Muneeb Rehman – 40058834

Gurvir Grewal – 40062915

Submitted 22nd December 2020

Game Description

Battle at Dawn is a 2D scroller arcade game with the goal to obtain the highest score possible. The user must operate a drone across a 2D landscape to collect as many coins as possible. Waves of enemies will be attacking your drone throughout the round, hounding the player with a spray of bullets. Worry not, for the player is equipped with their own tools to fight back; they can use their laser to aim and shoot their own return fire, netting them bonus points for defeating enemy drones. Waves of enemies get substantially more dangerous the more drones you destroy.

Summary of Game Aspects

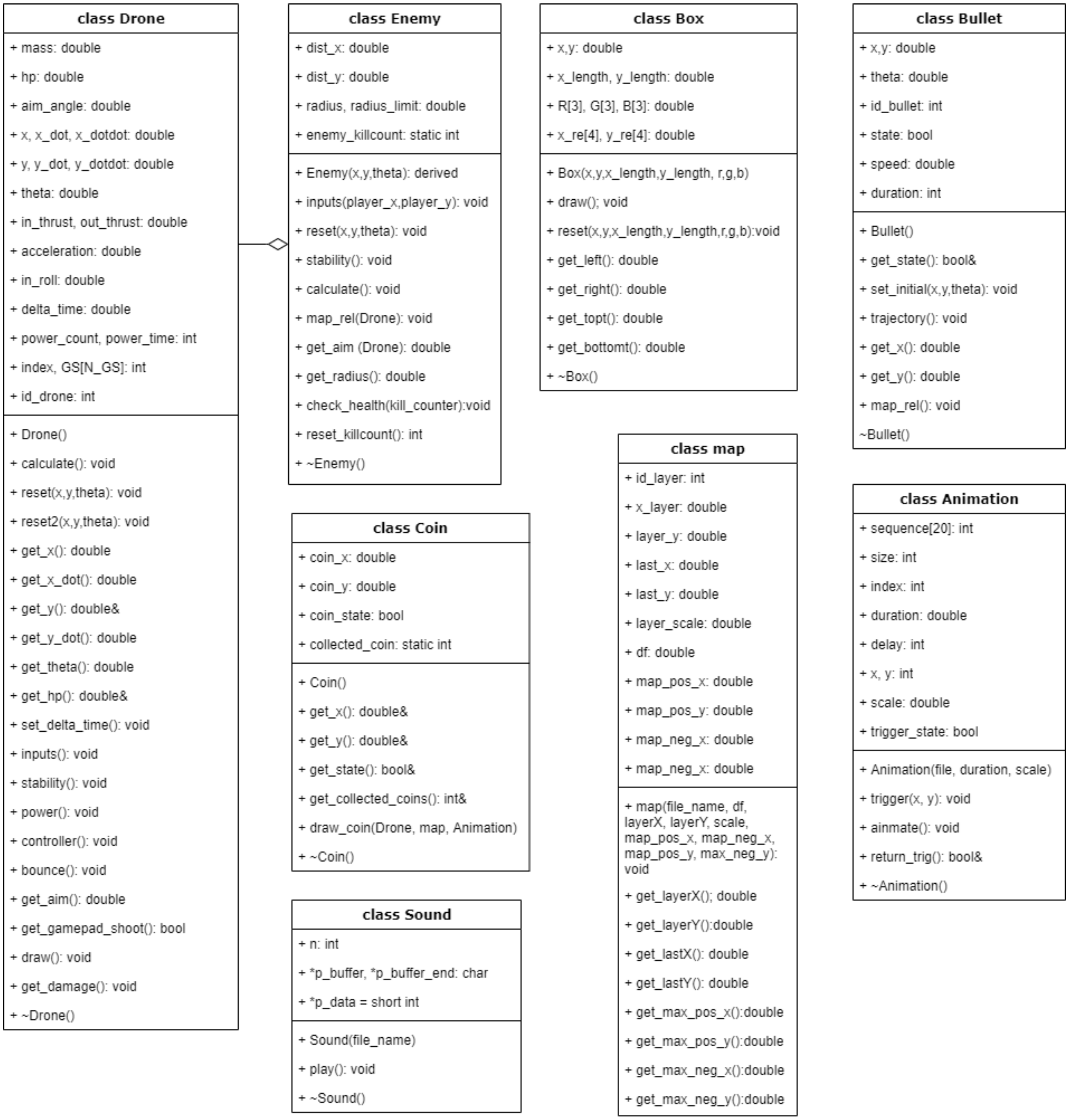
Important: Before starting the game, there is two things you must set. Access the "controller.txt" file and type 'n' or 'Y' (one or the other) to choose between controlling the drone with WASD or a controller. Y means you will have a controller plugged in to operate the drone with the left stick, laser with the right stick, R2 for shooting bullets, and n means you will be using WASD for movement, and 'O' to shoot. Next, if the drone and enemies are flying too quickly, adjust their speed in the "acceleration.txt" file accordingly. Lower the speeds to your liking, if 9.8 is too fast, try 4.2, etc.

When the game begins, your goal will be to fly across the map using the arrow keys, WASD, or a plugged in PS4 controller. Collecting coins is worth 20 points, and killing a drone is worth 100 points. There are six primary functionalities in the game and four secondary functionalities. The primary functionalities are:

1) Controlling the main player drone object to interact with the map and other variables
2) Enemy drone objects stalking the player, periodically shooting at you, and respawning in waves
3) Coin objects to collect for points, and respawning periodically
4) Bullets that the player and enemy drones can shoot, moving relative to the player and the map layers
5) An environment that scrolls using the parallax effect
6) A scoreboard that tracks player high scores and remembers them for future playthroughs

The secondary functionalities are:

1) Collision detection with the environment and bullets
2) Animation sequences for different interactions
3) Sound effects for different interactions
4) Controller compatibility



class Drone

+ mass: double
+ hp: double
+ aim_angle: double
+ x, x_dot, x_dotdot: double
+ y, y_dot, y_dotdot: double
+ theta: double
+ in_thrust, out_thrust: double
+ acceleration: double
+ in_roll: double
+ delta_time: double
+ power_count, power_time: int
+ index, GS[N_GS]: int
+ id_drone: int

+ Drone()
+ calculate(): void
+ reset(x,y,theta): void
+ reset2(x,y,theta): void
+ get_x(): double
+ get_x_dot(): double
+ get_y(): double&
+ get_y_dot(): double
+ get_theta(): double
+ get_hp(): double&
+ set_delta_time(): void
+ inputs(): void
+ stability(): void
+ power(): void
+ controller(): void
+ bounce(): void
+ get_aim(): double
+ get_gamepad_shoot(): bool
+ draw(): void
+ get_damage(): void
+ ~Drone()

class Enemy

+ dist_x: double
+ dist_y: double
+ radius, radius_limit: double
+ enemy_killcount: static int

+ Enemy(x,y,theta): derived
+ inputs(player_x,player_y): void
+ reset(x,y,theta): void
+ stability(): void
+ calculate(): void
+ map_rel(Drone): void
+ get_aim (Drone): double
+ get_radius(): double
+ check_health(kill_counter):void
+ reset_killcount(): int
+ ~Enemy()

class Coin

+ coin_x: double
+ coin_y: double
+ coin_state: bool
+ collected_coin: static int

+ Coin()
+ get_x(): double&
+ get_y(): double&
+ get_state(): bool&
+ get_collected_coins(): int&
+ draw_coin(Drone, map, Animation)
+ ~Coin()

class Sound

+ n: int
+ *p_buffer, *p_buffer_end: char
+ *p_data = short int

+ Sound(file_name)
+ play(): void
+ ~Sound()

class Box

+ x,y: double
+ x_length, y_length: double
+ R[3], G[3], B[3]: double
+ x_re[4], y_re[4]: double

+ Box(x,y,x_length,y_length, r,g,b)
+ draw(); void
+ reset(x,y,x_length,y_length,r,g,b):void
+ get_left(): double
+ get_right(): double
+ get_topt(): double
+ get_bottomt(): double
+ ~Box()

class map

+ id_layer: int
+ x_layer: double
+ layer_y: double
+ last_x: double
+ last_y: double
+ layer_scale: double
+ df: double
+ map_pos_x: double
+ map_pos_y: double
+ map_neg_x: double
+ map_neg_x: double

+ map(file_name, df, layerX, layerY, scale, map_pos_x, map_neg_x, map_pos_y, max_neg_y): void
+ get_layerX(); double
+ get_layerY():double
+ get_lastX(): double
+ get_lastY(): double
+ get_max_pos_x():double
+ get_max_pos_y():double
+ get_max_neg_x():double
+ get_max_neg_y():double

class Bullet

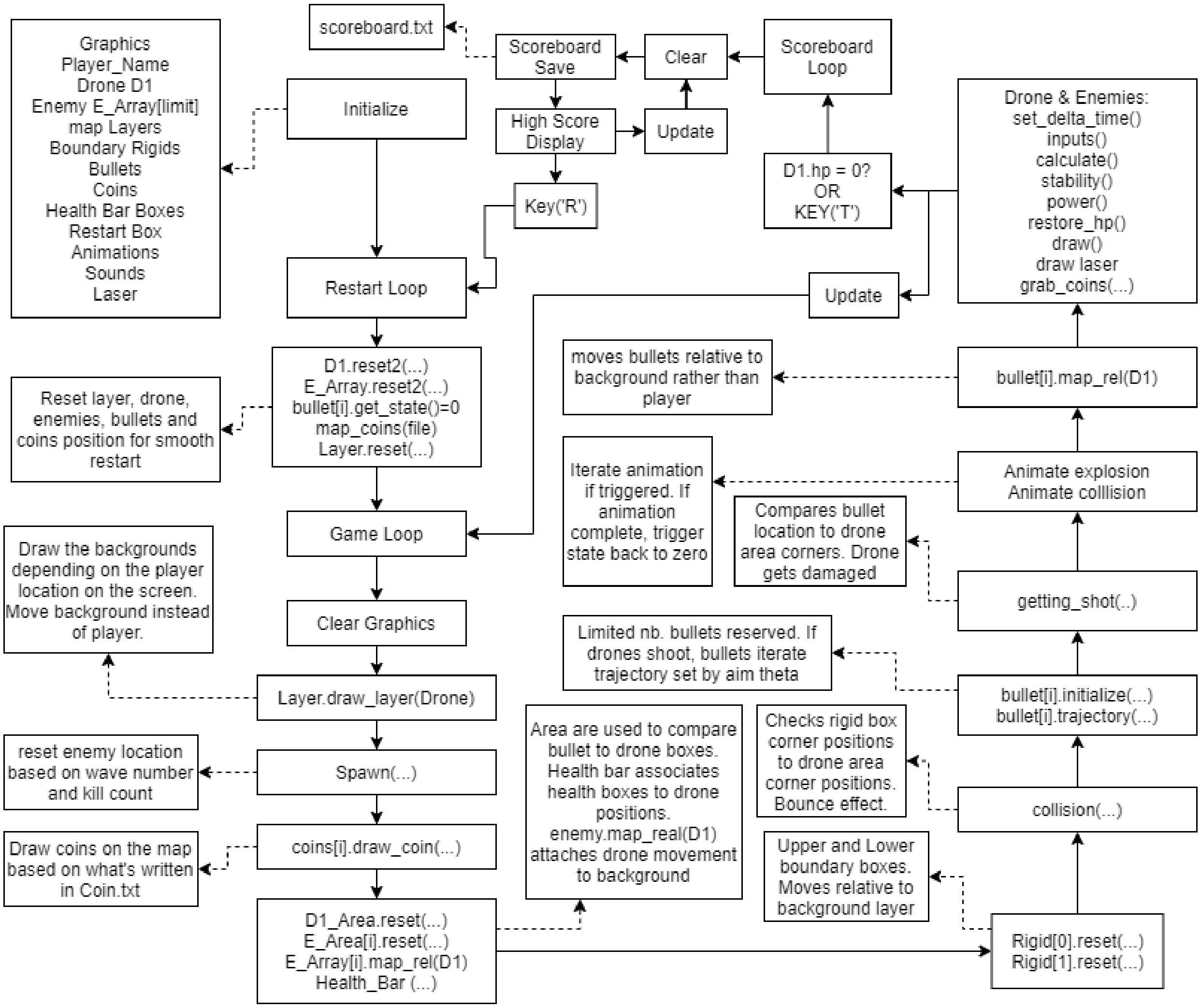
+ x,y: double
+ theta: double
+ id_bullet: int
+ state: bool
+ speed: double
+ duration: int

+ Bullet()
+ get_state(): bool&
+ set_initial(x,y,theta): void
+ trajectory(): void
+ get_x(): double
+ get_y(): double
+ map_rel(): void
~Bullet()

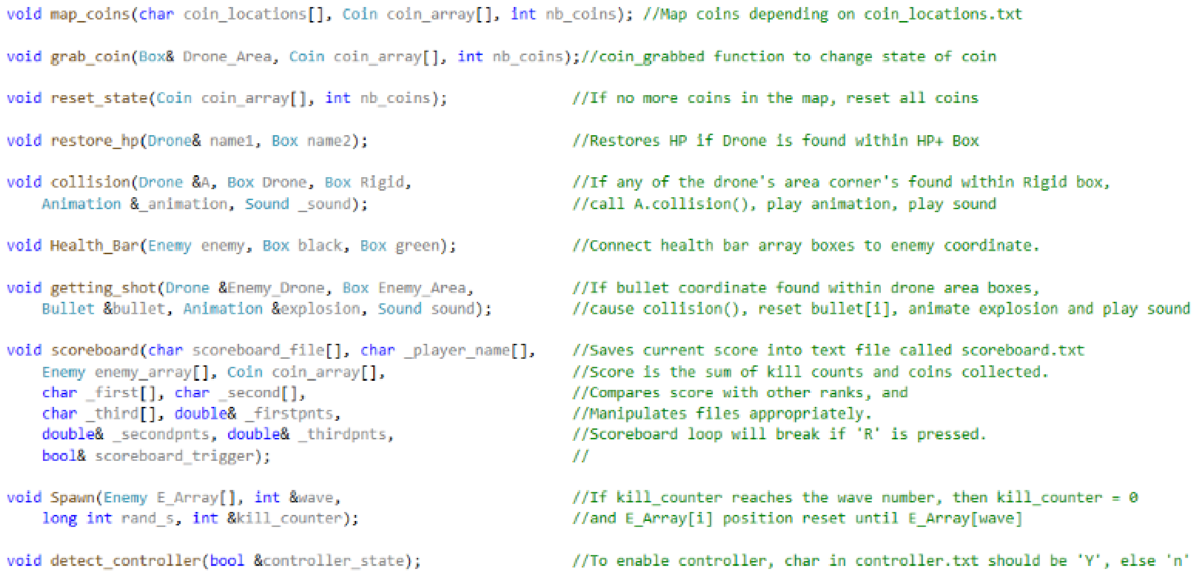
class Animation

+ sequence[20]: int
+ size: int
+ index: int
+ duration: double
+ delay: int
+ x, y: int
+ scale: double
+ trigger_state: bool

+ Animation(file, duration, scale)
+ trigger(x, y): void
+ ainmate(): void
+ return_trig(): bool&
+ ~Animation()


Graphics
Player_Name
Drone D1
Enemy E_Array[limit]
map Layers
Boundary Rigids
Bullets
Coins
Health Bar Boxes
Restart Box
Animations
Sounds
Laser
scoreboard.txt
Initialize
Scoreboard Save
Clear
Scoreboard Loop
High Score Display
Update
Key('R')
D1.hp = 0? OR KEY('T')
Drone & Enemies:
set_delta_time()
inputs()
calculate()
stability()
power()
restore_hp()
draw()
draw laser
grab_coins(...)
Restart Loop
Update
D1.reset2(...)
E_Array.reset2(...)
bullet[i].get_state()=0
map_coins(file)
Layer.reset(...)
Reset layer, drone, enemies, bullets and coins position for smooth restart
moves bullets relative to background rather than player
bullet[i].map_rel(D1)
Iterate animation if triggered. If animation complete, trigger state back to zero
Animate explosion
Animate colllision
Game Loop
Compares bullet location to drone area corners. Drone gets damaged
getting_shot(..)
Draw the backgrounds depending on the player location on the screen. Move background instead of player.
Clear Graphics
Limited nb. bullets reserved. If drones shoot, bullets iterate trajectory set by aim theta
Layer.draw_layer(Drone)
bullet[i].initialize(...)
bullet[i].trajectory(...)
reset enemy location based on wave number and kill count
Spawn(...)
Area are used to compare bullet to drone boxes. Health bar associates health boxes to drone positions. enemy.map_real(D1) attaches drone movement to background
Checks rigid box corner positions to drone area corner positions. Bounce effect.
collision(...)
Draw coins on the map based on what's written in Coin.txt
coins[i].draw_coin(...)
Upper and Lower boundary boxes. Moves relative to background layer
D1_Area.reset(...)
E_Area[i].reset(...)
E_Array[i].map_rel(D1)
Health_Bar (...)
Rigid[0].reset(...)
Rigid[1].reset(...)

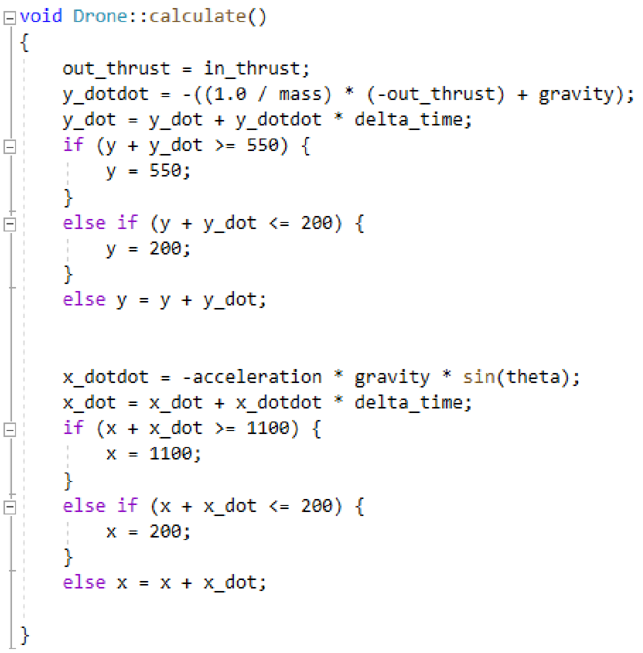
Functions in int main()



```
void map_coins(char coin_locations[], Coin coin_array[], int nb_coins); //Map coins depending on coin_locations.txt

void grab_coin(Box& Drone_Area, Coin coin_array[], int nb_coins);//coin_grabbed function to change state of coin

void reset_state(Coin coin_array[], int nb_coins);              //If no more coins in the map, reset all coins

void restore_hp(Drone& name1, Box name2);                       //Restores HP if Drone is found within HP+ Box

void collision(Drone &A, Box Drone, Box Rigid,                  //If any of the drone's area corner's found within Rigid box,
    Animation &_animation, Sound _sound);                       //call A.collision(), play animation, play sound

void Health_Bar(Enemy enemy, Box black, Box green);             //Connect health bar array boxes to enemy coordinate.

void getting_shot(Drone &Enemy_Drone, Box Enemy_Area,           //If bullet coordinate found within drone area boxes,
    Bullet &bullet, Animation &explosion, Sound sound);         //cause collision(), reset bullet[i], animate explosion and play sound

void scoreboard(char scoreboard_file[], char _player_name[],    //Saves current score into text file called scoreboard.txt
    Enemy enemy_array[], Coin coin_array[],                     //Score is the sum of kill counts and coins collected.
    char _first[], char _second[],                              //Compares score with other ranks, and
    char _third[], double& _firstpnts,                          //Manipulates files appropriately.
    double& _secondpnts, double& _thirdpnts,                    //Scoreboard loop will break if 'R' is pressed.
    bool& scoreboard_trigger);                                  //

void Spawn(Enemy E_Array[], int &wave,                          //If kill_counter reaches the wave number, then kill_counter = 0
    long int rand_s, int &kill_counter);                        //and E_Array[i] position reset until E_Array[wave]

void detect_controller(bool &controller_state);                 //To enable controller, char in controller.txt should be 'Y', else 'n'
```

Modeling Equation:

https://charlestytler.com/quadcopter-equations-motion/



```
void Drone::calculate()
{
    out_thrust = in_thrust;
    y_dotdot = -((1.0 / mass) * (-out_thrust) + gravity);
    y_dot = y_dot + y_dotdot * delta_time;
    if (y + y_dot >= 550) {
        y = 550;
    }
    else if (y + y_dot <= 200) {
        y = 200;
    }
    else y = y + y_dot;


    x_dotdot = -acceleration * gravity * sin(theta);
    x_dot = x_dot + x_dotdot * delta_time;
    if (x + x_dot >= 1100) {
        x = 1100;
    }
    else if (x + x_dot <= 200) {
        x = 200;
    }
    else x = x + x_dot;

}
```

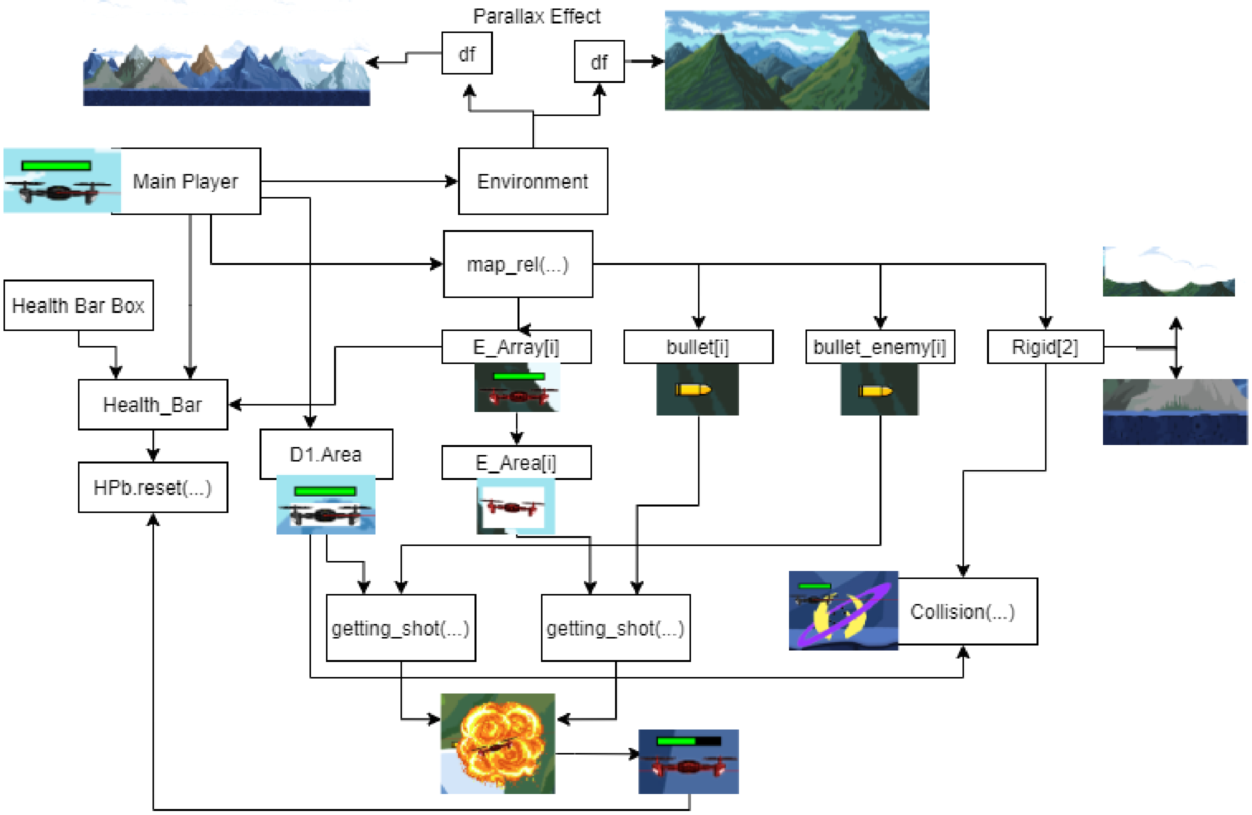
$$\dot{u} = -g\sin(\theta) + rv - qw$$
$$\dot{v} = g\sin(\phi)\cos(\theta) - ru + pw$$
$$\dot{w} = \frac{1}{m}(-F_z) + g\cos(\phi)\cos(\theta) + qu - pv$$

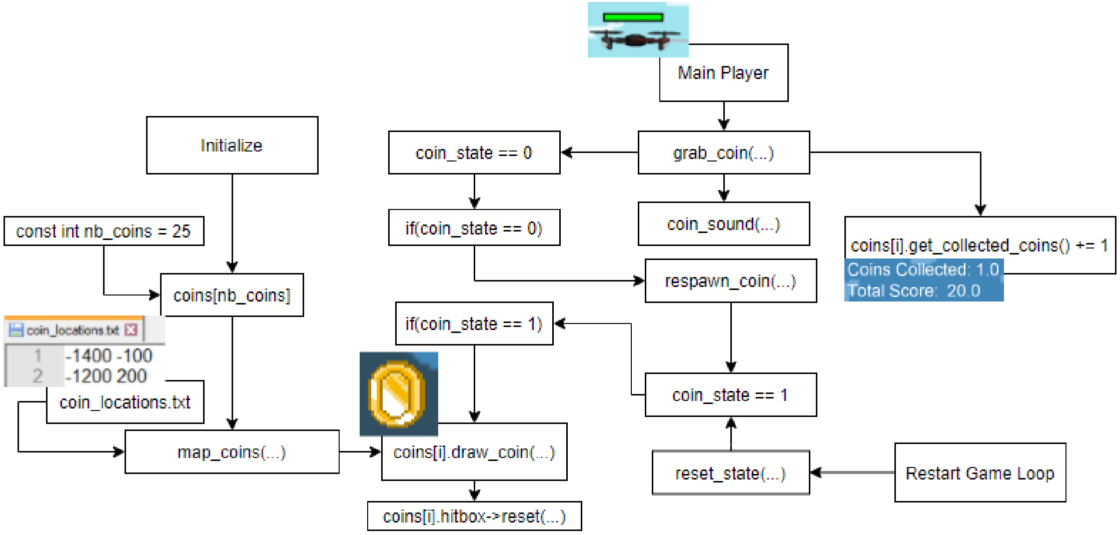
Note:

Some computers may run the simulation slower or faster than others. To fix this issue, change the value in acceleration.txt and try again.

To play with the PS4 controller, type 'Y' under controller.txt, else type 'n'. You also need to install DS4Windows. Left joystick to move, right joystick to aim, R2 to shoot.

For Keyboard only, WASD and 'O' to shoot. 'T' to stop game and 'R' to restart.





Key Explanations of Code

We would have liked to implement networking to the project, but this was a two-man team and the networking material came late in the semester before/during exams and other deadlines. We do not have additional members that we can distribute the work load to make network capabilities a working function of this game, as the two of us worked on this project and all the other functionalities throughout the semester.

The map parallax effect, in the map class, is done by creating "layers" that are background sprites moving at different speeds. Their layer positioning will decide their speed, where the most forefront layer moves the fastest, according to the pixel position of the player's drone.

The animation effects, in the Animation class, are done by creating a sequence of images that are stored in a text file. The sprite image id's are initialized as array elements in a sequence[] array, and when triggered through an interaction in the game, trigger_state variable = 1 and each image is drawn to the screen using the animate() member function. The process involves incrementing through each image id in the array using a delay and duration variable. The duration variable is set to decide how long each image frame will appear, and the game loop increments the delay variable, where each frame is cycled through as the delay increases. Once all images have been displayed, the trigger_state variable is set back to 0, where the animation is no longer appearing until another interaction in the game is met by the player to trigger the next animation.

The scoreboard screen is implemented by creating multiple loops. There is the overall restart loop that resets all the object variables and trigger conditions, the game loop where the game is played, and a scoreboard loop which displays the scoreboard until the user breaks it by pressing 'R' and entering the restart loop. In the scoreboard loop, the scoreboard function is called once; the function uses FILE IO to grab the current list of names and their scores from the high-score list, "scoreboard.txt" file. The data is then organized correctly, and the player's score is compared to these past playthrough high scores. The scoreboard.txt file is then updated, and this information is then displayed to the player using the text(...) 2D graphics library function until the player presses R and breaks the scoreboard loop. The scoreboard contains up to three high scores, and even if these scores are not manually written in the correct order of first to last place in the scoreboard.txt file, the scoreboard(...) function will organize the data from first to third place every time before comparing the player's score.