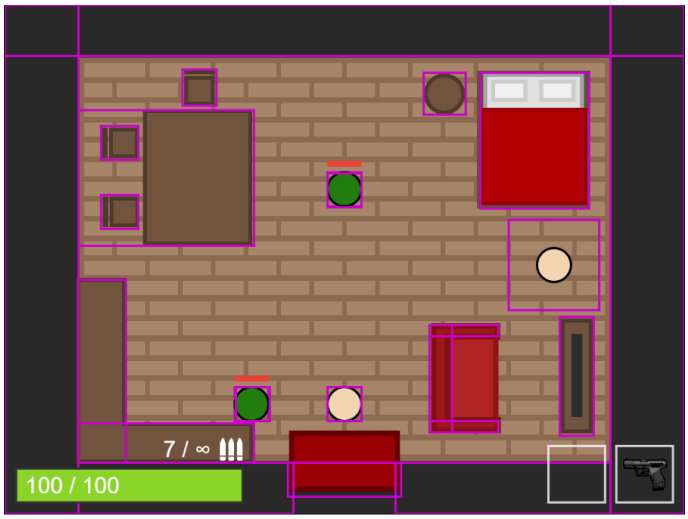
Introduction

I have created a top-down shooter titled ‘Zombie Attack,’ where the player must escape by killing zombies. The player is equipped with a gun, that allows them to shoot zombies in their way. In addition to this, the player can find several collectables as they progress, including keys and medkits. Keys will allow them to unlock doors that are locked, and medkits will restore the player’s health points (HP). In addition to reaching the escape, the player can rescue multiple non-player characters (NPCs), giving the player an additional mission as they complete the game. The purpose of this report is to detail how the previous mechanics work and how they have been implemented in the code, as well as reflect on what existing bugs may be present in the game and areas for improvement.

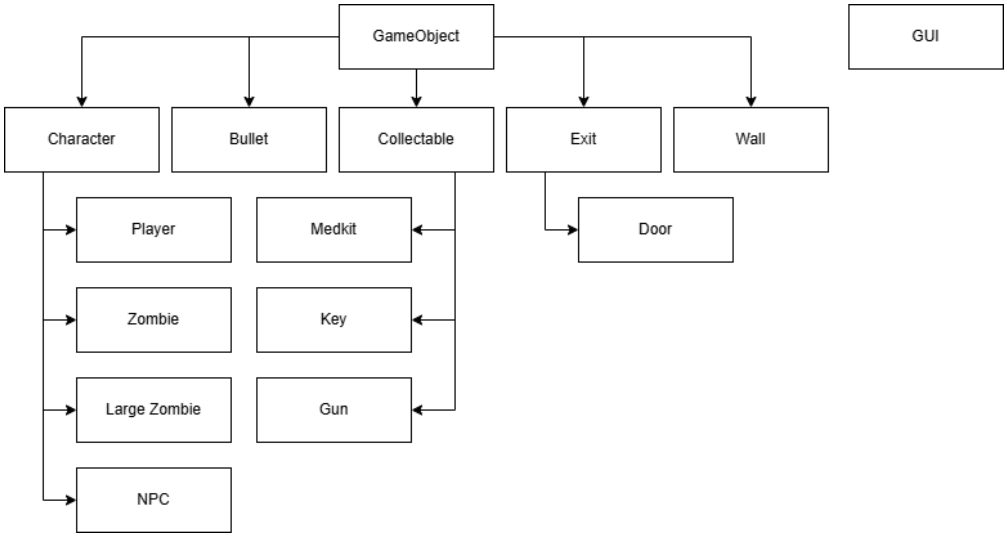
Functionality and Implementation

Collision Detection

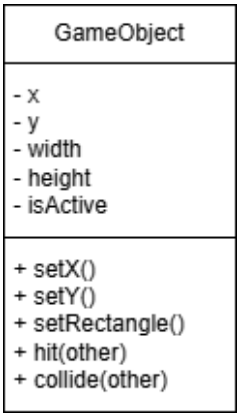
One of the main features that needed to be implemented was collisions. Collisions are extremely important in the game, as they allow the player to collide with obstacles, interact with NPCs, collect items and most crucially, shoot zombies. To achieve this, I used bounding-box collision detection to determine when two objects have collided. In short, this form of collision detection detects when the two boxes overlap with one another by checking each edge of the two boxes. In comparison to grid-based collision detection, I concluded that bounding-box collision detection was more suited for the game I was creating. Grid-based collision detection detects when a moving object hits a stationary object. In my game, it was more important to check for moving objects, such as between the player and a zombie, or a bullet and a zombie. Even though there were a few stationary objects, it would be more practical to handle collisions with these objects through bounding-boxes. As there is a lot of collisions that need to be checked, bounding-box collision would be more efficient in handling this task. Below is a visualisation of all the collision rectangles in a single room.



To implement this form of collision detection, each object within the game would need a bounding box. Because of this, I have created a parent ‘GameObject’ class that each class in the game will inherit from, aside from the GUI class.



As seen in the class diagram, you can see the relationship between each of the classes to the game object class. The game object class has attributes and functions that each class will inherit and utilise to track collisions. It consists of x, y, width and height attributes to draw the collision rectangle. The setX() and setY() functions will set the position of the object, and setRectangle() will set the boundaries of the bounding box. This allows every declared object to have a collision rectangle. The hit(other) function determines whether an object has hit another object. As shown below, the function checks each edge of its own object and the other object, which is being passed through the function, and determines if there is an overlap on any of the rectangle's edges with the other object. If there is, then the function will return true, allowing for use elsewhere in the code.

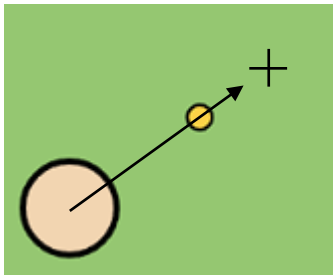



```
hit(other) {
    if (this.#x + this.#width >= other.getX()
        && this.#x <= other.getX() + other.getWidth()
        && this.#y + this.#height >= other.getY()
        && this.#y <= other.getY() + other.getHeight()) {
            return true;
        }
    return false;
}
```

The collide function utilises the hit function to stop two objects from overlapping, such as when the player encounters a wall object, or two zombies that are walking into each other. If two objects are hitting each other, it will reverse the direction of the object so that they are no longer colliding.

```
collide(other) {
    if (other.getX() < this.getX() && this.hit(other)) {
        other.setX(other.getX() - other.getSpeed());
    }
    if (other.getX() > this.getX() + this.getWidth() - 2 && this.hit(other)) {
        other.setX(other.getX() + other.getSpeed());
    }
    if (other.getY() < this.getY() && this.hit(other)) {
        other.setY(other.getY() - other.getSpeed());
    }
    if (other.getY() > this.getY() + this.getHeight() - 2 && this.hit(other)) {
        other.setY(other.getY() + other.getSpeed());
    }
}
```

Gun and Bullet Direction



Shooting a gun is one of the core mechanics of the game and is required to defeat enemies. The user can click at any point on the canvas, and the bullet will shoot from the player's position to the mouse position, provided there isn't any obstacle within the way. As a result of being one of the key features of the game, it took the longest to perfect the functionality as to ensure that the gameplay was not too easy or difficult.

Initial inspiration of this idea came from an online browser game called ZombsRoyale (End Game, 2018), a battle-royale, top-down shooter which uses a similar mechanic to shoot a weapon, albeit with the camera moving along with the player, rather than the camera remaining static on the game that I have created. After seeing this mechanic in another game, I wanted to make use of it in this project as it makes the gameplay engaging and rewarding.

Figuring out how to implement this mechanic took time to figure out. It is easy getting the bullet to travel to the mouse coordinate, but it is more difficult to get it to travel at a constant speed regardless of the distance between the two points. In the first implementation of this code, I had come up with a method that was not ideal. To get the bullet to travel at a constant speed, you must find the ratio of the x and y velocity. This then allows the bullet to travel in the right direction and at the right speed. While the implementation below succeeded in some degree, it failed to deliver consistency across all directions, and was more complicated than it needed to be.

```
this.#xDif = this.#mx - this.getX();
this.#yDif = this.#my - this.getY();

if (this.#xDif >= this.#yDif && this.#xDif > 0) {
    this.#yDif = this.#yDif / this.#xDif * 5;
    this.#xDif = 5;
}
if (this.#yDif >= this.#xDif && this.#yDif > 0) {
    this.#xDif = this.#xDif / this.#yDif * 5;
    this.#yDif = 5;
}
if (this.#xDif <= this.#yDif && this.#xDif < 0) {
    this.#yDif = this.#yDif / this.#xDif * -5;
    this.#xDif = -5;
}
if (this.#yDif <= this.#xDif && this.#yDif < 0) {
    this.#xDif = this.#xDif / this.#yDif * -5;
    this.#yDif = -5;
}
```

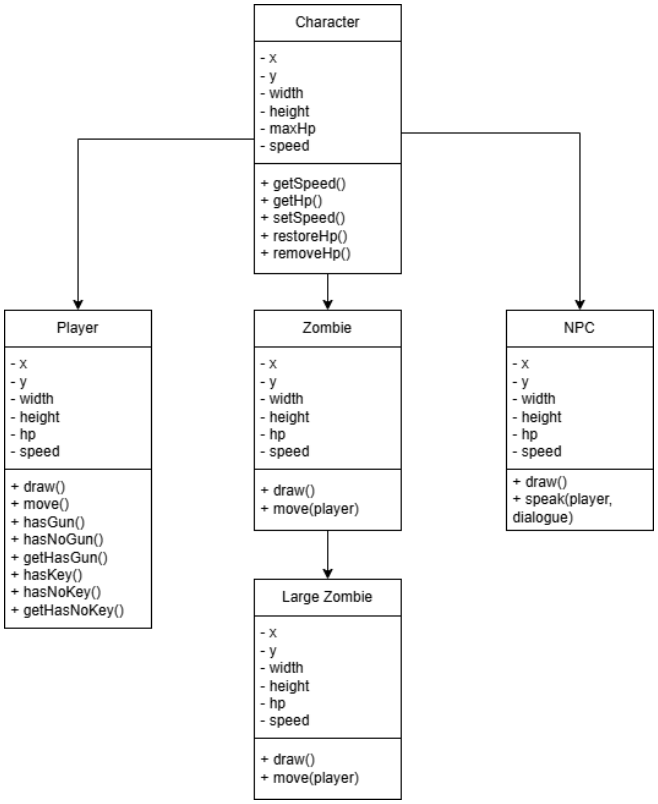
Upon further investigation and research of the maths involved and the JavaScript functions, I came up with an updated method. My initial implementation was missing the crucial step of finding the distance between the two points, and then finding the ratio of the speed, which allows me to update the x and y coordinates of the bullets accordingly. This implementation succeeded in providing a consistent experience when aiming and shooting.

```
this.#xDif = this.#mx - this.getX();
this.#yDif = this.#my - this.getY();

/** * This code uses an example from the MDN web docs *
* Author: Mozilla (author name unknown)
* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt
* Accessed: 15/05/2024 */
this.#dist = sqrt(this.#xDif * this.#xDif + this.#yDif * this.#yDif);

this.#xRatio = this.#xDif / this.#dist;
this.#yRatio = this.#yDif / this.#dist;

this.#xSpeed = this.#xRatio * 5;
this.#ySpeed = this.#yRatio * 5;
```

In addition to aiming, it was important to handle how quickly the bullets should fire. While testing earlier versions of the game, I found that the game ended up being too easy as the player can repeatedly click the mouse, and easily be able to kill enemies. As a result of this, I implemented a cooldown system between shots to prevent this from happening. I had also introduced a reloading system, which means the player must reload once the magazine of the gun had emptied. These additional elements to the gameplay made shooting and killing enemies more rewarding and made the game slightly more challenging.

Characters

Characters are another vital component of the game. There are three different characters present within the game; the player, which the user will control; the zombie, which will follow the player; and the NPC, who the player will be able to interact with and talk to. To implement this within the code, the ‘Player’, ‘Zombie’ and ‘NPC’ classes will inherit from the parent ‘Character’ class which has important attributes and methods that are unique to character objects. These include speed and HP, which will allow characters to move around the canvas and take damage.



Player

The player object controls movement around the canvas, as well as some basic inventory management. With the move() function, the player can move around the canvas with the W, A, S and D keys. It updates the player's x and y coordinates with the player's speed and the corresponding key. The move function also features boundary collision, stopping the player from going out of bounds of the canvas. In addition to this, there are multiple get and set methods in the player class so that the inventory can be managed when the player picks up a gun or a key.

```
move() {
    if(this.getActive()) {
        if (keyIsDown(87)) { // W
            this.setY(this.getY() - this.getSpeed());
        }
        if (keyIsDown(83)) { // S
            this.setY(this.getY() + this.getSpeed());
        }
        if (keyIsDown(65)) { // A
            this.setX(this.getX() - this.getSpeed());
        }
        if (keyIsDown(68)) { // D
            this.setX(this.getX() + this.getSpeed());
        }

        //Boundary Collision
        if (this.getX() < 0) {
            this.setX(this.getX() + this.getSpeed());
        }
        if (this.getX() > width - this.getWidth()) {
            this.setX(this.getX() - this.getSpeed());
        }
        if (this.getY() < 0) {
            this.setY(this.getY() + this.getSpeed());
        }
        if (this.getY() > height - this.getWidth()) {
            this.setY(this.getY() - this.getSpeed());
        }
    }
}
```

Zombie

The zombie will follow the player object on the canvas. If the zombie is hitting the player, the zombie will gradually damage the player. Zombie objects are added to a set, allowing for multiple zombie objects in one area. They are deleted when they are killed or the room changes. There is a for loop that iterates through the set, allowing for the player to deplete health and to ensure collisions for walls and other zombies.

```
move(player) {
    if (this.getX() < player.getX()) {
        this.setX(this.getX() + this.getSpeed());
    }
    if (this.getX() > player.getX()) {
        this.setX(this.getX() - this.getSpeed());
    }
    if (this.getY() < player.getY()) {
        this.setY(this.getY() + this.getSpeed());
    }
    if (this.getY() > player.getY()) {
        this.setY(this.getY() - this.getSpeed());
    }
}
```

```
function manageZombies() {
    for (let zombie of zombies) {
        zombie.draw();
        zombie.move(player);

        if (!zombie.getActive()) {
            killCount += 1;
            zombies.delete(zombie);
        }

        if (player.hit(zombie)) {
            player.removeHp(1);
        }

        for (let wall of walls) {
            wall.collide(zombie);
        }

        for(let zombie2 of zombies) {
            zombie.collide(zombie2);
        }
    }
}
```

NPC

NPCs have a unique function called speak(), which passes through the player object and NPC dialogue. Similar to how collectables work, the player can stand next to the NPC and view their dialogue. When the user presses ‘E’, they can advance the conversation further. Some NPCs will provide the player with collectables when they advance the conversation. The dialogue system is implemented with the speak function checking when the player object is hitting the NPC object, which then displays the first element of the ‘npcDialogue’ array. When the user presses ‘E’, the first element of the array is removed, which then causes the second line of dialogue to be displayed.

```
speak(player, dialogue) {
    if (this.hit(player)) {
        textAlign(CENTER, BOTTOM);
        noStroke();
        textSize(15);
        fill(255);
        text(dialogue[0], this.getX() - 60, this.getY() - 80, 200, 100);
        textAlign(LEFT, BASELINE);
    }
}
```

```
function keyPressed() {
    if (player.hit(npc) && keyCode === 69) {
        npcDialogue.shift();
    }
}
```

Collectables

There are three different collectables present within the game, all with their own unique functions. These consist of the gun, medkit and key. The gun is picked up at the start of the game, allowing them to shoot enemies. The medkit will restore the players health back to the maximum, and the key will enable the player to unlock doors. By pressing ‘E’, the player will be able to interact with the collectable when colliding with its respective collision rectangle.

```
collect(player) {
    if (this.hit(player) && this.getActive()) {
        noStroke();
        textSize(15);
        fill(255);
        text('Take (E)', this.getX() - 3, this.getY() - 5);
        if (keyIsDown(69)) {
            this.deactivate();
            return true;
        }
    }
    return false;
}
```

As seen in the function, text will display when the user stands next to the collectable, indicating what the user should press and what will happen once they press it. This function is then used in the main sketch, where, depending on the item, a certain function will be executed, such as the restoreHp() function when a player takes a medkit, or toggling the ‘hasKey’ or ‘hasGun’ player variables when the player picks up a key or gun.

Exits and Doors

Exits are crucial within the game to ensure that the player can traverse from room to room. Exits tend to be placed at the edges of a room and will spawn the player on the opposite side of the next room, creating a smooth transition between each room. Doors are like exits, but they have the option to be locked or unlocked, and they will teleport the player to a specific position on

the canvas. For locked doors, the player needs to have a key in their inventory to unlock and enter the door. They will be able to see the status of the door with text on the screen, stating whether the door is able to be unlocked or not.

```
function manageExits() {
    for (let exit of exits) {
        if (exit.getIsDoor()) {
            if (exit.open(player)) {
                gameState = exit.getDestination();
                player.setX(exit.getTpX());
                player.setY(exit.getTpY());

                clearSets();
                sceneSetup = false;

                soundUnlock.play();
            }
        }
        else if (player.hit(exit)) {
            gameState = exit.getDestination();

            if (exit.getIsHorizontal()) {
                if (player.getX() < exit.getX()) {
                    player.setX(10);
                }
                if (player.getX() > exit.getX()) {
                    player.setX(width - player.getWidth() - 10);
                }
            }
            else {
                if (player.getY() < exit.getY()) {
                    player.setY(10);
                }
                if (player.getY() > exit.getY()) {
                    player.setY(height - player.getHeight() - 10);
                }
            }

            clearSets();
            sceneSetup = false;
        }
    }
}
```

```
open(player) {
    if (this.hit(player) && this.#isLocked && !player.getHasKey()) {
        noStroke();
        textSize(15);
        fill(255);
        text('Door is locked', this.getX() - 20, this.getY() - 15);
    }
    else if (this.hit(player) && this.#isLocked && player.getHasKey()) {
        noStroke();
        textSize(15);
        fill(255);
        text('Unlock (E)', this.getX() - 8, this.getY() - 15);
        if (keyIsDown(69)) {
            player.hasNoKey();
            this.unlockDoor();
        }
    }
    else if (this.hit(player) && !this.#isLocked) {
        return true;
    }

    return false;
}
```

Game Management

To bring all the elements of the game together, the game needs to be managed to deliver a cohesive gameplay experience to the player. There are multiple sets that are consistently updated in each room, which stores different objects. These include the zombie, bullet, wall and exit sets. As there will be multiple of these objects in the game, it would be easier to access these objects using a set. Compared to using arrays, I found that sets were more suitable for the task as they are more efficient for iterating through multiple times.

The 'gameState' variable would manage what is currently displayed on the canvas, whether it being a specific screen or room. Using a finite state machine, the game will update the sets depending on the current room to deliver a unique number of objects in each room, such as the number of walls, exits and doors, the number of characters, and the collectables that are present within the room.

Reflection

For the most part, the game works very well, and delivers the experience that I wanted it to. There are a few small bugs that are present within the game, but they are extremely minor and don't interfere with the gameplay. For example, the current collision system doesn't work fully as intended. Sometimes, the player can get stuck on certain edges of an object, as well as being

accelerated in a specific direction at the corner of a wall. Another bug is that the player can re-enter the room of a survivor they have already saved, and this will add to the count of the number of survivors saved. Lastly, there is a bug in which the player can be teleported to the wrong location if they get stuck between a room's wall and exit. These bugs are difficult to fix with how the code is currently designed and would require alterations to how certain mechanic's work.

For future development, I would have liked to implement more mechanics into the game. A melee system, in addition to the current gun system, would provide another strategic element when in combat with a zombie. I would also have liked to implement more improvements to enemies. Had I known how, I would have improved the pathfinding of enemies so that they would be able to navigate past walls more easily. A greater variety in enemies would also be a good improvement, such as a zombie that shoots at the player, or a final boss that the player must defeat to reach the end. Lastly, I would have liked to add more rooms and areas for the player to explore to make the game feel more comprehensive and complex.

Bibliography

End Game (2018) *ZombsRoyale* [Game]. Available from: https://zombsroyale.io/ [Accessed at 17 May 2024]