ZAPPY Documentation

Multiplayer Network Game Simulator

Participants

- Louis FIHOL-VALANTIN
- Ethan BRANCHEREAU
- Lilyan BOUYER
- Thea GORY-LAURET
- Yanis BEN SAID

📘 Summary

Chapter one

✨ Welcome to our Zappy Documentation

Zappy is a multiplayer simulation project designed and developed during our second year at EPITECH. The objective was to build an interactive, network-based ecosystem where strategy and coordination are key to victory.

In this simulation, teams of players evolve by gathering resources and working together on a shared tile-based map. The end goal? To guide a team of six players to their final evolution stage, requiring cooperation, planning, and efficient execution.

To achieve this, the project is divided into three interconnected components:

- **Server** — the central system that governs the game mechanics, player movements, resource management, and communication protocols.
- **Graphical Interface (GUI)** — a visual tool that reflects the real-time state of the server and allows observers to follow the game's progression dynamically.
- **Artificial Intelligence (AI)** — autonomous players that analyze the game environment and make strategic decisions to evolve efficiently.

Zappy offered us a valuable opportunity to sharpen our understanding of network communication, software architecture, and teamwork in a large-scale project setting. Through collaboration and iteration, we learned how to turn complex requirements into a functioning, playable system.

ℹ️ Information

This project was built by a group of five students. To ensure smooth progress, we decided to assign a lead person to each of the three main components: Server, GUI, and AI. The two remaining members were designated as "supporters," allowing them to step in wherever needed and assist on any part facing delays or increased complexity.

Our time management was structured using a **GitHub Project board**, which allowed us to break down our tasks precisely, track their progress, and prioritize efficiently. Additionally, we maintained a **Discord server** with dedicated channels for each section of the project. This setup helped us quickly communicate issues, share push notifications, and request help when blockers arose.

This organization proved to be extremely effective and was key to ensuring that every part of the project was delivered on time.

Chapter three

3.1 Usage

Compilation

The project uses a unified Makefile system with component-specific targets:

```
make zappy_server    # Compile the C server
make zappy_gui       # Compile the C++ GUI
make zappy_ai        # Compile the Python AI
make                 # Compile all components
```

Server Launch

```
./zappy_server -p port -x width -y height -n name1 name2 -c clientsNb -f freq
```

Parameters:

- `-p port`: Network port for client connections
- `-x width`: World map width (tiles)
- `-y height`: World map height (tiles)
- `-n name1 name2...`: Team names (space-separated)
- `-c clientsNb`: Maximum clients per team
- `-f freq`: Time frequency multiplier (default: 100)

Example:

```
./zappy_server -p 4242 -x 10 -y 10 -n Gryffindor Slytherin -c 5 -f 100
```

GUI Connection

./zappy_gui -p port -h machine

Parameters:

- `-p port`: **Server port to connect to**
- `-h machine`: **Hostname/IP of the server (default: localhost)**

Example:

```
./zappy_gui -p 4242 -h localhost
```

AI Client

./zappy_ai -p port -n name -h machine

Parameters:

- `-p port`: **Server port to connect to**
- `-n name`: **Team name to join**
- `-h machine`: **Server hostname (default: localhost)**

Example:

```
./zappy_ai -p 4242 -n Gryffindor -h localhost
```

3.2 Commands

Player Commands Protocol

All player commands follow a simple text-based protocol. Each command is sent as a string terminated by `\n`.

Movement Commands

```
// From commands.c - Movement implementation
void cmd_forward(struct s_player *p)
{
    static const int DX[4] = {0, 1, 0, -1};
    static const int DY[4] = {-1, 0, 1, 0};
    const char *msg = "ok\n";

    p->x = (p->x + DX[p->dir] + p->world->w) % p->world->w;
    p->y = (p->y + DY[p->dir] + p->world->h) % p->world->h;
    write(p->fd, msg, strlen(msg));
    if (p->net)
        gui_broadcast_ppo(p->net, p);
    if (p->q_len > 0)
        --p->q_len;
}
```

Command	Function	Time Cost	Response	Description
`Forward`	Move one tile forward	7/f	`ok`	Moves player in current direction
`Right`	Turn 90° clockwise	7/f	`ok`	Changes player orientation

`Left`	Turn 90° counter-clockwise	7/f	`ok`	Changes player orientation

Information Commands

Command	Function	Time Cost	Response	Description
`Look`	Scan surroundings	7/f	`[tile1,tile2,...]`	Returns vision field contents
`Inventory`	Check resources	1/f	`[food n, linemate n,...]`	Lists player's inventory
`Connect_nbr`	Team slots	0/f	`value`	Available team connection slots

Action Commands

Command	Function	Time Cost	Response	Description
`Take object`	Pick up resource	7/f	`ok/ko`	Collect item from current tile
`Set object`	Drop resource	7/f	`ok/ko`	Place item on current tile
`Eject`	Push other players	7/f	`ok/ko`	Eject players from current tile

Advanced Commands

Command	Function	Time Cost	Response	Description
`Fork`	Create egg	42/f	`ok`	Spawn new team slot
`Broadcast text`	Send message	7/f	`ok`	Broadcast to all players
`Incantation`	Level up ritual	300/f	`Elevation underway...`	Attempt evolution

Command Processing Architecture

```
// From commands.c - Command mapping system
static const cmd_map_t CMD_MAP[] = {
    {"Forward", cmd_forward, 7},
    {"Right", cmd_right, 7},
    {"Left", cmd_left, 7},
    {"Look", cmd_look, 7},
    {"Eject", cmd_eject, 7},
    {"Fork", cmd_fork, 42},
    {"Connect_nbr", cmd_connect_nbr, 0},
};
```

Command Queue System

The server implements a sophisticated command queuing system:

- Players can queue up to 10 commands before server blocking
- Commands are executed asynchronously based on their time cost
- Queue length is tracked per player: `pl->q_len`

3.3 Server

Core Architecture

The server is built as a single-threaded, event-driven system using `poll()` for socket multiplexing.

Main Loop Structure

```
// From main.c - Core server loop
static void run_loop(net_t *net, scheduler_t *sched)
{
    uint64_t now;
    int poll_timeout;
    uint64_t next_event;

    while (1) {
        now = now_ms();
        next_event = scheduler_time_until_next(sched, now);
        poll_timeout = (next_event > 50) ? 50 : (int)next_event;
        net_poll_once(net, poll_timeout);
        now = now_ms();
        hunger_check(net, now);
        scheduler_run_ready(sched, now);
    }
}
```

Key Components

1. Network Layer (`net_poll.c`)

- Socket Management: Handles client connections and disconnections
- Non-blocking I/O: All operations are asynchronous
- Client Authentication: Team joining and GUI identification
- Connection Limits: Maximum `NET_MAX_FDS` concurrent connections

```
// From net_poll.c - Client handling
void handle_client(net_t *net, int idx)
{
    player_t *pl = (player_t *)net->players[idx];
    char buf[256];
    ssize_t r = read(net->pfds[idx].fd, buf, sizeof(buf));

    if (r <= 0) {
        drop_fd(net, idx);
        return;
    }
    if (!pl->authed) {
        authenticate_player(net, idx, buf, r);
        return;
    }
    player_feed(pl, buf, (size_t)r, net->sched);
}
```

2. Scheduler System (`scheduler.c`)

The server uses a min-heap priority queue for efficient command scheduling:

```
// From scheduler.c - Heap-based scheduling
bool scheduler_push(scheduler_t *s, action_t act)
{
    int i;
    if (s->len >= SCHED_MAX)
        return false;
    i = s->len;
    s->len += 1;
    s->items[i] = act;
    // Bubble up to maintain heap property
    while (i && s->items[i].exec_at < s->items[(i - 1) / 2].exec_at) {
        swap(&s->items[i], &s->items[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
    return true;
}
```

Features:

- Time-based execution: Commands execute at precise timestamps
- Player action removal: Cleanup when players disconnect
- Efficient scheduling: O(log n) insertion and extraction

3. World Management (`world.c`)

- Spherical topology: Wraparound edges (players exit right, enter left)
- Resource spawning: Periodic replenishment every 20 time units
- Tile-based storage: Efficient 2D array representation

4. Incantation System (`command_incantation.c`)

Complex elevation rituals with strict validation:

```
// From command_incantation.c - Elevation requirements
const req_t REQS[8] = {
    {0, {0}},                          // Level 0->1
    {1, {0, 1, 0, 0, 0, 0, 0}},        // Level 1->2: 1 player, 1 linemate
    {2, {0, 1, 1, 1, 0, 0, 0}},        // Level 2->3: 2 players, linemate+deraumere+sibur
    {2, {0, 2, 0, 1, 0, 2, 0}},        // Level 3->4: 2 players, 2 linemate, 1 sibur, 2 phiras
    // ... additional levels
};
```

Validation Process:

- Pre-check: Verify requirements before starting ritual
- Duration: 300 time units of vulnerability
- Post-check: Re-verify requirements after completion
- Resource consumption: Items removed only on success

5. GUI Protocol Integration

Real-time communication with graphical clients:

```
// From gui.c - Broadcasting tile changes
void gui_broadcast_tile(net_t *net, int x, int y)
{
    char line[GUI_BUF_SZ];
    size_t n;
    tile_t *t = &net->world->tiles[y * net->world->w + x];

    n = (size_t)snprintf(line, sizeof(line),
        "bct %d %d %u %u %u %u %u %u\n",
        x, y,
        t->res[RES_FOOD], t->res[RES_LINEMATE],
        t->res[RES_DERAUMERE], t->res[RES_SIBUR],
        t->res[RES_MENDIANE], t->res[RES_PHIRAS],
        t->res[RES_THYSTAME]);
    broadcast(net, line, n);
}
```

Server Performance Features

Memory Management

- Static allocation: Fixed-size arrays for predictable performance
- Resource pooling: Reuse of player and world objects
- Garbage collection: Automatic cleanup of disconnected players

Timing System

- Millisecond precision: Uses `gettimeofday()` for accurate timing
- Frequency scaling: All timings scale with server frequency parameter
- Hunger mechanics: Players must consume food every 126 time units

Error Handling

- Graceful disconnection: Proper cleanup of player state
- Command validation: Invalid commands return `ko\n`
- Resource limits: Queue and connection limits prevent server overload

The server architecture ensures scalability, reliability, and real-time responsiveness while maintaining the complex game mechanics required for the Zappy simulation.

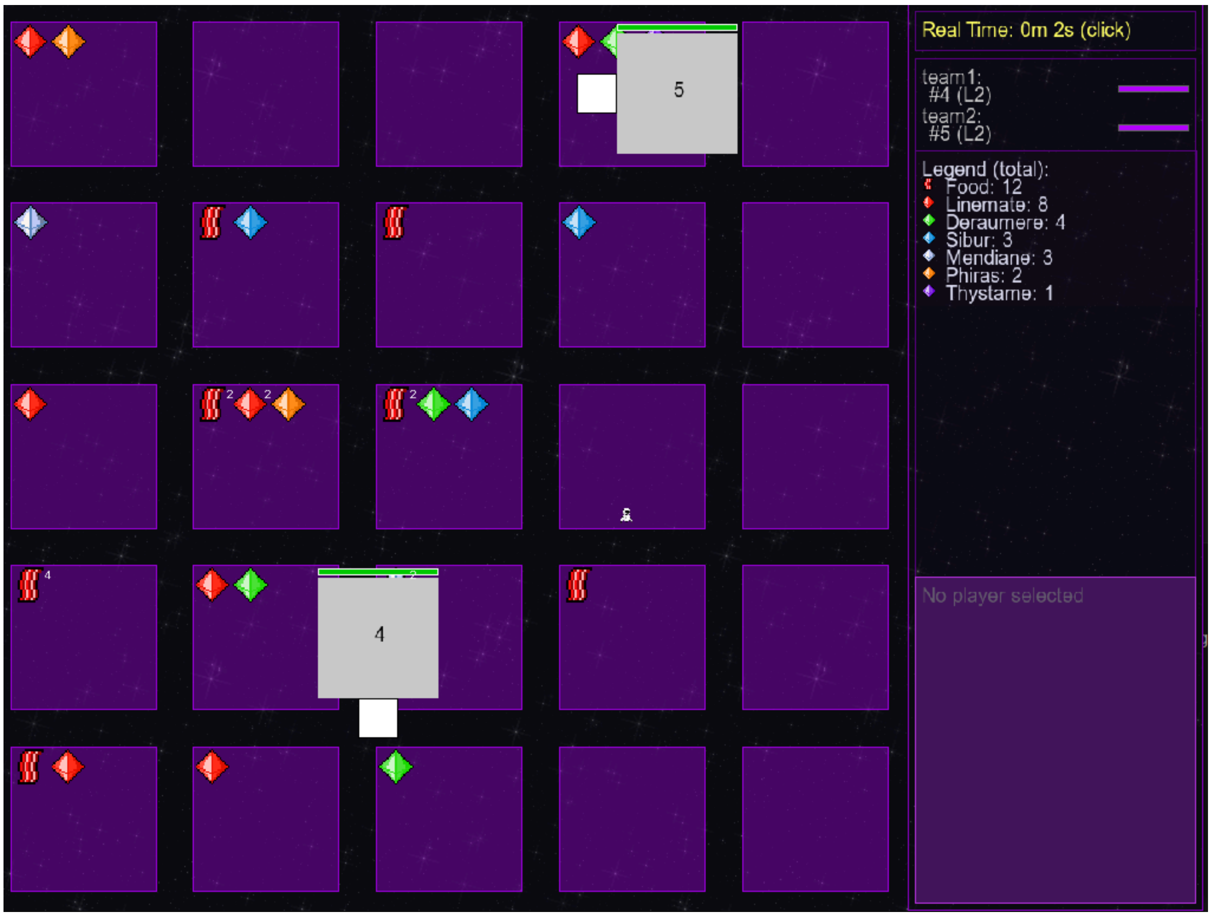
3.4 GUI

Architecture Overview

The GUI is built using C++ with SDL2 and follows a modern component-based architecture with real-time server synchronization.

Core Application Structure

```
// From App.hpp - Main application class
class App {
    private:
        // SDL Infrastructure
        SDL_Window* window = nullptr;
        SDL_Renderer* renderer = nullptr;

        // Game Components
        std::unique_ptr<BoardRenderer> board;
        std::unique_ptr<AudioManager> audio;
        std::unique_ptr<PlayerAnimator> playerAnimator;
        std::unique_ptr<EffectSystem> effectSystem;
        std::unique_ptr<SidebarRenderer> sidebarRenderer;

        // Server Communication
        ServerUpdateManager server;

        // Game State
        enum class GameState { MENU, GAME };
        GameState state = GameState::MENU;
};
```


Real Time: 0m 2s (click)

team1:
 #4 (L2)
team2:
 #5 (L2)

Legend (total):
Food: 12
Linemate: 8
Deraumere: 4
Sibur: 3
Mendiane: 3
Phiras: 2
Thystame: 1

No player selected

Key Components

1. Real-Time Server Communication

The GUI implements the official Zappy GUI Protocol for seamless server communication:

Connection Process:

```
// From main.cpp - Server connection with IPv4 validation
bool parseArguments(int argc, char* argv[], GuiConfig& config) {
   // Parse -p port and -h hostname arguments
   if (isValidIPv4(hostname)) {
      config.hostname = hostname;
   } else {
      std::cerr << "Invalid IPv4, using 127.0.0.1" << std::endl;
      config.hostname = "127.0.0.1";
   }
}
```

Protocol Implementation: The GUI supports the complete protocol specification:

- **Map Data:** `msz X Y`, `bct X Y q0 q1...`, `mct`, `tna N`

- **Player Events:** `pnw #n X Y O L N`, `ppo #n X Y O`, `plv #n L`
- **Game Events:** `pic X Y L #n...`, `pie X Y R`, `pfk #n`, `pdi #n`
- **Resource Events:** `pdr #n i`, `pgt #n i`
- **System:** `sgt T`, `sst T`, `seg N`, `smg M`

2. Advanced Rendering System

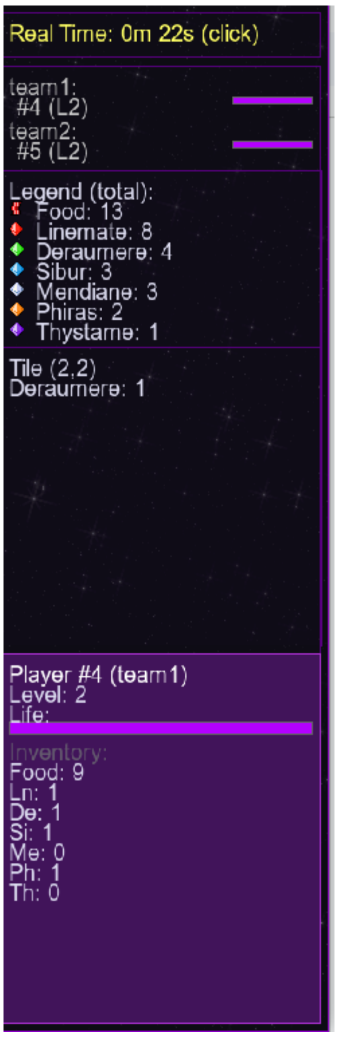
```
// From BoardRenderer.cpp - Optimized tile rendering
void BoardRenderer::render(const LayoutMetrics& metrics,
                        std::function<const ServerUpdateManager::Resources*(int, int)>
getResources,
                        std::function<std::vector<const ServerUpdateManager::Egg*>(int, int)>
getEggs,
                        int selectedX, int selectedY) {

    for (int y = 0; y < _boardHeight; ++y) {
        for (int x = 0; x < _boardWidth; ++x) {
            // Draw cell background with selection highlighting
            drawCellBackground(x, y, metrics.cellWidth, metrics.cellHeight, isSelected);

            // Draw resources with sprite icons
            const auto* resources = getResources(x, y);
            if (resources && _showResourceIcons) {
                drawResources(x, y, metrics.cellWidth, metrics.cellHeight, resources);
            }

            // Draw eggs with team colors
            auto eggs = getEggs(x, y);
            for (const auto* egg : eggs) {
                if (egg && egg->alive) {
                    drawEgg(x, y, metrics.cellWidth, metrics.cellHeight, egg->team,
isSelected);
                }
            }
        }
    }
}
```

Resource Visualization:

- Sprite-based rendering with PNG assets for food and gems
- Fallback colored squares for missing assets
- Quantity indicators for multiple resources per tile
- Team-colored eggs with dynamic texture generation

3. Audio System Integration

```
// From AudioManager.cpp - SDL_mixer integration
class AudioManager {
    public:
        bool loadSFX(const std::string& id, const std::string& path);
        bool loadMusic(const std::string& id, const std::string& path);
        void playSFX(const std::string& id, int loops = 0);
        void playMusic(const std::string& id, int loops = -1);
        void setMusicVolume(int volume); // 0-128 range
};
```



```
Real Time: 0m 22s (click)

team1:
 #4 (L2)
team2:
 #5 (L2)

Legend (total):
◆ Food: 13
◆ Linemate: 8
◆ Deraumere: 4
◆ Sibur: 3
◆ Mendiane: 3
◆ Phiras: 2
◆ Thystame: 1

Tile (2,2)
Deraumere: 1

Player #4 (team1)
Level: 2
Life:

Inventory:
Food: 9
Ln: 1
De: 1
Si: 1
Me: 0
Ph: 1
Th: 0
```

Audio Features:

- Event-driven sound effects: Incantations, player login/logout, resource drops
- Background music system: Multiple tracks with smooth transitions
- Volume controls: Separate SFX and music volume management
- Easter egg audio: Special "Snoop Dogg" theme with custom music

4. Interactive Features

Mouse Interactions:

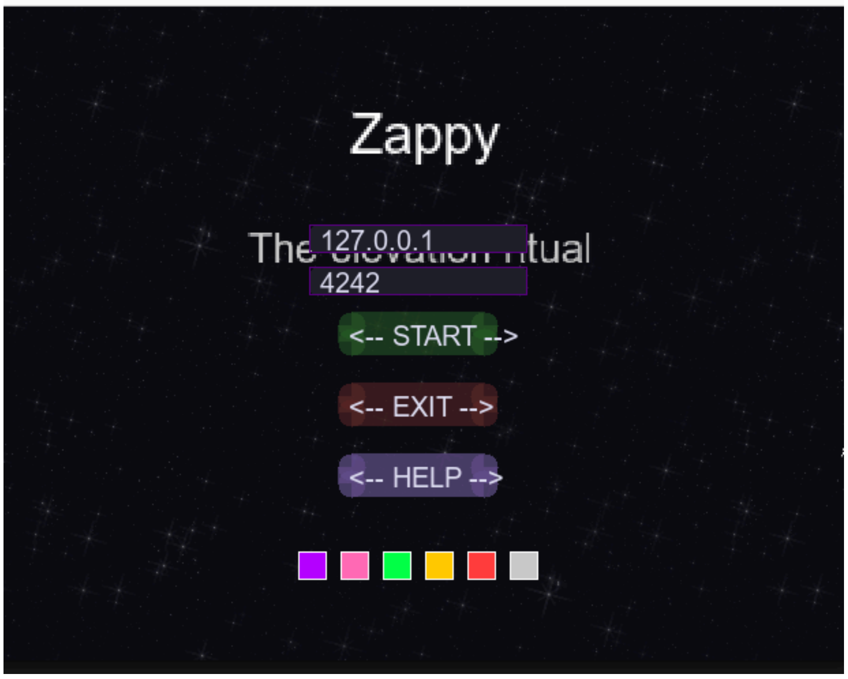
```
// From App.cpp - Click handling with layout management
if (layoutManager.isPointInBoardArea(e.button.x, e.button.y, currentLayout)) {
   SDL_Point boardPos = layoutManager.screenToBoard(e.button.x, e.button.y,
currentLayout);
   if (boardPos.x >= 0 && boardPos.y >= 0) {
      selectedX = boardPos.x;
      selectedY = boardPos.y;
      // Add visual effect at click location
      effectSystem->addTileBurst(boardPos.x, boardPos.y, {255, 255, 100, 255}, 8, 0.4f);
   }
}
```

Keyboard Controls:

- ESC: Pause game / Close help screens
- F: Toggle fullscreen mode
- R: Cycle through resolutions (windowed mode)
- T: Quick theme switching
- Arrow Keys: Navigate help pages
- Mouse Wheel: Scroll sidebar content

5. Visual Effects System

- Particle effects for incantations and level-ups
- Smooth animations with tween system
- Player movement interpolation for fluid motion
- Visual feedback for all game events



Build System

Makefile Structure:

```
# From src/gui/Makefile - SDL2 integration
CXX = g++
CXXFLAGS = -g -O0 -Wall -Wextra -Werror -std=c++17 `sdl2-config --cflags`
LDFLAGS = `sdl2-config --libs` -lSDL2_mixer -lSDL2_ttf -lSDL2_image

# Automatic source discovery
SRC = $(shell find src/core src/engine -name '*.cpp')
```

Dependencies:

- SDL2: Core graphics and windowing
- SDL2_mixer: Audio playback and mixing
- SDL2_ttf: Text rendering with TrueType fonts
- SDL2_image: PNG/JPG asset loading

Performance Optimizations

Rendering Efficiency

- VSync support for smooth frame rates
- Text batching for improved font rendering performance
- Viewport optimization for large displays
- Texture caching for team-colored sprites

Memory Management

```
// From App.cpp - Proper resource cleanup
App::~App() {
    if (cursorTexture) SDL_DestroyTexture(cursorTexture);
    if (cursorSurface) SDL_FreeSurface(cursorSurface);
    if (parallaxTexture) SDL_DestroyTexture(parallaxTexture);

    // Cleanup team egg textures
    for (auto& [_, texture] : _teamEggTextures) {
        if (texture) SDL_DestroyTexture(texture);
    }
}
```

Special Features

Theme System

- Multiple visual themes: GameCube, Gameboy Pink, Matrix, N64, Red, White
- Runtime theme switching with instant visual updates
- Theme preview in menu and pause screens

Easter Egg: SNOOP_DOG Mode

```
// From App.cpp - Dynamic asset switching
if (team == "SNOOP_DOG") {
    if (loadSnoopAssets()) {
        snoopModeActive = true;
        board->setSnoopMode(true, snoopResourceTextures);
        sidebarRenderer->setSnoopMode(true, snoopResourceTextures);
        // Switch to Snoop Dogg music
        audio->playMusic("snoop", -1);
    }
}
```

Help System

- **Multi-page help screens: Lore, Rules, GUI Controls**
- **Interactive navigation with mouse and keyboard**
- **Context-sensitive information for game mechanics**

User Interface Design

Responsive Layout:

- **Adaptive scaling based on window resolution**
- **Flexible sidebar with scroll support**
- **Dynamic text sizing for different screen sizes**
- **Custom cursor with hotspot positioning**

Information Display:

- **Real-time player tracking with ID-based selection**
- **Resource counters with visual icons**
- **Game timer with real/game time toggle**
- **Team statistics and player inventories**

The GUI provides a comprehensive, real-time visualization of the Zappy game world with professional-grade features including advanced graphics, immersive audio, and intuitive user interactions.

3.5 AI

Architecture Overview

The AI client is implemented in Python 3 and features a sophisticated state-driven autonomous system with team coordination capabilities.

Core AI Structure

```
# From ia.py - Main AI class with intelligent decision making
class ZappyIA:
    ELEVATION_REQUIREMENTS = [
        None,
        {"player": 1, "linemate": 1, "deraumere": 0, "sibur": 0, "mendiane": 0, "phiras": 0,
"thystame": 0},  # 1 -> 2
        {"player": 2, "linemate": 1, "deraumere": 1, "sibur": 1, "mendiane": 0, "phiras": 0,
"thystame": 0},  # 2 -> 3
        # ... complete elevation requirements for all levels
    ]

    def __init__(self, team="team1", host="localhost", port=4242, verbose=True):
        self.level = 1
        self.inventory_cache = {}
        self.state = "EXPLORING"
        self.message_queue = []
```

Intelligent Behaviors

1. State Machine Architecture

The AI uses a finite state machine for intelligent decision-making:

Primary States:

- EXPLORING: Random movement and resource discovery
- GATHERING: Targeted resource collection
- FORAGING: Emergency food collection for survival
- CALLING_FOR_INCANTATION: Broadcasting for team coordination
- GOING_TO_INCANTATION: Responding to team calls
- LEVELING_UP: Executing elevation rituals

2. Survival Intelligence

```
# From ia.py - Smart survival mechanics
def _needs_to_forage(self):
    """Check if the AI needs to find food."""
    food_stock = self.inventory_cache.get("food", 0)
    return food_stock < 10

def _search_for_item(self, item_name):
    """Look for a specific item and move towards it."""
    look_raw = self.look()
    tiles = self._parse_look(look_raw)

    for i, tile_content in enumerate(tiles):
        if item_name in tile_content:
            if i == 0:
                self.take(item_name)
                self._update_inventory()
            else:
                self._go_to_tile(i)
            return True

    # Explore if not found in vision
    random.choice([self.left, self.right, self.forward])()
    return False
```

Survival Features:

- Food monitoring: Maintains minimum 10 food units
- Emergency foraging: Prioritizes food when critically low
- Inventory management: Real-time resource tracking
- Adaptive exploration: Intelligent pathfinding when resources not visible

3. Team Coordination System

```
# From ia.py - Advanced team communication
TEAM_SECRET = "zappy_rocks"

def run(self):
    """State-driven autonomous loop with communication."""
    if self.message_queue:
        msg = self.message_queue.pop(0)
        try:
            _, content = msg.split(',', 1)
            secret, level_str, action = content.strip().split(':')
            if secret == TEAM_SECRET and action == "INCANTATION_START" and
int(level_str) == self.level:
                self.state = "GOING_TO_INCANTATION"
                self._log(f"[IA] Received call for level {self.level} incantation.")
        except (ValueError, IndexError) as e:
            self._log(f"[IA] Could not parse broadcast: {msg}")
```

Coordination Features:

- Secure messaging: Team-specific secret for authenticated communication
- Level-specific calls: Only responds to same-level incantation requests
- Directional navigation: Follows broadcast directions to reach team members
- Leadership rotation: Any player can initiate team coordination

4. Strategic Resource Management

```
# From ia.py - Intelligent resource planning
def _get_needed_stones(self):
    """Get a dict of stones needed for the next elevation."""
    if self.level >= 8:
        return {}

    reqs = self.ELEVATION_REQUIREMENTS[self.level].copy()
    needed = {}

    for stone, required_count in reqs.items():
        if stone == "player":
            continue
        current_count = self.inventory_cache.get(stone, 0)
        if current_count < required_count:
            needed[stone] = required_count - current_count

    return needed
```

Strategic Planning:

- Goal-oriented collection: Only gathers stones needed for next level
- Requirement optimization: Calculates exact deficits for efficient gathering
- Elevation preparation: Ensures all prerequisites before attempting rituals
- Tile management: Clears unwanted items from incantation sites

5. Advanced Movement Intelligence

```
# From ia.py - Sophisticated pathfinding
def _go_to_tile(self, tile_index):
    """Move towards a tile index from the 'look' command output."""
    if tile_index == 0:
        return

    moves = []
    if tile_index in [1, 2, 3]:
        moves.append(self.forward)
        if tile_index == 1: moves.append(self.left)
        if tile_index == 3: moves.append(self.right)
    elif tile_index in [4, 5, 6, 7, 8]:
        moves.append(self.forward)
        moves.append(self.forward)
        # Complex multi-step pathfinding for distant tiles
    for move in moves:
        move()
        time.sleep(0.1)
```

Movement Features:

- Vision-based navigation: Calculates optimal paths to visible resources
- Multi-step pathfinding: Handles complex routes to distant targets
- Collision avoidance: Adapts when paths are blocked
- Exploration patterns: Randomized movement when targets not visible

Communication Protocol

Network Layer

```
# From ia.py - Robust network communication
def cmd(self, command: str) -> str:
    """Send command to server and wait for response."""
    self._send_line(command)
    try:
        response = self._read_response()
    except socket.timeout:
        self._log("[IA] Timeout: no response")
        response = ""
    if response == "dead":
        raise SystemExit("[IA] You are dead :(")
    return response
```

Network Features:

- Timeout handling: Graceful recovery from server delays
- Death detection: Automatic termination on player death
- Message queuing: Asynchronous broadcast message handling
- Connection robustness: Maintains stable server communication

Server Command Integration

The AI implements the complete Zappy command set:

- **Movement:** `Forward`, `Right`, `Left`
- **Perception:** `Look`, `Inventory`
- **Actions:** `Take object`, `Set object`
- **Communication:** `Broadcast text`
- **Advanced:** `Fork`, `Incantation`, `Connect_nbr`

Autonomous Decision Logic

Priority System

The AI follows a sophisticated priority hierarchy:

1. Survival (Critical): Food gathering when below 10 units
2. Team Coordination (High): Responding to incantation calls
3. Reproduction (Medium): Forking when food > 20 and slots available
4. Elevation (Medium): Leading incantations when requirements met
5. Resource Gathering (Low): Collecting stones for future levels
6. Exploration (Lowest): Random movement when no specific goals

Intelligent Incantation Management

```
# From ia.py - Complex elevation logic
if not needed_stones:
    # Check if tile is properly prepared
    on_tile_str = self._parse_look(self.look())[0]
    tile_is_clear = True
    required_stones_for_check = {stone for stone, count in reqs.items()
                               if count > 0 and stone != "player"}

    # Clean unwanted items from incantation site
    for item in on_tile_items:
        if item not in required_stones_for_check and item not in ["player", "food"]:
            self.take(item)
            tile_is_clear = False

    # Verify player count and execute ritual
    players_on_tile = on_tile_items.count("player")
    if players_on_tile >= reqs["player"]:
        # Place required stones and attempt incantation
        for stone, count in reqs.items():
            if stone == "player": continue
            for _ in range(count): self.set(stone)

        result = self.incantation()
        # Handle success/failure scenarios
```

Build System

Makefile Structure:

```
# From src/ia/Makefile - Python wrapper generation
NAME = ../../zappy_ia
PYTHON = python3
SRC = ia.py

$(NAME): $(SRC)
```

```
echo "#!/bin/sh" > $(NAME)
echo '$(PYTHON) $(dirname $0)/src/ia/ia.py "$@"' >> $(NAME)
chmod +x $(NAME)
```

Deployment Features:

- Shell wrapper: Creates executable binary for easy deployment
- Argument forwarding: Passes all CLI arguments to Python script
- Path resolution: Automatically locates source files relative to binary
- Cross-platform compatibility: Works on any Unix-like system with Python 3

Performance Optimizations

Efficiency Features

- Inventory caching: Reduces server requests by caching resource counts
- Message queuing: Asynchronous processing of broadcast communications
- Strategic delays: Optimized timing to balance responsiveness and server load
- Resource prioritization: Focuses on immediate needs rather than hoarding

Error Handling & Robustness

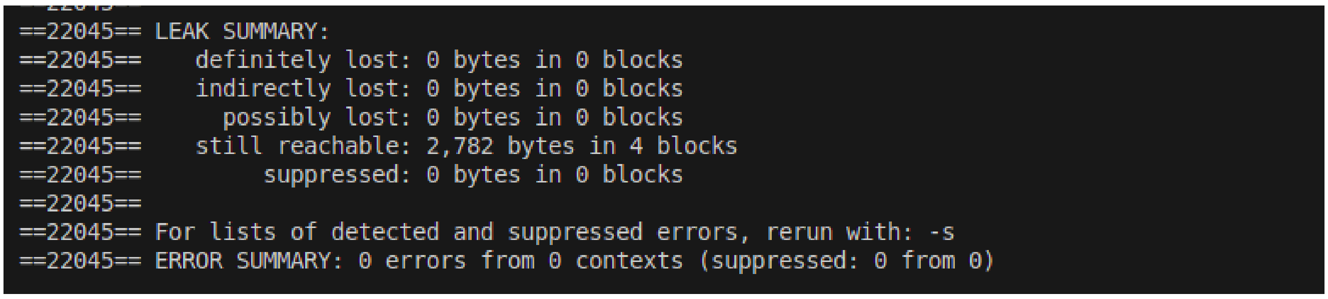
- Connection recovery: Graceful handling of network timeouts
- State consistency: Maintains coherent internal state across server responses
- Message validation: Secure parsing of team communications
- Fallback behaviors: Default actions when optimal strategies fail

The AI demonstrates advanced strategic thinking, efficient resource management, and sophisticated team coordination, making it a formidable autonomous player capable of competing effectively in complex Zappy scenarios.

Clean Project🥳

✅ Zero Memory Leaks Confirmed

Valgrind validation shows clean memory management across all server components.



```
==22045== LEAK SUMMARY:
==22045==    definitely lost: 0 bytes in 0 blocks
==22045==    indirectly lost: 0 bytes in 0 blocks
==22045==      possibly lost: 0 bytes in 0 blocks
==22045==    still reachable: 2,782 bytes in 4 blocks
==22045==         suppressed: 0 bytes in 0 blocks
==22045==
==22045== For lists of detected and suppressed errors, rerun with: -s
==22045== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```