Pouria Salehi katozi s192865

Worms

INFO0004-2 Object-Oriented Programming Projects

Université de Liège
Faculté des Sciences Appliquées
Année académique 2022-2023

1 Introduction

This report provides an in-depth exploration of the Worms project developed in C++, shedding light on its architecture, component interactions, communication mechanisms, and collaborative enhancements. It showcases the synergy of diverse components that contribute to the creation of a highly functional and engaging software product.

2 Software architecture

The main function initializes the Game object and sets up the game loop. It's important to note that the main function also handles user input and communicates with the Game object to control the flow of the game. The Game class acts as the controller in the MVC pattern. It manages the game flow, player turns, and interactions. Additionally, the Game class communicates with the View class to render game elements on the screen.

The GameObject class and its subclasses (Worms, Obstacle, Weapon, Projectile, Powerbar) represent the model in the MVC pattern. These classes encapsulate the game's data and logic. The Worms class handles player movement, actions, and interactions.

The Worms class represents player characters in the game ; it handles player movement, actions, and interactions with the game world, player's health, and manages the state of the player (e.g., moving left, falling, aiming).

The Weapon class, along with its two subclasses Bazooka and Shotgun, is responsible for representing different types of weapons that players can use. Bazooka and Shotgun inherit from the Weapon base class, which provides common attributes and methods. All methods concerning firing, reloading, and managing ammunition are implemented in the Weapon class.

The Projectile class is responsible for representing projectiles fired by weapons, handling the movement, collision detection, and rendering of projectiles. This class stores information about the projectile's type, angle, speed, and position.

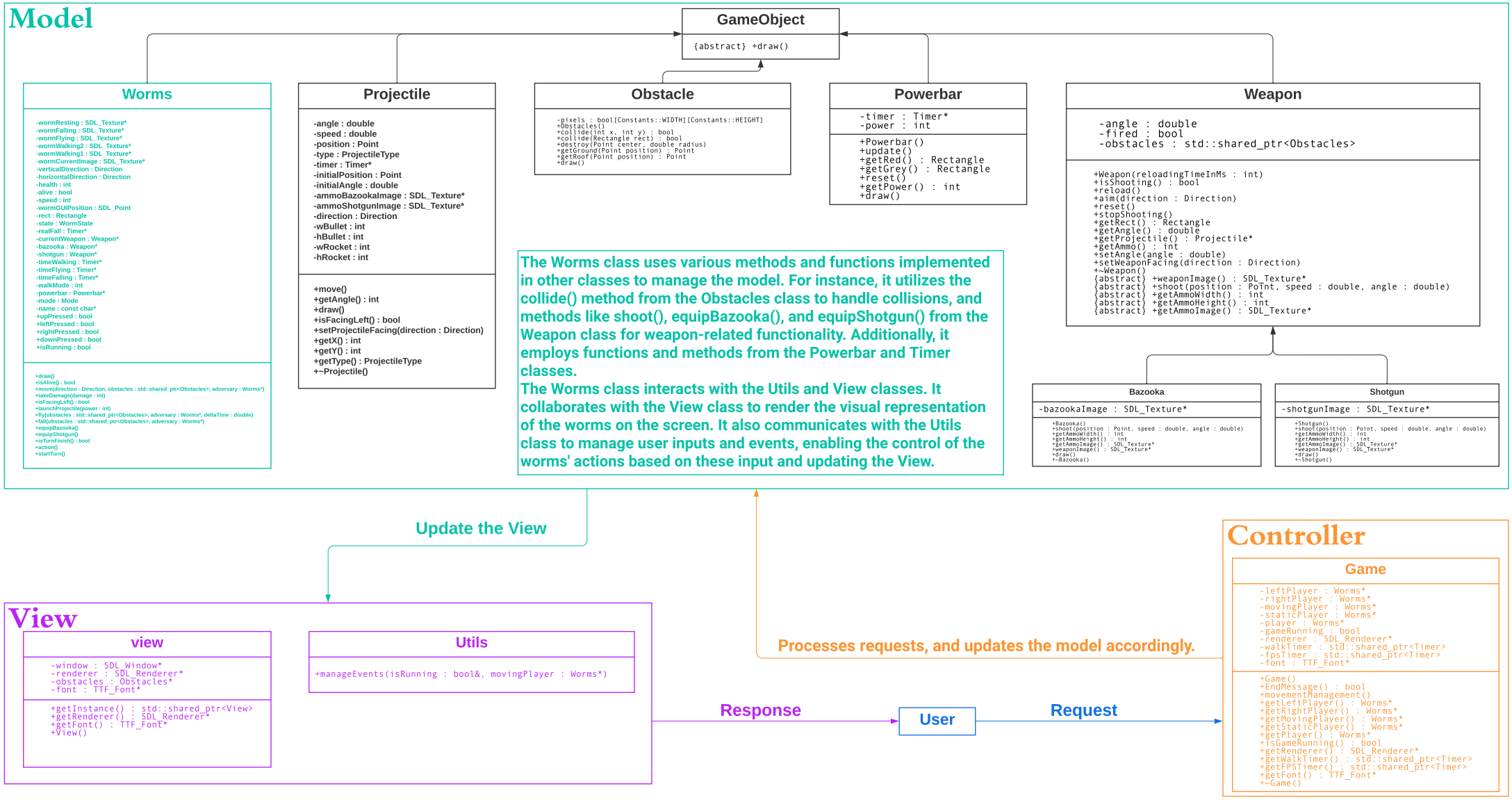
The Utils and View classes are part of the view in the MVC pattern. The Utils class mainly handles user input and event management, while the View class is responsible for graphical rendering. The Time and Shapes classes provide different functionalities used in various parts of the project. For example, the Time class is used for managing timers and animations, while the Shapes class is used for collision detection between game objects. There is a Constants class which provides all necessary constant variables.

The Constants class gathers vital game parameters into a central repository, enhancing code readability and maintenance.

The Timer class offers a flexible time management mechanism, enabling precise animation, events, and turn-based mechanics.

The Shape class provides essential geometric structures like Point, Rectangle, and Circle for efficient collision detection and spatial calculations within the game environment. The Direction enum facilitates standardized orientation control, aiding in managing player and object movements.

In a nutshell, the game's architecture adheres to the Model-View-Controller (MVC) pattern. The Game class serves as the controller, orchestrating gameplay and interactions, while the GameObject and its subclasses (Worms, Obstacle, Weapon, Projectile, Powerbar) constitute the model, encapsulating data and logic. The Utils and View classes manage input and rendering as part of the view. By following this pattern, the game achieves separation of concerns, modularity, and enhanced maintainability.


Model
GameObject
{abstract} +draw()
Worms
-wormResting : SDL_Texture*
-wormFalling : SDL_Texture*
-wormFlying : SDL_Texture*
-wormWalking2 : SDL_Texture*
-wormWalking1 : SDL_Texture*
-wormCurrentImage : SDL_Texture*
-verticalDirection : Direction
-horizontalDirection : Direction
-health : int
-alive : bool
-speed : int
-wormGUIPosition : SDL_Point
-rect : Rectangle
-state : WormState
-realFall : Timer*
-currentWeapon : Weapon*
-bazooka : Weapon*
-shotgun : Weapon*
-timeWalking : Timer*
-timeFlying : Timer*
-timeFalling : Timer*
-walkMode : int
-powerbar : Powerbar*
-mode : Mode
-name : const char*
+upPressed : bool
+leftPressed : bool
+rightPressed : bool
+downPressed : bool
+isRunning : bool
+draw()
+isAlive() : bool
+move(direction : Direction, obstacles : std::shared_ptr<Obstacles>, adversary : Worms*)
+takeDamage(damage : int)
+isFacingLeft() : bool
+launchProjectile(power : int)
+fly(obstacles : std::shared_ptr<Obstacles>, adversary : Worms*, deltaTime : double)
+fall(obstacles : std::shared_ptr<Obstacles>, adversary : Worms*)
+equipBazooka()
+equipShotgun()
+isTurnFinish() : bool
+action()
+startTurn()
Projectile
-angle : double
-speed : double
-position : Point
-type : ProjectileType
-timer : Timer*
-initialPosition : Point
-initialAngle : double
-ammoBazookaImage : SDL_Texture*
-ammoShotgunImage : SDL_Texture*
-direction : Direction
-wBullet : int
-hBullet : int
-wRocket : int
-hRocket : int
+move()
+getAngle() : int
+draw()
+isFacingLeft() : bool
+setProjectileFacing(direction : Direction)
+getX() : int
+getY() : int
+getType() : ProjectileType
+~Projectile()
Obstacle
-pixels : bool[Constants::WIDTH][Constants::HEIGHT]
+Obstacles()
+collide(int x, int y) : bool
+collide(Rectangle rect) : bool
+destroy(Point center, double radius)
+getGround(Point position) : Point
+getRoof(Point position) : Point
+draw()
Powerbar
-timer : Timer*
-power : int
+Powerbar()
+update()
+getRed() : Rectangle
+getGrey() : Rectangle
+reset()
+getPower() : int
+draw()
Weapon
-angle : double
-fired : bool
-obstacles : std::shared_ptr<Obstacles>
+Weapon(reloadingTimeInMs : int)
+isShooting() : bool
+reload()
+aim(direction : Direction)
+reset()
+stopShooting()
+getRect() : Rectangle
+getAngle() : double
+getProjectile() : Projectile*
+getAmmo() : int
+setAngle(angle : double)
+setWeaponFacing(direction : Direction)
+~Weapon()
{abstract} +weaponImage() : SDL_Texture*
{abstract} +shoot(position : Point, speed : double, angle : double)
{abstract} +getAmmoWidth() : int
{abstract} +getAmmoHeight() : int
{abstract} +getAmmoImage() : SDL_Texture*
Bazooka
-bazookaImage : SDL_Texture*
+Bazooka()
+shoot(position : Point, speed : double, angle : double)
+getAmmoWidth() : int
+getAmmoHeight() : int
+getAmmoImage() : SDL_Texture*
+weaponImage() : SDL_Texture*
+draw()
+~Bazooka()
Shotgun
-shotgunImage : SDL_Texture*
+Shotgun()
+shoot(position : Point, speed : double, angle : double)
+getAmmoWidth() : int
+getAmmoHeight() : int
+getAmmoImage() : SDL_Texture*
+weaponImage() : SDL_Texture*
+draw()
+~Shotgun()
The Worms class uses various methods and functions implemented in other classes to manage the model. For instance, it utilizes the collide() method from the Obstacles class to handle collisions, and methods like shoot(), equipBazooka(), and equipShotgun() from the Weapon class for weapon-related functionality. Additionally, it employs functions and methods from the Powerbar and Timer classes.
The Worms class interacts with the Utils and View classes. It collaborates with the View class to render the visual representation of the worms on the screen. It also communicates with the Utils class to manage user inputs and events, enabling the control of the worms' actions based on these input and updating the View.
Update the View
View
view
-window : SDL_Window*
-renderer : SDL_Renderer*
-obstacles : Obstacles*
-font : TTF_Font*
+getInstance() : std::shared_ptr<View>
+getRenderer() : SDL_Renderer*
+getFont() : TTF_Font*
+View()
Utils
+manageEvents(isRunning : bool&, movingPlayer : Worms*)
Response
User
Request
Processes requests, and updates the model accordingly.
Controller
Game
-leftPlayer : Worms*
-rightPlayer : Worms*
-movingPlayer : Worms*
-staticPlayer : Worms*
-player : Worms*
-gameRunning : bool
-renderer : SDL_Renderer*
-walkTimer : std::shared_ptr<Timer>
-fpsTimer : std::shared_ptr<Timer>
-font : TTF_Font*
+Game()
+EndMessage() : bool
+movementManagement()
+getLeftPlayer() : Worms*
+getRightPlayer() : Worms*
+getMovingPlayer() : Worms*
+getStaticPlayer() : Worms*
+getPlayer() : Worms*
+isGameRunning() : bool
+getRenderer() : SDL_Renderer*
+getWalkTimer() : std::shared_ptr<Timer>
+getFPSTimer() : std::shared_ptr<Timer>
+getFont() : TTF_Font*
+~Game()

3 Improvements

To enhance the overall structure and design of the codebase, it is advisable to reorganize the game logic into distinct classes, each responsible for specific tasks. This reorganization, coupled with the utilization of inheritance, aims to achieve increased robustness, efficiency, and adherence to object-oriented programming principles. By encapsulating functionalities within the appropriate classes, the codebase can transition into a more modular form, simplifying maintenance and alignment with established design patterns.

Furthermore, these changes eliminate any remaining bugs and memory leaks that caused issues such as game freezes, worm teleportations, projectile collisions, projectile appearance discrepancies, and hindered smooth program execution. Notably, the main file lacks any game logic functions or methods, while macros are replaced by a dedicated constants class. Within this constants class, all constants are defined as static variables and with the constexpr keyword, contributing to a smoother program execution and accelerated runtime and use less memory.

4 Possible Improvements

There are several potential areas for improvement in the current codebase :

1.Code Refactoring : Consider refactoring certain sections of the code to make it more concise and readable. This can lead to better maintainability and easier debugging.

2.Modularity : Make the code more modular

3. This project is a rework of the previous submission. Due to time constraints, starting from scratch was not feasible. While the majority of the project has been overhauled, there were instances where certain parts, such as angle calculations, required readaptation instead of an entire conceptual overhaul. This was done to address a bug involving the appearance of projectiles. One possible improvement for the future could involve revisiting and revising this specific part.

4. A Better UML representation.

5 Limits

The most significant limitations and constraints of this project were the lack of knowledge in UML design and software patterns. This is primarily because I have not yet completed courses covering these topics (**INFO0027-2 - Programming techniques :** ALGORITHMICS and SOFTWARE PATTERNS).

6 Constrants class

```
#ifndef _CONSTANTS_H
#define _CONSTANTS_H

struct Color
{
    int r, g, b, alpha;
};

const Color BLUE = {0, 0, 255, 255};
const Color BROWN = {128, 96, 20, 255};
const Color RED = {255, 0, 0, 255};
const Color GREY = {211, 211, 211, 255};
const Color BLACK = {0, 0, 0, 255};

class Constants
{
public:
    static constexpr int WIDTH = 1280;
    static constexpr int HEIGHT = 720;
    static constexpr double PI = 3.14;
    static constexpr int INITIAL_PLAYERX1 = 70;
    static constexpr int INITIAL_PLAYERY1 = 0;
    static constexpr int INITIAL_PLAYERX2 = 1100;
    static constexpr int INITIAL_PLAYERY2 = 0;
    static constexpr int HEALTH = 100;
    static constexpr int WORM_WIDTH = 30;
    static constexpr int WORM_HEIGHT = 50;
    static constexpr int WEAPON_WIDTH = 30;
    static constexpr int WEAPON_HEIGHT = 15;
    static constexpr int FPS = 30;
    static constexpr int ROCKET_MASS = 5;
    static constexpr int BULLET_MASS = 2;
    static constexpr int CROSS_SECTIONAL_AREA = 1;
    static constexpr int GRAVITY = 500;
    static constexpr int AIR_DENSITY = 1;
    static constexpr double DRAG_COEFFICIENT = 0.001;
    static constexpr int BAZOOKA_RELOADING_DELAY = 30000;
    static constexpr int BAZOOKA_INITIAL_AMMUNITION = 2;
    static constexpr int BAZOOKA_DAMAGE_POINT = 10;
    static constexpr int BAZOOKA_RADIUS_OF_DAMAGE = 24;
    static constexpr int SHOTGUN_RELOADING_DELAY = 20000;
    static constexpr int SHOTGUN_INITIAL_AMMUNITION = 4;
    static constexpr int SHOTGUN_DAMAGE_POINT = 20;
    static constexpr int ANGULAR_SPEED_WEAPON = 1;
    static constexpr int POWER_BAR_X = 730;
    static constexpr int POWER_BAR_Y = 20;
    static constexpr int POWER_BAR_WIDTH = 400;
    static constexpr int POWER_BAR_HEIGHT = 50;
    static constexpr int POWER_BAR_TICKS_MILLIS = 10;
    static constexpr int POWER_BAR_MAX_POWER = 500;
    static constexpr int POWER_BAR_POWER_STEP = 10;
};

#endif
```