

Unit_Type is a enum class to represent the type of Unit.

```
enum class Unit_Type {
	Hero,
	Enenmy,
```

```
        Bullet,
        Trap,
        Null
};
```

Character

The “Character” class is for hero object.It public Inherited from the Unit class,this class store the basic data of the player-controls character .

```
enum class Move_Status {
        Front = 0,
        Back = 1,
        Left = 2,
        Right = 3,
        Dead = 4,
        MAX_TYPES
};



class Charactor : public Unit
{

private:
        GamesEngineeringBase::Image
image[static_cast<int>(Move_Status::MAX_TYPES)];
        int health; //charactor's health,storage the current health num
        unsigned int speed; //move speed
        unsigned int attack;
        float attack_cd;
        float aoe_cd;
        float aoe_range = 300.f;
        unsigned int aoe_num = 5;
        float invincible_time;
        Move_Status m_status;
};
```

Enemy

The “Enemy” class is for enemy object.It public Inherited from the Unit class:

```
const unsigned int enemy_move_status_num = 4;
class Enemy : public Unit
{
        std::string enemy_name;
        Enemy_type type;
        GamesEngineeringBase::Image image[enemy_move_status_num];
        int health; //enemy's health,storage the current health num
        unsigned int speed;
        unsigned int attack;
        unsigned int attack_cd = 0;
        Move_Status m_status;

}
```

The “Enemy_index” class is to store the different enemy’s detail, when you want to know enemy’s detail, you just need to know the enemy’s name of enemy’s type, then you can get all data by the “Enemy_index” instances.

```
enum class Enemy_type
```

```
{
	Slime = 0,
	Bug = 1,
	FlySpookmoth = 2,
	Pebblin = 3,
	MAX_TYPES
};

struct Enemy_data {
	std::string name;
	Enemy_type type;
	int health;
	unsigned int speed;
	unsigned int attack;
	unsigned int attack_cd;
};

class Enemy_index
{
	unsigned int num = static_cast<unsigned int>(Enemy_type::MAX_TYPES);
	Enemy_data enemy_templates[static_cast<unsigned
int>(Enemy_type::MAX_TYPES)] = {
	{"Slime",		Enemy_type::Slime,		10,	50, 3, 0},
	{"Bug",			Enemy_type::Bug,		5,	100, 1, 0},
	{"FlySpookmoth",	Enemy_type::FlySpookmoth,	10,	0, 5, 2},
	{"Pebblin",		Enemy_type::Pebblin,		20,	10, 5, 0}
	};

public:

	unsigned int const get_enemy_index_num() { return num; }

	Enemy_type string_to_enemy_type(std::string name)
	{
		for (unsigned int i = 0; i < static_cast<unsigned
int>(Enemy_type::MAX_TYPES); i++)
		{
			if (name == enemy_templates[i].name)
				return static_cast<Enemy_type>(i);
		}
	}

	std::string enemy_type_to_string(Enemy_type e_type)
	{
		return enemy_templates[static_cast<unsigned int>(e_type)].name;
	}

	Enemy_data& operator[] (unsigned int index) { return
enemy_templates[index]; }
	Enemy_data& operator[] (Enemy_type name) { return
enemy_templates[static_cast<int>(name)]; }
};
```

Bullet

The “Bullet” class is used to store the data of the porjectiles, it store the type of where it form to identify who will be hit by the bullet, the health of the bullet is used for calculate how long it will exsit.

```
const unsigned int bullet_move_status_num = 1;
class Bullet : public Unit
{
```

```
	Bullet_index b_index;
	std::string bullet_name;
	Bullet_type type;
	Unit_Type from;
	GamesEngineeringBase::Image image[bullet_move_status_num];
	float health = 5;
	unsigned int speed;
	unsigned int attack;
	Move_Status m_status = Move_Status::Front;

};
```

Bullet also has a Bullet_index to store different types:

```
enum class Bullet_type
{
	Blue = 0,
	Red = 1,
	Light = 2, //Lighting
	MAX_TYPES
};

struct Bullet_data {
	std::string name;
	Bullet_type type;
	unsigned int speed;
	unsigned int attack;
};

class Bullet_index
{
	unsigned int num = static_cast<unsigned int>(Bullet_type::MAX_TYPES);
	Bullet_data bullet_templates[static_cast<unsigned
int>(Bullet_type::MAX_TYPES)] = {
	{"Blue",	Bullet_type::Blue,	5, 5},
	{"Red",	 Bullet_type::Red,	7, 2},
	{"Light",	Bullet_type::Light,	10, 20},
	};
};
```

Tiles

"Tiles" class is used to load all tile image in this class, "fileroot" can store the root of the tiles image.

```
const unsigned int tile_num = 25;
class Tiles
{
	GamesEngineeringBase::Image tile[tile_num];
	unsigned int tile_width, tile_height;
	std::string fileroot;
};
```

Trap

"Trap" class is used to store the impassable tiles detail, it's just like a can't move enemy, but the elements are less than the enemy:

```
const unsigned int trap_status_num = 1;
class Trap : public Unit
{
	std::string trap_name;
```

```
	GamesEngineeringBase::Image image[trap_status_num];
	int health;
	unsigned int speed;
	unsigned int attack;
	unsigned int attack_cd = 0;
	Move_Status m_status;
};
```

Map

The “Map” class is used to store the map’s detail, it also needs to store the trap detail when it load the map file, map_width and map_height are the tile’s scale of the map, not the pixel’s amount.

“tiles” point a array which only store the picture index of whole map, It's a one-dimensional array, but when accessing it, I'll use a conversion function to find its corresponding value using two-dimensional coordinates.

“trap” is a array pointer, because only after load the map file, the program can know the amount of the trap and allocate the memory for them.

```
class Map
{
	std::string name;
	unsigned int tiles_width, tiles_height;
	unsigned int map_width, map_height;
	unsigned int trap_num;
	unsigned int* tiles;
	Trap** trap;

public:

		// return the index of the map when using the two-dimensional
positon
	unsigned int& at(unsigned int x, unsigned int y) { return tiles[y *
map_width + x]; }

};
```

To manage these basic classes, I created a separate management class for each to manage their state information during game runtime.

Manager_hero

“Manager_hero” class can control the states of the Hero instance, by update the position or other states of the hero, save and load the hero states from and to the save file, it also store the invincible time, attack and aoe elapse time to control the hero’s attack cooldown time. “is_upgraded” is used to judge if has create a powerup.

```
class Manager_hero
{
	Charactor* Hero;
	Move_Status move_status;
	float invincible_time_elapsed = 0;
	float attack_elapsed;
	float aoe_elapsed;
```

```
	bool is_upgraded = false;
};
```

Camera

The “Camera” class is used to make a virtual camera follow the hero, the camera’s width and height will get form the canvas, it need the hero’s data to update it self’s position, it also need the map’s data the judge if it is touch the map boundary, “Camera” class can also store the map’s pixel scale.

```
class Camera
{
	float locate_x, locate_y;
	unsigned int camera_width, camera_height;
	unsigned int map_width_pix, map_height_pix;
	bool is_at_boundry;
};
```

Manager_map

The “Manager_map” class is used to initialize the map data when player choose the mode of the game, draw the map pixels into the canvas when game running, and save load the map date to the save file.

```
class Manager_map
{
	Map map;
	Tiles tiles;
};
```

Manager_enemy

“Manager_enemy” class is more complicated , it has allocate a enemy array to store the enemy instance, when new enemy has been create, it will be store in the array, there are two more array call the “move_status” and “enemy_attack_time_elapsed”, they have the same capacity with the “enemy” array to store their current states, this makes each enemy has a different attack time counter. “max_size” is to store the max amount of the current enemy, “score” is the current score from the enemy you destroyed, “current_size” is the amount of the current enemy, “enemy_creat_time_elapsed” and “create_threshold” are used to control when to create an enemy.

```
const unsigned int max_enemy_num = 50;
class Manager_enemy
{
	Enemy_index e_index;
	Enemy* enemy[max_enemy_num];
	unsigned int max_size = 20;
	unsigned int score;
	unsigned int current_size = 0;
	Move_Status move_status[max_enemy_num];

	float enemy_create_time_elapsed = 0;
	float create_threshold = 3.f;
```

```
        float enemy_attack_time_elapsed[max_enemy_num];
};
```

Manager_bullet

"Manager_bullet" class is similar to "Manager_enemy", but does not store the attack elapsed, it store the bullet's(projectile) moving direction by the forward array. "Position" is a struct to store the position, but here it can also store the direction.

"aoe_Light_render_start" is used the store the time elapsed when the Lighting has been created, this makes the picture of the Lighting won't disappear immediately.

```
struct Position
{
        float x;
        float y;
};

const unsigned int max_bullet_num = 300;
class Manager_bullet
{
        Bullet* bullet[max_bullet_num];
        Bullet_index b_index;
        Move_Status move_status[max_bullet_num];
        Position forward[max_bullet_num];
        unsigned int current_size = 0;
        float aoe_Light_render_start = 0;

}
```

Each "Manager" class(except the Map) has a "update" function to update their data when the loop running. They also have a "draw" function to draw their elements which are controlled by them. The remaining function is their "get" method, which returns the element value of itself, allowing other classes to view their state and modify their own data.

2.2 technologies implemented

2.2.1 A virtual camera that follows the player-controlled (hero) character at its centre

After create the camera instance, use the `camera_init` function to initialize the camera data, the parameter should have the map's pixel width and height.

```
void Camera::camera_init(GamesEngineeringBase::Window& canvas, unsigned int
map_w, unsigned int map_h)
{
        camera_width = canvas.getWidth();
        camera_height = canvas.getHeight();
        map_width_pix = map_w;
        map_height_pix = map_h;
        locate_x = static_cast<float>(map_width_pix / 2 - camera_width / 2);
        locate_y = static_cast<float>(map_height_pix / 2 - camera_height / 2);
}
```

And in every loop, the "camera" will update itself by the data from the "Manager_hero" class. In the infinite map mode, it has another update function which doesn't has the if condition below, it will only update the position.

```
void Camera::update(Manager_hero& hero)
{
	is_at_boundry = false;
	locate_x = hero.get_x() + hero.get_width() / 2 - camera_width / 2;
	locate_y = hero.get_y() + hero.get_height() / 2 - camera_height / 2;

	// Determine if the camera is touching the map boundary
	if (locate_x <= 0)
	{
		is_at_boundry = true;
		locate_x = max(locate_x, 0);
	}
	if (static_cast<int>(locate_x) + camera_width >= map_width_pix)
	{
		is_at_boundry = true;
		locate_x = min(locate_x, static_cast<float>(map_width_pix -
camera_width));
	}
	if (locate_y <= 0)
	{
		is_at_boundry = true;
		locate_y = max(locate_y, 0);
	}
	if (static_cast<int>(locate_y)+camera_height >= map_height_pix)
	{
		is_at_boundry = true;
		locate_y = min(locate_y, static_cast<float>(map_height_pix -
camera_height));
	}
}
```

It has a function to convert map coordinates to camera position:

```
float mapx_to_camerax(const float map_x) { return map_x - locate_x; }
```

To make the hero stay at the centre of the canvas, “Charactor” class has two types of draw function.Here is the one draw function to draw the hero when the camera touch the boundary, use the “`mapx_to_camerax` function give the relative position to the draw function.

```
void Charactor::draw(GamesEngineeringBase::Window& canvas, unsigned int x,
unsigned int y)
{
	for (unsigned int w = 0; w < image[static_cast<unsigned
int>(this->m_status)].width; w++)
	{
		//Make sure the image's pixel in the cam
		if (w + x > 0 && w + x < canvas.getWidth())
			for (unsigned int h = 0; h < image[static_cast<unsigned
int>(this->m_status)].height; h++)
			{
				if (h + y > 0 && h + y < canvas.getHeight())
					//Make sure image has pixel
					if (image[static_cast<unsigned
int>(this->m_status)].alphaAtUnchecked(w, h) > 0)
						//Draw the image pixel with it's x,y
						canvas.draw(x + w, y + h,
image[static_cast<unsigned int>(this->m_status)].atUnchecked(w, h));

			}
	}
```

```
}
```

Then the camera will follow the hero automatically, other manager class's update function will base the camera classes' data to determine if draw their pixels.

2.2.2 A number of NPCs that attack the character

Generated randomly outside of camera view

This method is implemented through this function.This function need the map and camera's pixel size, and can choose create the position near the camera or not.

```
Position Manager_enemy::create_out_camera_pos(Manager_map& map, Camera& cam, bool if_near_cam)
{
	//When I use srand() and rand(),it sometimes happen the continuous number are the same;
	static std::random_device rd;
	static std::mt19937 gen(rd());

	Position pos{ 0, 0 };
	float margin = 20.f;

	if (if_near_cam)
	{
		// create random number from 0 to 3 to choose the direction to create the enemy
		std::uniform_int_distribution<int> side_dist(0, 3);
		// create random number in the camera's scale
		std::uniform_int_distribution<int> offset_x(0, cam.get_cam_width());
		std::uniform_int_distribution<int> offset_y(0, cam.get_cam_height());

		int side = side_dist(gen);
		switch (side)
		{
		case 0: // Up, current x + random number, and y - a margin number to create the position up the camera
			pos.x = cam.get_x() + offset_x(gen);
			pos.y = cam.get_y() - margin;
			break;
		case 1: // Bottom
			pos.x = cam.get_x() + offset_x(gen);
			pos.y = cam.get_y() + cam.get_cam_height() + margin;
			break;
		case 2: // Left
			pos.x = cam.get_x() - margin;
			pos.y = cam.get_y() + offset_y(gen);
			break;
		case 3: // Right
			pos.x = cam.get_x() + cam.get_cam_width() + margin;
			pos.y = cam.get_y() + offset_y(gen);
			break;
		}
	}
	else
	{
		std::uniform_int_distribution<int> distX(0, map.get_map_width_pix());
		std::uniform_int_distribution<int> distY(0, map.get_map_height_pix());
```

```
		// rondom create the position in the map, if it is not in the
camera, then return it
		while (true)
		{
			pos.x = static_cast<float>(distX(gen));
			pos.y = static_cast<float>(distY(gen));

			bool inCameraView = (pos.x > cam.get_x() && pos.x <
cam.get_x() + cam.get_cam_width() &&
				pos.y > cam.get_y() && pos.y < cam.get_y() +
cam.get_cam_height());
			if (!inCameraView)
				break;
		}
	}
	// make sure it is in the map
	pos.x = max(0, min(pos.x, map.get_map_width_pix()));
	pos.y = max(0, min(pos.y, map.get_map_height_pix()));

	return pos;
}
```

Their frequency increases over time and at least 4 different character types that differ in appearance, health and speed

This method is controlled by the “enemy_create_time_elapsed” and “create_threshold” data in the “Manager_enemy” class. Every loop the update function will execute the create function, the “enemy_create_time_elapsed” will plus the time elapsed for each frame, when it bigger than the “create_threshold” , the create function will create a random enemy. When creating an enemy, it will create a random index to choose create which enemy, and create a position outside of the camera, all the detail of enemies are in the enemy_index class has introduced before, so the function can use the index number to create the specific enemy and store it in the enemy pointer array.

```
void Manager_enemy::create_enemy(Manager_map& map, Camera& cam, Manager_hero&
hero)
{
	//When I use srand() and rand(),it sometimes happen the continuous number
are the same;
	static std::random_device rd;
	static std::mt19937 gen(rd());
	// avoid create the upgrade
	static std::uniform_int_distribution<> dist(0,
e_index.get_enemy_index_num() - 2);

	//not touch the max amount
	if (current_size < max_size)
		if (enemy_create_time_elapsed > create_threshold)
		{
			for (unsigned int i = 0; i < max_enemy_num; i++)
			{
				//find a null place
				if (enemy[i] == nullptr)
				{
					//create a position
					Position pos = create_out_camera_pos(map, cam,
true);

					unsigned int index = dist(gen);
					// create enemy
```

```
                                enemy[i] = new Enemy(e_index[index].name,
e_index[index].type, pos.x, pos.y, e_index[index].health,
                                        e_index[index].speed,
e_index[index].attack, e_index[index].attack_cd);
                                //enemy_attack_time_elapsed[i] =
e_index[index].attack_cd;
                                enemy[i]->load_image();
                                current_size++;
                                enemy_create_time_elapsed = 0.f;
                                create_threshold -= 0.2f;
                                create_threshold = max(create_threshold, 0.5f);
                                //std::cout << "creat enemy" << std::endl;
                                break;
                        }
                }
        }
}
```

General NPC behaviour that directs them directly towards the player and one NPC behaviour makes it static but launches projectiles

The enemies(NPC) will update there position at every frame while loop, they will find the shortest way to the hero's position and change their direction in the "update" function.

The method to implement is calculate the distance between hero and enemy. Divide the difference between their x-coordinates and y-coordinates by the distance to obtain the components of their movement in the x and y directions, then multiply by the velocity multiplied by the time component to ensure smoothness of the velocity value. And then update the new position.

```
        float speed;
        for (unsigned int i = 0; i < max_enemy_num; i++)
        {
                if (enemy[i] == nullptr) continue;

                if (enemy[i]->get_health() == 0)
                {
                        delete_enemy(i);
                        continue;
                }
                //The speed follow the time to change
                speed = static_cast<float>(enemy[i]->get_speed()) * time;

                float dx = hero.get_hitbox_x() - enemy[i]->get_hitbox_x();
                //std::cout << hero.get_hitbox_x() << " " <<
enemy[i]->get_hitbox_x() << std::endl;
                float dy = hero.get_hitbox_y() - enemy[i]->get_hitbox_y();
                float len = sqrt(dx * dx + dy * dy);
                if (dx <= 0)
                        move_status[i] = Move_Status::Left;
                else
                        move_status[i] = Move_Status::Right;

                if (len > 0.01f)
                {
                        dx /= len;
                        dy /= len;
                }
```

```
            float new_x = enemy[i]->get_hitbox_x() + dx * speed;
            float new_y = enemy[i]->get_hitbox_y() + dy * speed;

            new_x = max(0, min(new_x, static_cast<float>(map.get_map_width_pix()
- enemy[i]->get_width())));
            new_y = max(0, min(new_y,
static_cast<float>(map.get_map_height_pix() - enemy[i]->get_height())));

            enemy[i]->update(canvas, new_x -
static_cast<float>(enemy[i]->get_width() / 2),
                  new_y - static_cast<float>(enemy[i]->get_height() / 2),
move_status[i]);
      }
};
```

The static enemy's speed is 0,so it won't move at every loop's update, it will launches projectiles automatically, determining by the "enemy_attack_time_elasped".

The method of creating the projectiles is controlled by the "Manager_bullet" class. The "Manager_bullet" instance will update at every frame loop to determine create a bullet or not. The enemy create the "Red" bullet and only the "FlySpookmoth" enemy can create the enemy bullet.

```
            if (enemy[j]->get_type() == Enemy_type::FlySpookmoth)
            {
                  //std::cout << "enemy_attack_time_elapsed" << j << ":" <<
enemy.get_enemy_attack_time_elapsed(j) << std::endl;
                  if (enemy.get_enemy_attack_time_elapsed(j) <
enemy[j]->get_attack_cd())
                        continue;
                  for (unsigned int i = 0; i < max_bullet_num; i++)
                  {
                        if (bullet[i] != nullptr)
                              continue;
                        bullet[i] = new Bullet("Red", Bullet_type::Red,
Unit_Type::Enenmy,
                              enemy[j]->get_hitbox_x(),
enemy[j]->get_hitbox_y(), 4, enemy[j]->get_attack());
                        bullet[i]->load_image();
                        current_size++;
                        break;
                  }

            }
```

In the update function, use if to judge the Enemy_type bullet, and use "move_to_nearest_hero" function to keep move to hero. This bullet will follow the hero's position until it's health drops to 0 and they are deleted.

```
if (bullet[i]->get_from() == Unit_Type::Enenmy)
{
      bullet[i]->sub_health(time);
      move_to_nearest_hero(i, hero, time);
      continue;
}
```

This function use the same method as the enemy moving method.

```
void Manager_bullet::move_to_nearest_hero(unsigned int bullet_index,
Manager_hero& hero, float time)
```

```
{
	float dx = hero.get_hitbox_x() - bullet[bullet_index]->get_hitbox_x();
	float dy = hero.get_hitbox_y() - bullet[bullet_index]->get_hitbox_y();
	float dist = sqrt(dx * dx + dy * dy);

	if (dist > hero.get_hitbox())
	{

		float speed =
max(static_cast<float>(bullet[bullet_index]->get_speed()) * time, 1.0f);

		if (dist > 0.01f)
		{
			dx /= dist;
			dy /= dist;
		}

		bullet[bullet_index]->update(bullet[bullet_index]->get_hitbox_x() +
dx * speed - static_cast<float>(bullet[bullet_index]->get_width() / 2),
			bullet[bullet_index]->get_hitbox_y() + dy * speed -
static_cast<float>(bullet[bullet_index]->get_height() / 2));
	}
	else
	{
		hero.suffer_attack(bullet[bullet_index]->get_attack());
		delete_bullet(bullet_index);
	}
}
```

2.2.3 Collision system

All the collision system method are implemented by calculating the distance between two unit, if their distance is shorter than the sum of their hitbox, the collision will happened. Whether damage has been caused depends on their unit type.

Hero vs NPCs

Each enemy's distance to the hero is calculated in one frame loop; if the distance is less than the sum of the hitboxes of hero and enemy, and also the hero is not at the invincible time, the hero takes damage. The method of calculate the distance all the same.

```
if (dist_enemy_to_hero < mid_dist && dist_enemy_to_hero > 0.1f &&
hero.get_invincible_time_elapsed() > hero.get_invincible_time())
{
			hero.suffer_attack(e_index[enemy[i]->get_type()].attack);
			hero.zero_invincible_time_elapsed();
}
```

And in the hero's "update" function, it will always check hero's health, if it's health is 0, the hero states will change to "Dead", the game will stop.

```
void Manager_hero::update(GamesEngineeringBase::Window& canvas, Manager_map& map,
float time)
{
	if(!Hero->check_if_dead())
	{
		…
	}
	else
```

```
	{
		move_status = Move_Status::Dead;
		Hero->update(move_status);
	}

}
```

Hero vs impassable terrain

Every impassable terrain tile is a unit and has a hitbox.If hero touch its hitbox,the hero will be rebound from the impassable terrain and get hurt(the rebound also happens between the enemies).

The rebound funciton recieve the original position(from), the position of the object(to) you will touch, and the sum of hitboxs. If the distance between the two is too short, the position coordinates will not be modified to avoid freezing. If the distance is appropriate, the “from” element will be moved in the opposite direction proportionally based on the difference between the distance and the hitbox, and its new position will be returned.

```
Position static rebound(Position from, Position to, float hitbox_len)
{
	float dx = from.x - to.x;
	float dy = from.y - to.y;
	float dist = sqrt(dx * dx + dy * dy);
	//avoid it is too close
	if (dist < 1e-4f)
		return from;

	float nx = dx / dist;
	float ny = dy / dist;
	float overlap = (hitbox_len - dist);
	if (overlap <= 0.0f)
		return from;
	// push number is the half of the difference between both
	float push = overlap * 0.7f;
	return { from.x + nx * push , from.y + ny * push };
}
```

Hero projectiles vs NPCs and NPC projectiles vs Hero

Every bullet(projectile) instance has its own type, it will identify who launched them.

In the “Manager_bullet” update function. It will check the bullet’s type and where they from(who created it):

```
void Manager_bullet::update(GamesEngineeringBase::Window& canvas, Manager_map&
map, Manager_hero& hero,
	Manager_enemy& enemy, Camera& cam, float time)
{
	if (enemy.get_enemy_current_num() > 0)
		this->create_hero_bullet(hero, enemy);
	this->create_enemy_bullet(enemy);


	for (unsigned int i = 0; i < max_bullet_num; i++)
	{
		if (bullet[i] != nullptr)
		{
```

```
                    //check if bullet is fly out the map
                    if (bullet[i]->get_hitbox_x() < 0 ||
bullet[i]->get_hitbox_x() > map.get_map_width_pix() ||
                           bullet[i]->get_hitbox_y() < 0 ||
bullet[i]->get_hitbox_y() > map.get_map_height_pix())
                    {
                           delete_bullet(i);
                           continue;
                    }
                    if (bullet[i]->get_health() <= 0)
                    {
                           delete_bullet(i);
                           continue;
                    }
                    // bullet from hero
                    if (bullet[i]->get_from() == Unit_Type::Hero &&
bullet[i]->get_type() == Bullet_type::Blue)
                    {
                           keep_move_to_enemy(i, enemy, time);
                           continue;
                    }
                    // bullet from enemy
                    if (bullet[i]->get_from() == Unit_Type::Enenmy)
                    {
                           bullet[i]->sub_health(time);
                           move_to_nearest_hero(i, hero, time);
                           continue;
                    }
                    // bullet from hero and it is Lighting
                    if (bullet[i]->get_from() == Unit_Type::Hero &&
bullet[i]->get_type() == Bullet_type::Light)
                    {
                           check_delete_Light(i, enemy, time);
                           continue;
                    }
             }
      }
}
```

They will use the same way to calculate the distance between itself and its target.But use the different way to move to their target and make different damage. When it touch its target, the bullet will get its data from the “Bullet_inde” by their own type and call the damage function from the target class, then delete itself.e.g:

```
enemy.suffer_attack(enemy_index, bullet[bullet_index]->get_attack());
delete_bullet(bullet_index);
```

2.2.4 The hero attacks the NPCs with at least two different types of attack

A linear attack that targets closest NPC (has cooldown) – the attack runs automatically all the time

The “Manager_bullet” class’s “update” function will try to create the bullet at every frame loop, it will determined by the “attack_elapsed” in the “Manager_hero” class and the “attack_cd” in the “Character” class.The elapsed time will be update when the loop running:

```
      void update_cd(float time)
      {
            attack_elapsed += time;
```

```
            if (aoe_elapsed <= Hero->get_aoe_cd())
            {
                  aoe_elapsed += time;
            }
            invincible_time_elapsed += time;
      }
```

The hero's bullets' direction will be initialize when it has been created by the "set_forward" function in "Manager_bullet" class, it will calculate each distance between hero and enemy, and store the shortest distance in the "min_dist". After the for loop, calculate the direction and return it, it is stored in "Manager_bullet" class's "forward" array:

```
Position Manager_bullet::set_forward(unsigned int bullet_index, Manager_enemy&
enemy)
{
      float min_dist = 1e9f;
      float mindx = 0;
      float mindy = 0;
      if (enemy[i] != nullptr && enemy[i]->get_type() != Enemy_type::Upgrade){
            if (enemy[i] != nullptr)
            {
                  float dx = enemy.get_hit_box_x(i) -
bullet[bullet_index]->get_hitbox_x();
                  float dy = enemy.get_hit_box_y(i) -
bullet[bullet_index]->get_hitbox_y();
                  float dist = sqrt(dx * dx + dy * dy);
                  if (dist < min_dist)
                  {
                        mindx = dx;
                        mindy = dy;
                        min_dist = dist;
                  }
            }
      }

      if (min_dist > 0.01f)
      {
            mindx /= min_dist;
            mindy /= min_dist;
      }

      return { mindx, mindy };
}
```

The bullet will keep flying until it touch the enemy or the map boundary, it will calculate the distance to each enemy and check if they have touch each other.

A special area of effect (AOE) attack that targets the top N max health NPC – triggerred by hero via separate button.

To implement this method, I have to write a stack container to store the N max health enemy, because I think it is the best way to store the unordered indexes with unknow amount. Furthermore, when searching for the next enemy with the highest health, previously identified enemies must be excluded. I only implemented the basic pop, push, find and get head data functions.

```cpp
template<typename T>
class My_Stack {
private:
	struct Node {
		T data;
		Node* next;
		Node(const T& value) : data(value), next(nullptr){}
	};

	Node* head;
	unsigned int  stack_size;

public:
	My_Stack() :
		head(nullptr),
		stack_size(0) {}

	bool empty() const { return stack_size == 0; }
	unsigned int get_size() const { return stack_size; }

	T get_head() { return head->data; }

	bool find(T value)
	{
		Node* curr = nullptr;
		if (head == nullptr)
			return false;

		curr = head;

		while(curr !=nullptr)
		{
			if (curr->data == value)
				return true;
			else
				curr = curr->next;
		}
	}

	void push(const T& value) {
		Node* new_node = new Node(value);
		if (empty())
			head = new_node;
		else {
			new_node->next = head;
			head = new_node;
		}
		stack_size++;
	}

	void pop() {

		Node* temp = head;
		head = head->next;
		delete temp;
		stack_size--;
	}

	void clear() {
		while (!empty()) {
			pop();
		}
	}
```

```
        ~My_Stack() {
                clear();
        }
};
```

And then use for loop to find the enemy which is in the AOE range and has the biggest health, then push it into the stack

```
for (unsigned int j = 0; j < max_enemy_num; j++)
{
        if (enemy[j] != nullptr && enemy[j]->get_type() != Enemy_type::Upgrade)
        {
                // calculate the distance
                float dx = enemy.get_hit_box_x(j) - hero.get_hitbox_x();
                float dy = enemy.get_hit_box_y(j) - hero.get_hitbox_y();

                float dist = sqrt(dx * dx + dy * dy);

                if (dist < hero.get_aoe_range() && current_aoe_bullet < aoe_num)
                {
                        float hp = enemy[j]->get_health();
                        if (hp > max_health && !max_health_enemy_index.find(j))
                        {
                                max_health = hp;
                                max_health_enemy = j;
                        }
                }
        }
}
```

```
if (max_health_enemy != 1e9f)
{
        max_health_enemy_index.push(max_health_enemy);
        current_aoe_bullet++;
}
```

After searching the all enemies, we get the N max health enemies store in the stack(it may not equal the maximum amount of AOE attack), then start create the AOE bullet at the enemy's position, get the enemy data from the stack, then pop it.

```
while (max_health_enemy_index.get_size() != 0)
{
        unsigned int target_index = max_health_enemy_index.get_head();
        max_health_enemy_index.pop();

        for (unsigned int i = 0; i < max_bullet_num; i++)
        {
                if (bullet[i] == nullptr)
                {
                        bullet[i] = new Bullet(
                                "Light",
                                Bullet_type::Light,
                                Unit_Type::Hero,
                                enemy[target_index]->get_x(),
                                enemy[target_index]->get_y());

                        bullet[i]->load_image();
                        current_size++;
                        current_aoe_bullet--;
```

```
                        break;
                }
        }
}
```

To prevent the lightning image from disappearing too quickly and becoming almost invisible, I set a timer to make the lightning bullet last for 0.5 seconds before disappearing. Since the lightning bullet doesn't know which enemy it's targeting, it needs to be identified again here.

```
void Manager_bullet::check_delete_Light(unsigned int bullet_index, Manager_enemy&
enemy, float time)
{
        // find the nearest enemy
        ……
        // if enemy has benn destoried
        if (min_dist > hitbox)
        {
                if (aoe_Light_render_start > 0.5f)
                        delete_bullet(bullet_index);
        }
        // damage the enemy
        else
        {
                enemy.suffer_attack(enemy_index,
bullet[bullet_index]->get_attack());
                if (aoe_Light_render_start > 0.5f)
                        delete_bullet(bullet_index);
        }
}
```

A powerup that increases either the speed of the linear attack and number N of NPCs targetted by the AOE

The powerup is just a kind of enemy which has 0 attack, 0 speed, only 1 health, when the hero touch it, it will call the hero's upgrade function to change hero's data to make hero more powerful, and then destroy it self.

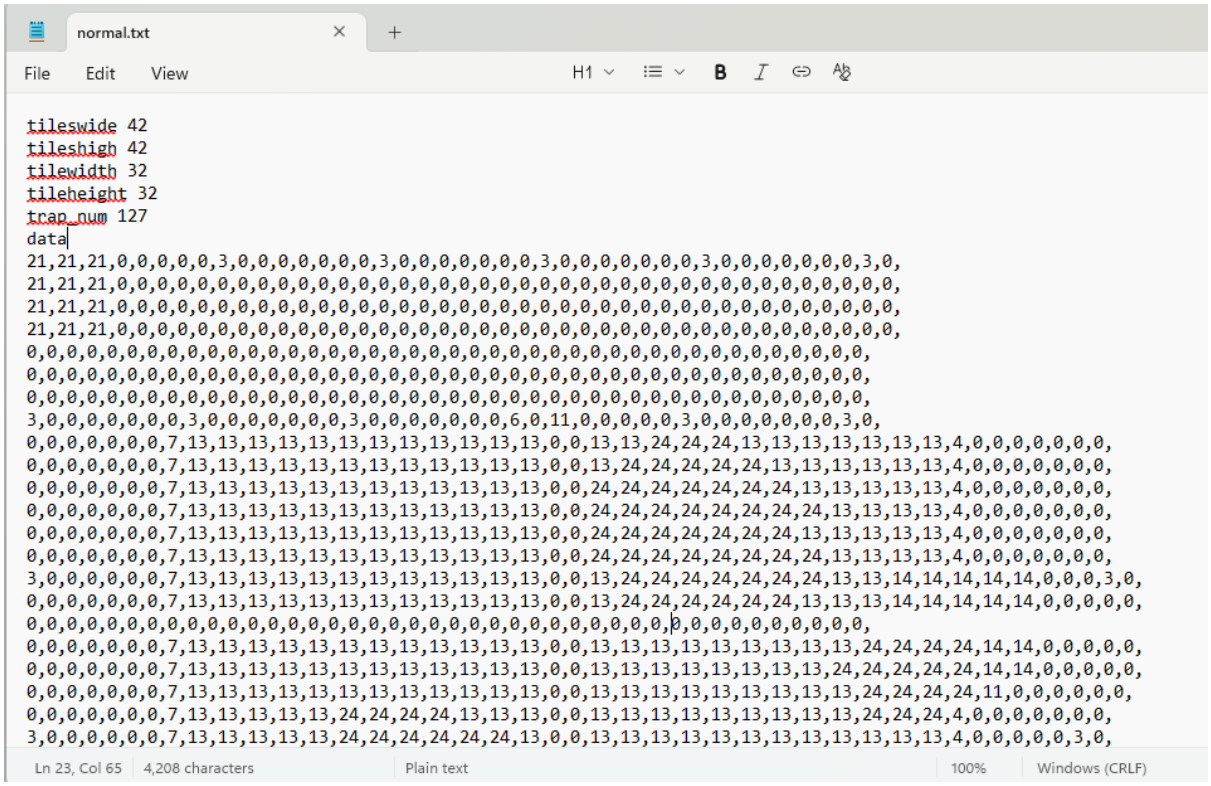
```
if (dist_enemy_to_hero < mid_dist && enemy[i]->get_type() == Enemy_type::Upgrade)
{
        hero.upgrade();
        // kill the upgrade object
        enemy[i]->suffer_attack(1);
        delete_enemy(i);
        //std::cout << "delete upgrade " << std::endl;
        break;
}

void Character::upgrade() {
        attack_cd = 0.2f;
        aoe_num = 10;
        std::cout << "upgrade!!" << std::endl;
}
```

2.2.5 A tile-based method (composed of a number of 32x32 pixel tiles) for displaying the background

At least four different tile types .One of which is impassable for the hero. NPCs can traverse.Data driven level loading,

The map file is a txt file. “tileswide” and “tileshigh” means the number of tiles in width and height(not the pixels), the “Map” class need to know this to create the array to store the map. “tilewidth” and “tileheight” means the tile image’s pixel size. “trap_num” means the amount of impassable tile for the hero, The “Map” class needs to read this from file then allocate the memory for the trap(magma) instance. And then, it start read the file and store it in the array.

The whole map using the same coordinate system, the left top corner is the 0 coordinate. The position of a pixel are calculated by converting its position on the map and the width of the tiles.




```
for (unsigned int h = 0; h < map_height; h++)
{
	std::getline(map_file, line);
	std::stringstream ss(line);
	for (unsigned int w = 0; w < map_width; w++)
	{
		int map_id;
		char comma;
		ss >> map_id;
		ss >> comma;
		at(w, h) = map_id;
		if(map_id == 24)
			for (unsigned int i = 0; i < trap_num; i++)
```

```
                        {
                                if (trap[i] != nullptr)
                                        continue;
                                trap[i] = new Trap("tarp", w * get_tiles_width(), h *
get_tiles_height(),
                                        10, 0, 2, 0);
                                trap[i]->trap_init(get_tiles_width(),
get_tiles_height());
                                //std::cout << "trap" << trap[i]->get_hitbox() << " "
<< trap[i]->get_hitbox_x() << trap[i]->get_hitbox_y() << std::endl;
                                break;
                        }
                std::cout << at(w, h) << " ";
        }
                std::cout << std::endl;
}
```

"at" function is used to convert the two-dimensional position to one-dimensional position.

```
unsigned int& at(unsigned int x, unsigned int y) { return tiles[y * map_width +
x]; }
```

All the tiles image are store in the "Tiles" instance, which is stored in the "Manager_map" class,

```
class Manager_map
{
        Map map;
        Tiles tiles;

}
```

When draw the map, get the map index from the "Map" array, and use "Camera" function to get the map's camera position. At the end, use "Tile" "draw" function to draw the pixel at the canvas.

```
void Tiles::draw(GamesEngineeringBase::Window& canvas, unsigned int id,
        unsigned int map_to_cam_x, unsigned int map_to_cam_y)
{
        for (unsigned int w = 0; w < tile_width; w++)
        {
                if (map_to_cam_x + w > 0 && map_to_cam_x + w < canvas.getWidth())
                        for (unsigned int h = 0; h < tile_height; h++)
                        {
                                if (map_to_cam_y + h > 0 && map_to_cam_y + h <
canvas.getHeight())
                                {
                                        //std::cout << id << std::endl;
                                        canvas.draw(map_to_cam_x + w, map_to_cam_y + h,
tile[id].atUnchecked(w, h));
                                }
                        }
        }
}

// locate_x is camera's x position

const float mapx_to_camerax(const float map_x) { return map_x - locate_x; }

void Manager_map::draw(GamesEngineeringBase::Window& canvas, Manager_hero& hero,
Camera& cam)
```

```
{
	/*cam.update(hero);*/
	for (unsigned int w = 0; w < map.get_map_width(); w++)
	{
		//std::cout << w << " ";
		for (unsigned int h = 0; h < map.get_map_height(); h++)
		{
			tiles.draw(canvas, map.get(w, h),

	static_cast<int>(cam.mapx_to_camerax(static_cast<float>(w *
map.get_tiles_width()))),

	static_cast<int>(cam.mapy_to_cameray(static_cast<float>(h *
map.get_tiles_height())));
	}
}
```

A version of the world which is infinite and a version with a fixed boundary

The infinite method is implemented by using the same map, when the hero cross the map boundary, its position will change to the other side, just like walk into a portal, hero and enemies(even the bullets) will appear from the other side. The camera will follow the hero permanently because it won't touch the boundary anymore.

In this map, the pathfinding methods for all units must be modified. To implement this method, we need to calculate the x direction distance first, judge it if bigger than the half width of the map, if it is, then we should calculate this from the opposite direction.

```
//find the shorest distance
float dx = enemy.get_hit_box_x(j) - hero.get_hitbox_x();
float dy = enemy.get_hit_box_y(j) - hero.get_hitbox_y();

// if the distance bigger the half width of map, then calculate the opposite
direction
if (fabs(dx) > map.get_map_width_pix() / 2)
{
	if (dx > 0) dx -= map.get_map_width_pix();
	else dx += map.get_map_width_pix();
}
if (fabs(dy) > map.get_map_height_pix() / 2)
{
	if (dy > 0) dy -= map.get_map_height_pix();
	else dy += map.get_map_height_pix();
}
```

To draw the infinite map, it have to draw 3x3 times, one is the center map, the rest of 8 times are the surrounding mirror copies. Use the "cam_offset_x" to get the camera offset from the map, because when the hero near the map boundary, the camera's position could be a negative number, and then calculate the coordinates relative to the camera. The "if" in the "Tiles" "draw" function will determine which pixel is in the camera and draw it at the canvas.

```
void Manager_map::draw_infinite(GamesEngineeringBase::Window& canvas,
Manager_hero& hero, Camera& cam)
{
	int map_w_pix = get_map_width_pix();
	int map_h_pix = get_map_height_pix();
```

```
	// Current camera offset relative to the origin
	float cam_offset_x = fmod(cam.get_x(), static_cast<float>(map_w_pix));
	float cam_offset_y = fmod(cam.get_y(), static_cast<float>(map_h_pix));

	// Keep the camera within loop.
	if (cam_offset_x < 0) cam_offset_x += map_w_pix;
	if (cam_offset_y < 0) cam_offset_y += map_h_pix;

	// Draw 3x3 map tiles (the center is the current map, and the surrounding
tiles are mirror copies).
	for (int dx = -1; dx <= 1; ++dx)
	{
		for (int dy = -1; dy <= 1; ++dy)
		{
			int offset_x = dx * map_w_pix -
static_cast<int>(cam_offset_x);
			int offset_y = dy * map_h_pix -
static_cast<int>(cam_offset_y);

			// Draw the current offset map
			for (unsigned int w = 0; w < map.get_map_width(); w++)
			{
				for (unsigned int h = 0; h < map.get_map_height(); h++)
				{
					tiles.draw(canvas, map.get(w, h),
						static_cast<int>(w * map.get_tiles_width()
+ offset_x),
						static_cast<int>(h *
map.get_tiles_height() + offset_y));
				}
			}
		}
	}
}
```

The map with fixed boundary only needs to limited the unit's coordinates, make it in the map when update the position. Fixed boundary map only need draw one time with every pixel

```
new_x = max(0, min(new_x, static_cast<float>(map.get_map_width_pix() -
enemy[i]->get_width())));
new_y = max(0, min(new_y, static_cast<float>(map.get_map_height_pix() -
enemy[i]->get_height())));
```

2.2.6 Game level runs for two minutes

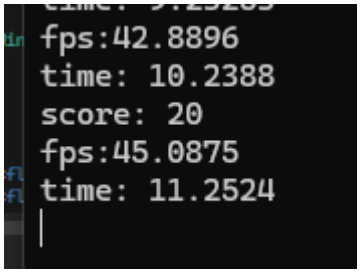
how at least two levels with different maps (one infinite, one fixed)

This implemented by using different map file and use different parameters when initialize the "Manager_enemy" class.

Score and FPS shown at the end (or during)

The FPS and score will be showed at the terminal, when you kill an enemy, the score data will update and store it in the "Manager_enemy" class.

The FPS is using the "GamesEngineeringBase::Timer" to get the time from last loop, then use 1 divided the time to get the FPS and print is.



2.2.7 The ability to save the current state of the game (and save to a file) at any point and reload it at the same exact point

Here is the save file's structure, this file store at same folder with the source code, it named "save.txt"

```
Mode States
1 4.42801
Map States
./Resource/map/normal.txt
Hero States
480 532 21 512 564 50 5 0.5 5 300 5 1 4.42801 0.418954 4.42801 0 0
Enemy States
1 0 1.42634
200.631 862.861 10 216.631 878.861 Bug 1 5 100 1 0 3 0
Bullet States
3 0
262.847 824.407 4 266.847 828.407 Blue 0 0 10 250 5 0 -0.69132 0.722548
350.293 732.691 4 354.293 736.691 Blue 0 0 10 250 5 0 -0.692004 0.721894
436.534 642.545 4 440.534 646.545 Blue 0 0 10 250 5 0 -0.692832 0.721099
```

The file store every data of the game, including every enemy, bullet and hero states.When load the file data, it will clear all states and units at first, and then reinitalize everything to the manager instance, create new enemies and bullet from the save data.

For example:when saving enemy's data, it will write every enemy, every states of enemy to the file to make sure it won't have bug.

```
void Manager_enemy::save_enemy_state(const std::string& filename)
{
	std::ofstream file(filename, std::ios::out | std::ios::app);
	if (!file.is_open()) {
		std::cerr << "[Error] Cannot open save file: " << filename <<
std::endl;
		return;
	}

	file << "Enemy States" << std::endl
		<< current_size << " " << score << " "
		<< enemy_create_time_elapsed << std::endl;

	for (unsigned int i = 0; i < max_enemy_num; i++) {
		if (enemy[i] == nullptr) continue;
```

```
                file
                        << enemy[i]->get_x() << " "
                        << enemy[i]->get_y() << " "
                        << enemy[i]->get_hitbox() << " "
                        << enemy[i]->get_hitbox_x() << " "
                        << enemy[i]->get_hitbox_y() << " "
                        << enemy[i]->get_name() << " "
                        << static_cast<int>(enemy[i]->get_type()) << " "
                        << enemy[i]->get_health() << " "
                        << enemy[i]->get_speed() << " "
                        << enemy[i]->get_attack() << " "
                        << enemy[i]->get_attack_cd() << " "
                        << static_cast<int>(move_status[i]) << " "
                        << enemy_attack_time_elapsed[i] << std::endl;
        }

        file.close();
        std::cout << "[Save] Enemies saved to " << filename << " (" <<
current_size << " units)" << std::endl;
}
```

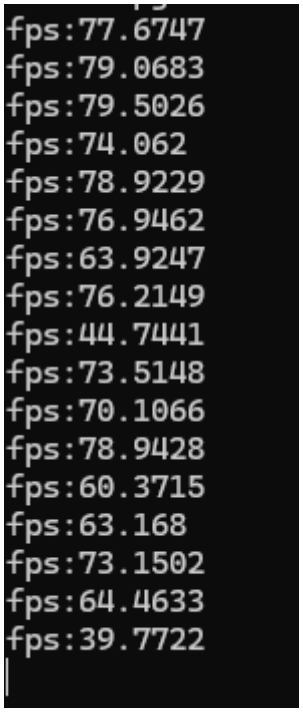
When load the file, it will delete the data before at first, then scan the file to find the key word, this is make sure to find its own data, because the amount of enemies and bullets are not fixed. After that, the function start to set its states and create new enemies, when the next time update function running, everything will be reload to the last saved time.

```
void Manager_enemy::load_enemy_state(const std::string filename)
{
        std::ifstream file(filename, std::ios::in);
        if (!file.is_open()) {
                std::cerr << "[Error] Cannot open load file: " << filename <<
std::endl;
                return;
        }

        for (unsigned int i = 0; i < max_enemy_num; i++)
        {
                if(current_size != 0)
                {
                        if (enemy[i] == nullptr)
                                continue;

                        delete enemy[i];
                        enemy[i] = nullptr;
                        move_status[i] = Move_Status::Front;
                        enemy_attack_time_elapsed[i] = 0;
                        current_size--;
                }
        }

        std::string line;
        while (std::getline(file, line))
        {
                std::cout << line << std::endl;
                if (line.find("Enemy States") != std::string::npos)
                        break;
        }
```

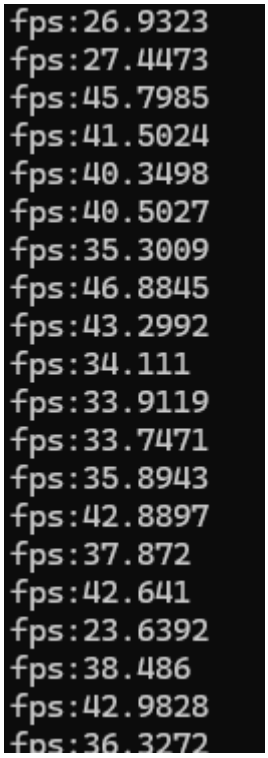
```
        file >> current_size >> score >>create_threshold;

        for (unsigned int i = 0; i < current_size; i++)
        {
                float x, y, hb, hb_x, hb_y;
                int health;
                unsigned int attack, attack_cd, speed, enemy_type;
                std::string enemy_name;
                int m_status;

                file >> x >> y >> hb >> hb_x >> hb_y
                        >> enemy_name >> enemy_type >> health
                        >> speed >> attack >> attack_cd
                        >> m_status >> enemy_attack_time_elapsed[i];

                move_status[i] = static_cast<Move_Status>(m_status);
                enemy[i] = new Enemy(enemy_name,
static_cast<Enemy_type>(enemy_type), x, y, health, speed, attack, attack_cd);
                enemy[i]->load_image();
        }


        file.close();


        std::cout << "[Load] Enemy state loaded from " << filename << std::endl;

}
```

3. FPS Evaluation

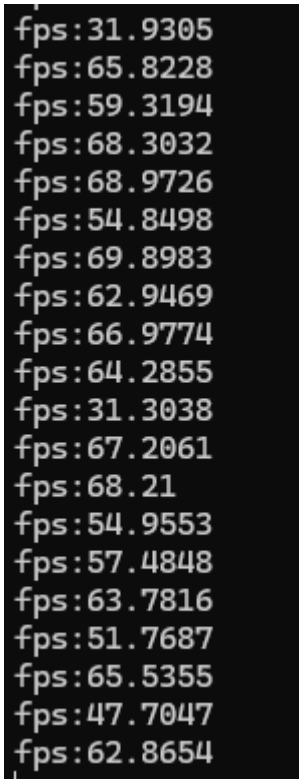
When the game runs in the normal, fixed boundary mode, the average FPS is about 65.



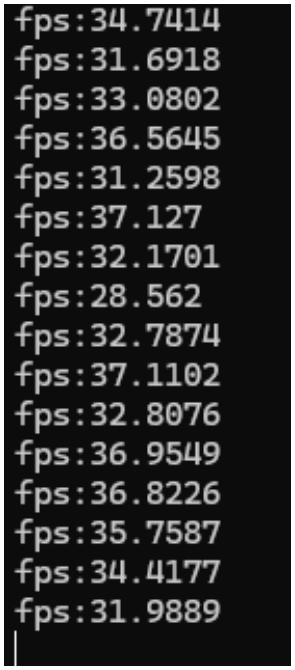
When running the normal infinite mode, the FPS is only about 40.



The Hard fixed boundary mode has lower FPS than the normal mode, only about 60fps.



The Hard infinite mode only has about 30FPS.



The main reason for this phenomenon is that the number of times the “draw” function is called during map rendering significantly impacts the loop execution time. Hard mode has a larger map than normal mode, resulting in a lower frame rate. In infinite mode, the “draw” function is called nine times more often than in fixed mode, thus its frame rate is much lower.

More enemies and more bullets could also be a reason impact the loop time, but its affect is smaller than the draw.

4. Limitation

One bug is that when a hero keeps moving inwards from the corner of a magma tile, there's a chance the hero might momentarily enter the magma tile. The chance of this bug triggering is extremely low. This might be due to a problem with the “rebound” function or the circular hitbox of the magma tile. Due to time constraints, I haven't yet found a better solution or a way to completely resolve this issue.

I've fixed almost all the bugs I found, but I'm not entirely satisfied with the implementation methods for some features. Even if the bugs are fixed, the code might not be concise or perfect enough.

The prohibition on using STL containers has presented some difficulties. Therefore, when implementing certain functions, I've been trying to avoid using these convenient containers, even though I could easily implement them myself. For example, when storing enemy pointers, I had to use a fixed-size array until I needed a container to store the index of enemies with maximum health in my AOE method, at which point I had to write a stack container.

5. Conclusion

This report has covered the game project from five main aspects. It started by demonstrating the gameplay and button controls, then explained the code structure of the game. The report provided detailed explanations of how each feature was implemented, including the methods behind them, and analysed the game's FPS (frames per second) and the factors affecting it. Finally, it listed the current bugs and limitations.

If let me to approach it again

When I first started programming, I always focused on making class designs more elegant and readable. However, as the project progressed, I realized I'd forgotten to add some member variables and methods, making the classes increasingly bloated and the functions more tightly coupled. If I were to rewrite it, I would restructure the classes based on this experience and use better storage methods for variables.

Initially, I wanted to add animations to each unit, but I couldn't think of a way to do it at first. After the initial architecture was complete, I figured out a way, but at the cost of almost completely refactoring the previous code, so I abandoned that idea. If I were to rewrite it, I would try to think about all the features I want to implement and their implementation methods from the beginning, and then build my code structure based on that.

During this assignment, I spent a lot of time gradually optimizing the code I had already written. I believe that with this experience, I won't spend so much time on optimization next time.

Github: https://github.com/Tabris-XiangyiChen/PFA_2D_Game.git