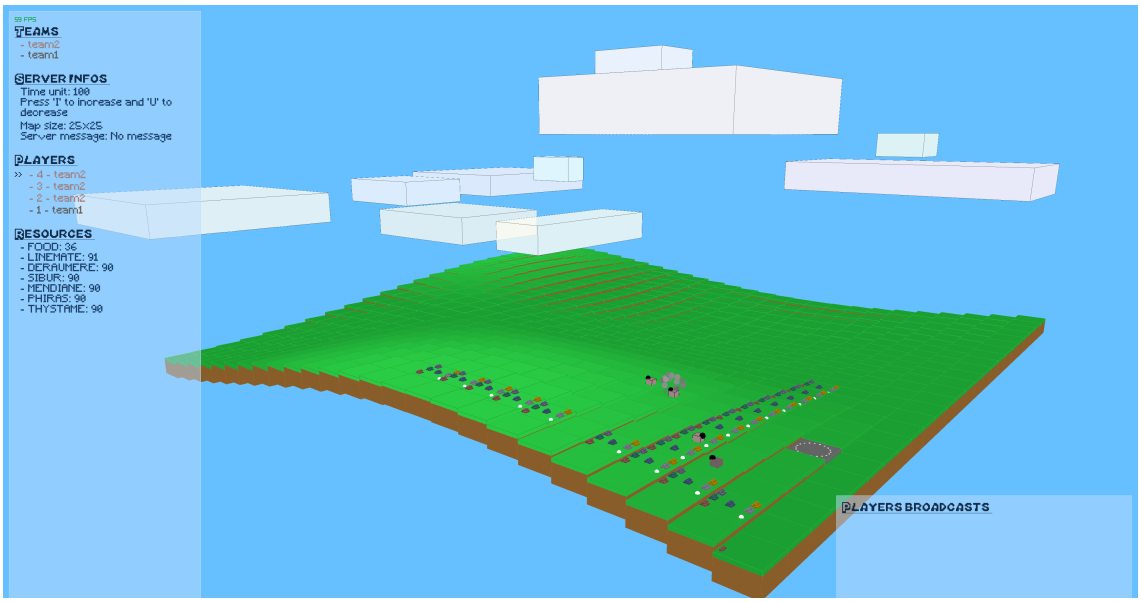


ZAPPY: Graphical Client

Louis Langanay

June 20, 2024

Contents

Chapter 1

Introduction

This document provides comprehensive documentation for the graphical game project using raylib. The project is compiled with a Makefile and provides a binary for usage.

1.1 Binary Usage

```
./zappy_gui -p port -h machine
option description
-p port         port number
-h machine      hostname of the server
```

Chapter 2

API

2.1 Overview

The API is designed to be multithreaded and operates independently of the core logic and graphical display of the game. This design ensures that the API can handle server communications efficiently without interfering with the main operations of the game.

2.2 Threading and Server Communication

The API initializes a separate thread to handle data fetching from the server. This approach allows the core game logic and graphical interface to run smoothly without being blocked by network operations.

2.3 Communication Library

The API utilizes a communication library that was developed in C during the MyTeams project. This library handles the underlying details of establishing connections, sending and receiving data, and managing communication protocols. Integrating this library ensures robust and reliable communication with the server.

2.4 API Initialization

The API is initialized with the server's hostname and port number. It attempts to establish a connection to the server and starts a background thread to continuously fetch data from the server.

```
Api::Api(
    const std::string& host,
    int port
) : _host(host),
    _port(port),
    _isRunning(true),
    _dataMutex()
{
    try {
        _client = protocol_client_create(host.c_str(), port);
        if (!_client || !protocol_client_is_connected(_client))
            throw std::runtime_error("Failed to connect to server");
        _fetchDataThread = std::thread(&Api::fetchDataLoop, this);
        sendCommand("GRAPHIC\n");
    } catch (const std::exception &e) {
        throw ApiException(e.what());
    }
}
```

2.5 Stopping the API

The API can be stopped by setting a flag and joining the background thread. This ensures that all network operations are gracefully terminated.

```
void Api::stop()
{
    _isRunning = false;
    _dataCondVar.notify_all();
    if (_fetchDataThread.joinable())
        _fetchDataThread.join();
}
```

2.6 Sending Commands

Commands can be sent to the server using the ‘sendCommand‘ method. This method ensures that the commands are properly formatted and transmitted to the server.

```
void Api::sendCommand(std::string command)
{
```

```
    if (!protocol_client_send(_client, "%s", command.c_str()))
        throw ApiException("Failed to send command to server");
}
```

2.7 Fetching Data

The API fetches data from the server in a loop running in a separate thread. It processes incoming messages and stores them in a queue, which can be accessed by other parts of the program.

```
void Api::fetchDataLoop()
{
    while (_isRunning && protocol_client_is_connected(_client)) {
        fetchDataFromServer();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void Api::fetchDataFromServer()
{
    protocol_payload_t *payload;

    protocol_client_listen(_client);
    while (!TAILQ_EMPTY(&_client->payloads)) {
        payload = TAILQ_FIRST(&_client->payloads);
        TAILQ_REMOVE(&_client->payloads, payload, entries);
        char *message = payload->message;

        if (message) {
            std::string messageStr(message);
            {
                std::lock_guard<std::mutex> lock(_dataMutex);
                _receivedData.push(messageStr);
                _dataCondVar.notify_one();
            }
        }
        free(payload);
    }
}
```

2.8 API Requests

The API provides various methods to request specific information from the server, such as map size, tile content, team names, player positions, and more. These methods send appropriate commands to the server and process the responses.

```
void Api::requestMapSize()
{
    sendCommand("msz\n");
}

void Api::requestTileContent(int x, int y)
{
    sendCommand("bct " + std::to_string(x) + " " + std::to_string(y) + "\n");
}

void Api::requestTeamsNames()
{
    sendCommand("tna\n");
}
```

2.9 Exception Handling

The API is designed to handle exceptions gracefully. If an error occurs during initialization or command transmission, the API throws an ‘ApiException‘ with a descriptive error message.

```
Api::Api(const std::string& host, int port)
{
    try {
        // Initialization code
    } catch (const std::exception &e) {
        throw ApiException(e.what());
    }
}
```

2.10 Conclusion

The API component of the graphical game project is crucial for handling server communications efficiently. Its multithreaded design ensures that it

operates independently of the core game logic and graphical interface, providing a smooth and responsive user experience.

Chapter 3

Core

3.1 Overview

The Core component serves as the intermediary between the API and the graphical display powered by raylib. It ensures smooth communication and data flow, enabling the game to render updates in real-time based on the data received from the server.

3.2 Integration with API and Raylib

The Core is responsible for receiving data packets from the API and updating the graphical display accordingly. This integration ensures that any changes or updates from the server are reflected in the game interface without delay.

3.3 Packet Handling

The Core includes a dedicated method for handling packets received from the API. This method processes incoming data and updates the game state as necessary. To maintain performance and responsiveness, packet handling is performed in a separate thread. This allows the game to manage packets concurrently with rendering the graphical interface.

3.3.1 Multithreading in Packet Handling

By utilizing multithreading, the Core can process incoming packets from the API without blocking the main rendering loop. This design choice ensures that the game remains responsive and can handle high volumes of data efficiently.

3.3.2 Example Code for Packet Handling

Below is a conceptual example of how packet handling might be implemented in the Core:

```
#include "Core.hpp"
#include "Api.hpp"
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>

Core::Core(Api& api)
    : _api(api), _isRunning(true)
{
    _packetHandlingThread = std::thread(&Core::handlePackets, this);
}

Core::~Core()
{
    stop();
}

void Core::stop()
{
    _isRunning = false;
    _packetCondVar.notify_all();
    if (_packetHandlingThread.joinable())
        _packetHandlingThread.join();
}

void Core::handlePackets()
{
    while (_isRunning) {
        std::string packet = _api.getData();
        if (!packet.empty()) {
            // Process the packet
            processPacket(packet);
        }
    }
}
```

```
void Core::processPacket(const std::string& packet)
{
    // Example packet processing logic
    // This function updates the game state based on the received packet
    if (packet == "some_command") {
        // Handle specific command
    } else {
        // Handle other commands
    }
}

void Core::update()
{
    // Update game state and render using raylib
    while (_isRunning) {
        // Update game state
        // Render using raylib
    }
}
```

3.4 Conclusion

The Core component is essential for linking the API with the graphical display using raylib. Its multithreaded packet handling method ensures that the game can process data from the server efficiently while maintaining a responsive and smooth graphical interface. This design enables the game to handle real-time updates and deliver a seamless user experience.

Chapter 4

HUD

The game being in 3D necessitates a 2D HUD to display more information effectively. To ensure a good display in the game, we have developed an interface `IHud` along with an abstract class `AHud`. This allows us to easily add custom HUD elements.

4.1 Interface `IHud`

The `IHud` interface defines the basic methods required for any type of HUD in the game.

```
namespace Zappy {
    class Map;

    class IHud {
    public:
        virtual ~IHud() = default;
        virtual void draw(Map *map) = 0;

        virtual void setHudWidth(float width) = 0;
        virtual float getHudWidth() = 0;

        virtual void setHudHeight(float height) = 0;
        virtual float getHudHeight() = 0;

        virtual void setHudPos(std::pair<float, float> pos) = 0;
    };
}
```

4.2 Abstract Class `AHud`

The abstract class `AHud` extends `IHud` and provides basic functionality for drawing the HUD.

```
namespace Zappy {
    class AHud : public IHud {
    public:
        virtual ~AHud() = default;
        void draw(Map *map) override = 0;

        void setHudWidth(float width) override;
        float getHudWidth() override;

        void setHudHeight(float height) override;
        float getHudHeight() override;

        void setHudPos(std::pair<float, float> pos) override;

    protected:
        AHud(
            std::pair<float, float> hudRes,
            std::pair<float, float> hudPos
        );

        virtual void drawTextWrapped(
            const std::string &text,
            float x,
            float &y,
            float maxWidth,
            float fontSize,
            const Color &color,
            bool underline
        );

        void drawSectionTitle(const std::string &title, float &y);

        virtual std::string typeToString(Resources::Type type);
        virtual std::string orientationToString(Orientation orientation);

        const float _titleSize = 40;
```

```
        const float _textSize = 35;
        const float _hudPadding = 20;
        float _hudWidth;
        float _hudHeight;
        std::pair<float, float> _hudPos;

        int _playerIndex = 0;
        int _selectedPlayer = -1;

        bool _sectionMargin = false;
    };
}
```

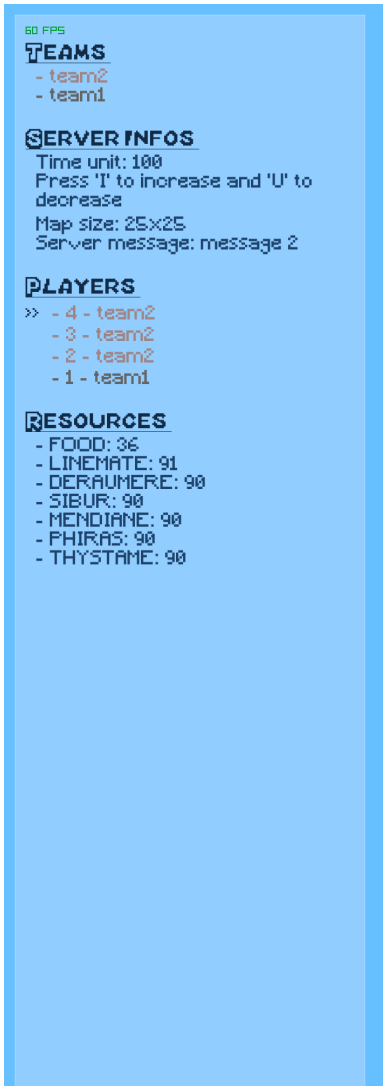
4.2.1 `drawTextWrapped` Method

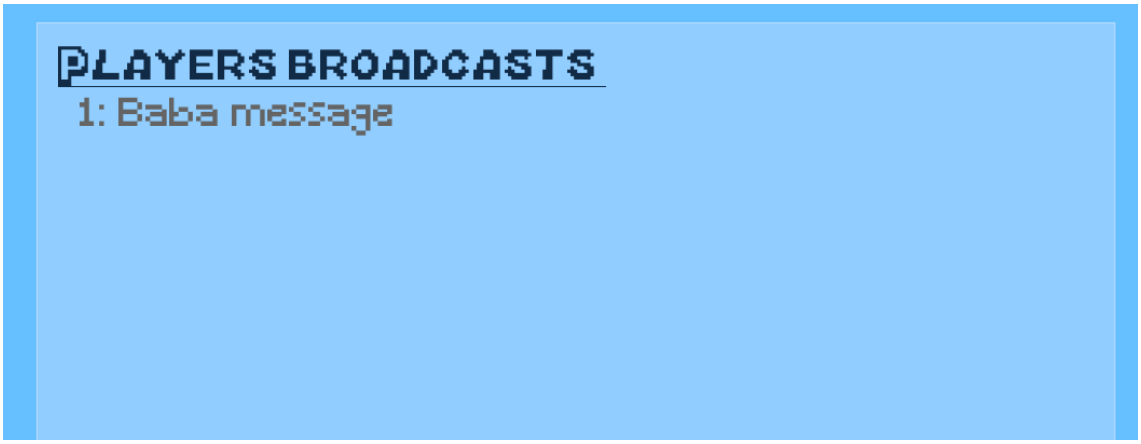
The `drawTextWrapped` method dynamically displays text in the HUD by wrapping it based on the specified maximum width.

```
void AHud::drawTextWrapped(
    const std::string &text,
    float x,
    float &y,
    float maxWidth,
    float fontSize,
    const Color &color,
    bool underline
)
```

4.3 HUD Responsiveness

The HUD can have variable height and width. Therefore, information needs to be displayed dynamically. The `drawTextWrapped` method is used to adjust text based on available space.


60 FPS
TEAMS
- team2
- team1
SERVER INFOS
Time unit: 100
Press 'I' to increase and 'U' to decrease
Map size: 25x25
Server message: message 2
PLAYERS
» - 4 - team2
- 3 - team2
- 2 - team2
- 1 - team1
RESOURCES
- FOOD: 36
- LINEMATE: 91
- DERAUMERE: 90
- SIBUR: 90
- MENDIANE: 90
- PHIRAS: 90
- THYSTAME: 90


PLAYERS BROADCASTS
1: Baba message

4.4 End-of-Game HUD

At the end of the game, a specific HUD is displayed to show the winning team and other relevant game statistics. This HUD provides a summary of the game and the final scores. The following screenshot shows an example of the end-of-game HUD.

4.4.1 Camera Animation

When the game ends, a camera animation is triggered to provide a visual overview of the game map and highlight the winning team. The camera moves through a predefined path, capturing the final positions and states of all players and important game elements. This animation helps to visually summarize the game events and provides a cinematic conclusion to the game.

Chapter 5

Map

The map in the game consists of various elements such as tiles, players, eggs, resources, and teams. Each of these elements plays a crucial role in the game dynamics and visual representation.

5.1 Tile

Each tile in the game represents a specific location on the map with associated resources and possible interactions.

5.1.1 Initialization and Properties

The 'Tile' class is initialized with coordinates (x, y) and a size

- **setResources**: Sets the quantities of different resources on the tile based on a provided vector of integers.
- **addResource** and **removeResource**: Modify the quantity of a specific resource type on the tile.
- **getResources**: Retrieves the current quantities of all resources on the tile.
- **startIncantation** and **endIncantation**: Initiates and concludes an incantation process on the tile, affecting gameplay based on the provided level and player involvement.
- **getIncantationPlayers**: Retrieves the list of player identifiers involved in the current incantation process.

5.1.2 Visualization

The tile is visually represented using 3D rendering techniques. It displays a cube representing the tile itself and may include spheres for food resources and 3D models for other resources.

- **draw**: Renders the tile visually, utilizing the Raylib library's functions to draw cubes and other shapes representing resources.
- The tile's appearance dynamically changes based on the quantities and types of resources present, providing visual feedback to players.

5.2 Player

Each player in the game is represented by an instance of the `Player` class, encapsulating various attributes and behaviors.

5.2.1 Initialization and Attributes

The `Player` class is initialized with the following attributes:

- **Player Number:** Unique identifier for each player.
- **Team:** The team to which the player belongs, encapsulated as a unique pointer to a `Team` object.
- **Orientation:** The direction the player is facing (`Orientation` enum).
- **Level:** Current level of the player.
- **Position and Movement:** Coordinates (`_x`, `_y`) and methods to update position based on movement.
- **Inventory:** A map (`_inventory`) storing quantities of different resource types.
- **State Management:** Flags (`_isMoving`, `_isSelected`) and accumulators for managing player state.

5.2.2 Methods

The class provides methods to interact with and manage player attributes:

- **Accessors:** Methods to retrieve player number, team, orientation, level, and position.
- **Inventory Management:** Methods to add, set, and remove resources from the player's inventory.
- **Broadcasting:** Methods to add and retrieve broadcast messages from the player.
- **Movement:** Methods to update player movement based on target coordinates and handle animation.
- **Drawing:** Utilizes Raylib's drawing functions to visually represent the player in the game world, including a cube for the player's body, a sphere for their head, and optional text for highlighting the selected player.
- **Incantation:** Placeholder methods for initiating and concluding an incantation process.

5.2.3 Visualization

Players are visually represented on the game map with a distinctive cube representing their body and a sphere for their head. The cube's color reflects the player's team, providing visual identification.

- The player's orientation affects the placement of the sphere representing their head.
- Visual effects like text waves are used to denote when a player is selected.

5.3 Egg

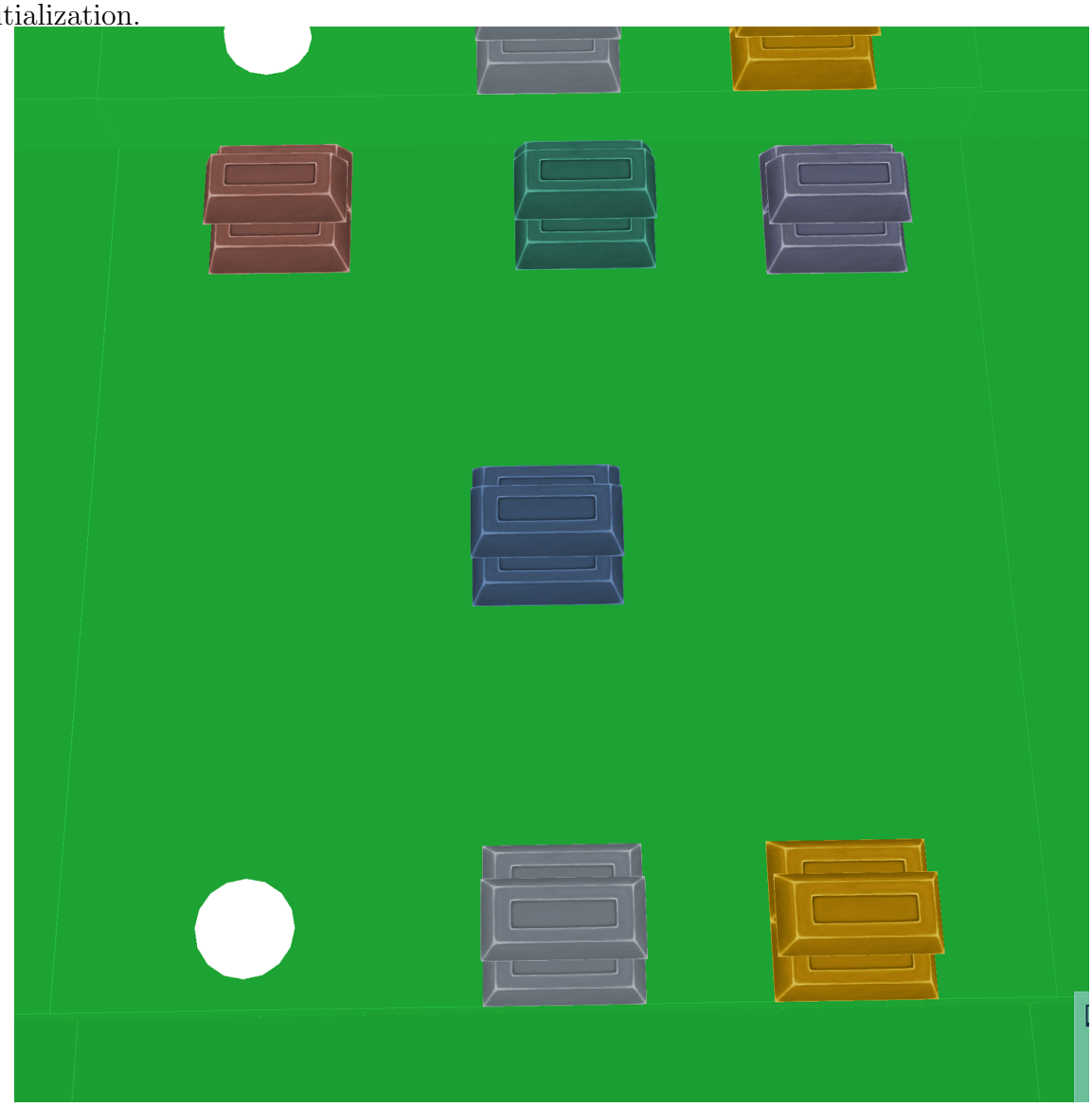
Eggs represent special objects or goals within the game. Players can interact with eggs by collecting or moving them. Each egg is uniquely identified and positioned on the map.

Eggs are instances of the 'Egg' class, added and removed dynamically from the map. They are visualized using 3D models to distinguish them from other elements.

5.4 Resources

Resources are materials or items scattered across the map that players collect to progress in the game. Each resource type has its own visual representation using 3D models.

Resources such as food, linemate, deraumere, sibur, mendiane, phiras, and thystame are represented by models loaded dynamically during map initialization.



5.5 Team

Each team in the game is represented by an instance of the `Team` class, encapsulating essential attributes and methods.

5.5.1 Initialization and Attributes

The `Team` class is initialized with the following attributes:

- **Name:** A string identifier representing the name of the team.
- **Color:** A `Color` object representing the visual identifier of the team.

5.5.2 Methods

The class provides methods to interact with and retrieve team attributes:

- **getName():** Returns the name of the team.
- **getColor():** Returns the color associated with the team.

5.5.3 Visualization

Teams are primarily identified by their assigned color, which is utilized to differentiate team members and their associated entities (such as players and resources) on the game map.

Chapter 6

Model3D

The `Model3D` class encapsulates functionality for loading, manipulating, and drawing 3D models with textures.

6.0.1 Initialization and Constructors

- **Model3D():** Default constructor.
- **Model3D(model_path: string, texture_path: string):** Constructs a `Model3D` object with a specified model and texture path.
- **Model3D(model_path: string, texture_path: string, x: float, y: float, z: float):** Constructs a `Model3D` object with a specified model, texture path, and position.
- **Model3D(model_path: string, texture_path: string, x: float, y: float, z: float, size: float):** Constructs a `Model3D` object with a specified model, texture path, position, and size.

6.0.2 Methods

- **setSize(size: float):** Sets the size of the model.
- **setPosition(x: float, y: float, z: float):** Sets the position of the model.
- **draw():** Draws the 3D model at its current position and size.
- **unload():** Unloads the texture and model resources when they are no longer needed.

6.0.3 Visualization

The `Model3D` class is used to manage and render 3D models within the game environment. It handles loading textures, setting positions, and adjusting sizes, providing a convenient interface for rendering detailed visual elements.

Chapter 7

Particle

7.1 IParticle

The `IParticle` interface defines the basic structure for all types of particles in the game.

7.1.1 Methods

- **virtual void update(float deltaTime)**
  Updates the state of the particle based on the elapsed time since last update.
- **virtual void draw() const**
  Renders the particle to the screen.
- **virtual bool isAlive() const**
  Checks if the particle is still active (i.e., has not expired).

7.1.2 Details

The `IParticle` interface provides a blueprint for implementing various types of particles, ensuring consistency in behavior across different particle types such as expulsion and incantation particles.

7.2 AParticle

The `AParticle` class is an abstract base class that implements common functionality for particles.

7.2.1 Inherits

`AParticle` inherits from `IParticle`.

7.2.2 Constructor

- **AParticle(float x, float y, float z, float vx, float vy, float vz, float lifetime)**
  Constructs a particle with initial position, velocity, and lifetime.

7.2.3 Methods

- **void update(float deltaTime) override**
  Updates the position and lifetime of the particle based on elapsed time.
- **bool isAlive() const override**
  Checks if the particle is alive based on its remaining lifetime.

7.2.4 Protected Attributes

- **_x, _y, _z:**
  Current position coordinates of the particle.
- **_vx, _vy, _vz:**
  Velocity components of the particle.
- **_lifetime:**
  Remaining lifetime of the particle.
- **_initialLifetime:**
  Initial lifetime of the particle.

7.2.5 Details

The `AParticle` class serves as a base for specific particle implementations. It provides methods to update the particle's position over time and determine if the particle is still active based on its lifetime.

Chapter 8

Text

The `Text` class provides functionality to render 3D text in the game environment.

8.0.1 Methods

- **void DrawTextCodepoint3D(Font font, int codepoint, Vector3 position, float fontSize, bool backface, Color tint)**
  Draws a single 3D text character at the specified position.

- **void DrawText3D(Font font, const char *text, Vector3 position, float fontSize, float fontSpacing, float lineSpacing, bool backface, Color tint)**
  Draws a string of 3D text starting at the specified position.

- **Vector3 MeasureText3D(Font font, const char* text, float fontSize, float fontSpacing, float lineSpacing)**
  Measures the dimensions (width, height) of a block of 3D text without drawing it.

- **void DrawTextWave3D(Font font, const char *text, Vector3 position, float fontSize, float fontSpacing, float lineSpacing, bool backface, WaveTextConfig* config, float time, Color tint)**
  Draws a string of 3D text with a wave effect applied, based on the provided configuration.

- **Vector3 MeasureTextWave3D(Font font, const char* text, float fontSize, float fontSpacing, float lineSpacing)**
  Measures the dimensions (width, height) of a block of 3D text with a wave effect without drawing it.

8.0.2 Details

The `Text` class implements methods to render text in a 3D space using a specified font. It provides flexibility to draw regular 3D text, measure text dimensions, and apply wave effects to text for dynamic visual presentation in the game.

Chapter 9

Exceptions

The `Exceptions` module provides custom exception classes for error handling in the Zappy project.

9.0.1 Exception

`Exception` class is the base class for all exceptions in the Zappy project.

9.0.2 MainException

`MainException` class represents exceptions related to main operations in the Zappy project.

9.0.3 ApiException

`ApiException` class represents exceptions related to API operations in the Zappy project.

9.0.4 Details

The `Exceptions` module defines three exception classes: `Exception`, `MainException`, and `ApiException`. Each class inherits from `std::exception` and provides a custom error message through the `what()` method. These classes are used to handle specific types of errors encountered during execution of the Zappy application.

Chapter 10

Tests

The `Tests` chapter provides insights into the testing practices adopted in the Zappy project. It highlights the use of Test Driven Development (TDD) and the role of testing in ensuring project progress and functionality.

10.0.1 Methodology

The Zappy project employs Test Driven Development (TDD) methodology to guide the development process. TDD involves writing tests before implementing the actual code functionality. This approach ensures that all code is thoroughly tested and meets specified requirements.

10.0.2 Testing Table

Throughout the project, testing is conducted using a structured testing table. This table includes a list of test points to be manually validated, accompanied by checkboxes to indicate whether the expected outcomes of each test were met. This iterative testing process allows the team to track project progress and verify functionality regularly.

10.0.3 Frequency

Tests based on the testing table are performed multiple times per week. This frequency ensures that newly implemented features and functionalities are rigorously tested and integrated into the project without introducing regressions.

10.0.4 Benefits

The testing practices in the Zappy project serve multiple purposes:

- Ensuring the correctness and reliability of implemented features.
- Tracking project progress and verifying the fulfillment of project requirements.
- Facilitating early detection and resolution of issues through proactive testing.

Chapter 11

Conclusion

11.1 Project Recap

The Zappy project was developed with the goal of creating an interactive game that incorporates various modern development technologies and concepts. Throughout this document, we have explored different components of the project, from initial design to detailed implementation.

11.2 Key Learnings

During the development of Zappy, several key learnings were identified:

- **Collaboration and Organization:** Effective collaboration within the team was crucial for project completion, with structured organization facilitating task distribution and issue resolution.
- **Adoption of Best Practices:** Embracing practices such as Test-Driven Development (TDD) ensured code quality and minimized regressions.
- **Flexibility and Adaptability:** The ability to adapt to technical challenges and adjust designs based on evolving project requirements was essential for meeting objectives.