Zappy

Documentation

Presentation

Zappy is the last project of the 2nd year of EPITECH.

This project was carried out by 6 people for 5 weeks.

The goal of this project is to create a network game where several teams confront each other on a tiles map containing resources.

The winning team is the first one where at least 6 players reach the maximum elevation (level 8).

This is project is separated in 3 parts:

- **GUI:** which allow the user to observe the game in progress
- **AI:** This is the part that controls every player of the game.
- **Server:** which handles all the game mechanics and progression, as well as all the players' behaviors.

The different parts of the project communicate as shown below:

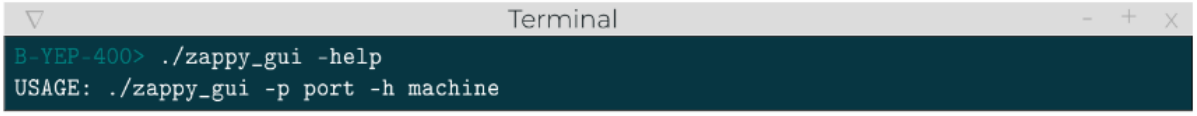
Each part of this project can be stated with a specific command:

- For the SERVER:

```
Terminal
B-YEP-400> ./zappy_server -help
USAGE: ./zappy_server -p port -x width -y height -n name1 name2 ... -c clientsNb -f freq
```

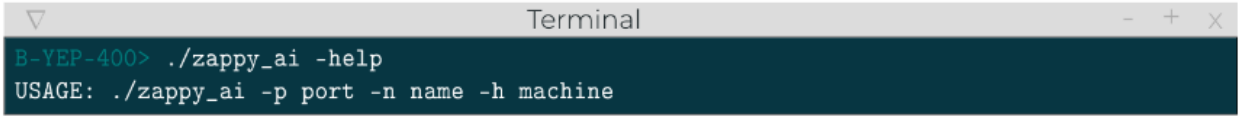
option	description
-p port	port number
-x width	width of the world
-y height	height of the world
-n name1 name2 ...	name of the team
-c clientsNb	number of authorized clients per team
-f freq	reciprocal of time unit for execution of actions

- For the GUI:



option	description
-p port	port number
-h machine	name of the machine

- For the AI:



option	description
-p port	port number
-n name	name of the team
-h machine	name of the machine; localhost by default

Server

The server manages all the commands sent by the AI and sends events to the GUI as well as the game logic.

Networking

The communication between the server and the different parts of the project are done through tcp sockets.

The server has a select loop, allowing new clients to connect, as well as handling reading and writing from the different sockets.

```c
bool server_loop(server_t *serv)
{
    struct timeval time = {0, (1.0f / (float) serv->freq) * 1000000};
    fd_set fdset;
    fd_set fdwset;

    set_fd(serv, &fdset, &fdwset);
    if (select(FD_SETSIZE, &fdset, &fdwset, NULL, &time) != -1) {
        send_buffered_data(serv, &fdwset);
        if (FD_ISSET(serv->socket, &fdset) != 0)
            server_add_client(serv);
        read_client_data(serv, &fdset);
    }
    run_client_commands(serv);
    remove_old_clients(serv);
    game_update(serv);
    return (check_win(serv));
}
```

All of the command processing is done in the read_client_data function, which bufferize each of the commands sent by a client and then processes it, allowing to have up to 10 commands sent by a client without having to wait for a server response.

Communication with the AI

The complete list of commands managed by the server is displayed below:

action	command	time limit	response
move up one tile	Forward	7/f	ok
turn 90° right	Right	7/f	ok
turn 90° left	Left	7/f	ok
look around	Look	7/f	[tile1, tile2,...]
inventory	Inventory	1/f	[linemate *n*, sibur *n*,...]
broadcast text	Broadcast text	7/f	ok
number of team unused slots	Connect_nbr	-	value
fork a player	Fork	42/f	ok
eject players from this tile	Eject	7/f	ok/ko
death of a player	-	-	dead
take object	Take object	7/f	ok/ko
set object down	Set object	7/f	ok/ko
start incantation	Incantation	300/f	Elevation underway Current level: k /ko

To handle the commands, the server uses a queue with the method First in First Out (FIFO). When a new command is sent by the AI, the server puts this command at the end of the list.

```c
static void add_client_command(client_t *client)
{
    char **cmds = NULL;

    if (client->read_buffer[0] == '\0')
        return;
    cmds = str_to_array(client->read_buffer, "\n", MAX_CONCURENT_COMMANDS);
    for (size_t i = 0; cmds[i] != NULL; i++)
        list_add_elem_at_back(&client->cmds, cmds[i]);
    free(cmds);
    memset(client->read_buffer, 0, sizeof(client->read_buffer));
}
```

This list of commands is then processed, starting with the first command, the oldest, and then removed from the list after being executed.

```c
static bool handle_commands(server_t *serv, client_t *client)
{
    char *cmd = NULL;
    bool result = true;

    for (size_t i = 0; i < list_get_size(client->cmds); i++) {
        if (!is_client_ready(serv, client))
            break;
        cmd = list_get_elem_at_position(client->cmds, i);
        if (cmd == NULL)
            continue;
        if (client->state == CREATED)
            result = handle_client_login(serv, client, cmd);
        else
            result = run_command(serv, client, cmd);
        list_del_elem_at_position(&client->cmds, i);
        free(cmd);
        if (!result)
            break;
    }
    return (result);
}
```

AI Command

Each command of the AI is associated with a specific function.

We use a command dispatcher to call the correct function, based on a specific string, which then calls the associated function.

```c
static const struct command_handler_s ai_cmds[] = {
    {.command = "Forward", .nb_args = 0, .ptr = {.ai_ptr = ai_move_forward}},
    {.command = "Right", .nb_args = 0, .ptr = {.ai_ptr = ai_turn_right}},
    {.command = "Left", .nb_args = 0, .ptr = {.ai_ptr = ai_turn_left}},
    {.command = "Inventory", .nb_args = 0, .ptr = {.ai_ptr = ai_inventory}},
    {.command = "Connect_nbr", .nb_args = 0, .ptr = {.ai_ptr = ai_connect_nbr}},
    {.command = "Fork", .nb_args = 0, .ptr = {.ai_ptr = ai_fork}},
    {.command = "EndForkServer", .nb_args = 0, .ptr = {.ai_ptr = ai_end_fork}},
    {.command = "Eject", .nb_args = 0, .ptr = {.ai_ptr = ai_eject}},
    {.command = "Take", .nb_args = 1, .ptr = {.ai_ptr = ai_take_object}},
    {.command = "Set", .nb_args = 1, .ptr = {.ai_ptr = ai_set_object}},
    {.command = "Look", .nb_args = 0, .ptr = {.ai_ptr = ai_look_around}},
    {.command = "Incantation", .nb_args = 0, .ptr = {.ai_ptr = ai_elevation}},
    {.command = "EndIncantationServer", .nb_args = 0, .ptr = {.ai_ptr = ai_end_elevation}},
    {.command = "Broadcast", .nb_args = 1, .ptr = {.ai_ptr = ai_broadcast}},
};
```

```c
static bool run_ai(server_t *serv, client_t *client, const char *cmd, const char *arg)
{
    bool ret = true;

    for (size_t i = 0; i != sizeof(ai_cmds) / sizeof(ai_cmds[0]); i++) {
        if (check_cmd(i, cmd, arg)) {
            server_log(serv, PROCESS, client->fd, cmd);
            ret = ai_cmds[i].ptr.ai_ptr(serv, client, arg);
            broadcast_map_content(serv);
            return (ret);
        }
    }
    server_log(serv, WARNING, 0, "Unknown command");
    return (false);
}
```

Doing so allows us to add new functions to the AI very quickly and easily.

Communication with the GUI

The server sends events to the GUI to allow the user to observe the situation of the game in progress.

The GUI can also send commands to the server to retrieve specific information about the game or a player.

The document below describes in detail the protocol used for the communication between the two parts.

Server	Client	Details
msz X Y\n	msz\n	map size
bct X Y q0 q1 q2 q3 q4 q5 q6\n	bct X Y\n	content of a tile
bct X Y q0 q1 q2 q3 q4 q5 q6\n * nbr_tiles	mct\n	content of the map (all the tiles)
tna N\n * nbr_teams	tna\n	name of all the teams
pnw #n X Y O L N\n		connection of a new player
ppo #n X Y O\n	ppo #n\n	player's position
plv #n L\n	plv #n\n	player's level
pin #n X Y q0 q1 q2 q3 q4 q5 q6\n	pin #n\n	player's inventory
pex #n\n		expulsion
pbc #n M\n		broadcast
pic X Y L #n #n ...\n		start of an incantation (by the first player)
pie X Y R\n		end of an incantation
pfk #n\n		egg laying by the player
pdr #n i\n		resource dropping
pgt #n i\n		resource collecting
pdi #n\n		death of a player
enw #e #n X Y\n		an egg was laid by a player
ebo #e\n		player connection for an egg
edi #e\n		death of an egg
sgt T\n	sgt\n	time unit request
sst T\n	sst T\n	time unit modification
seg N\n		end of game
smg M\n		message from the server
suc\n		unknown command
sbp\n		command parameter

Symbol	Meaning
X	width or horizontal position
Y	height or vertical position
q0	resource 0 (food) quantity
q1	resource 1 (linemate) quantity
q2	resource 2 (deraumere) quantity
q3	resource 3 (sibur) quantity
q4	resource 4 (mendiane) quantity
q5	resource 5 (phiras) quantity
q6	resource 6 (thystame) quantity

Symbol	Meaning
n	player number
O	orientation: 1(N), 2(E), 3(S), 4(W)
L	player or incantation level
e	egg number
T	time unit
N	name of the team
R	incantation result
M	message
i	resource number

The connection between the server and the GUI is treated as a normal Player connection with one difference, the team's name is " GRAPHICAL". This allows the server to send the event to the correct client. This distinction is made at the connection:

```c
bool handle_client_login(server_t *serv, client_t *client, const char *cmd)
{
    static int player_index = 0;

    if (cmd[0] == '\0')
        return (false);
    strcpy(client->team_name, cmd);
    if (strncmp(cmd, "GRAPHIC", 7) == 0) {
        client->state = GRAPHICAL;
        send_initial_gui_data(serv, client);
        return (true);
    }
    if (!create_player(serv, client, player_index)) {
        server_log(serv, WARNING, 0, "Unable to create player in team");
        return (false);
    }
    client->state = AI;
    player_index++;
    send_login_answer(serv, client);
    event_connnew_player(serv, client);
    return (true);
}
```

GUI Event

As seen before, there are many events. Those events are almost all related to an action of the player. Because of that, the functions concerning the event are called directly in the function of the player.

This is an example of an event happening when the player moves its position

```c
void event_player_position(server_t *serv, const client_t *client)
{
    char *buff = player_position(serv, client->player.number);

    if (buff == NULL)
        return;
    server_log(serv, EVENT, client->fd, "player position changed");
    server_event_send_many(serv, GRAPHICAL, buff);
    free(buff);
}
```

GUI Command

The GUI also has the possibility to send specific commands to the server. As for the AI, those commands are stored in a list with their corresponding function pointers.

This provides the same simplicity as the AI command to add or modify those commands.

```c
static const struct command_handler_s gui_cmds[] = {
    {.command = "msz", .nb_args = 0, .ptr = {.gui_ptr = gui_map_size}},
    {.command = "bct", .nb_args = 2, .ptr = {.gui_ptr = gui_tile_content}},
    {.command = "mct", .nb_args = 0, .ptr = {.gui_ptr = gui_map_content}},
    {.command = "tna", .nb_args = 0, .ptr = {.gui_ptr = gui_all_name}},
    {.command = "ppo", .nb_args = 1, .ptr = {.gui_ptr = gui_player_position}},
    {.command = "plv", .nb_args = 1, .ptr = {.gui_ptr = gui_player_level}},
    {.command = "pin", .nb_args = 1, .ptr = {.gui_ptr = gui_player_inventory}},
    {.command = "sgt", .nb_args = 0, .ptr = {.gui_ptr = gui_get_time_unit}},
    {.command = "sst", .nb_args = 1, .ptr = {.gui_ptr = gui_set_time_unit}}};
```

```c
static bool run_gui(server_t *serv, client_t *client, const char *cmd, char **args)
{
    for (size_t i = 0; i != sizeof(gui_cmds) / sizeof(gui_cmds[0]); i++) {
        if (strncmp(gui_cmds[i].command, cmd, strlen(gui_cmds[i].command)) == 0
            && (str_array_length(args) - 1) >= gui_cmds[i].nb_args) {
            server_log(serv, PROCESS, client->fd, cmd);
            return (gui_cmds[i].ptr.gui_ptr(serv, client, args));
        }
    }
    server_log(serv, WARNING, 0, "Unknown command");
    server_send_data(client, "suc\n");
    return (true);
}
```

Game Mechanics

As said previously, the server handles all the game mechanics. Those mechanics are tied with the AI behavior and are necessary to be able to play the game as intended.

Those mechanics are the following:

Elevation

The elevation allows the player to level up. This process has many constraints explained in this picture

elevation	nb players	linemate	deraumere	sibur	mendiane	phiras	thystame
1->2	1	1	0	0	0	0	0
2->3	2	1	1	1	0	0	0
3->4	2	2	0	1	0	2	0
4->5	4	1	1	2	0	1	0
5->6	4	1	2	1	3	0	0
6->7	6	1	2	3	0	1	0
7->8	6	2	2	2	2	2	1

As we can see, they need to fill all the conditions to level up, meaning having the correct number of players with the same level, and the correct amount of each stone on their current cell of the map.

Those constraints are checked twice, once when starting the elevation process, and then after when the elevation ends. All the conditions need to be filled out at both checks to complete the elevation process.

On the server side, this process is handled by functions that check each condition, one for the number of players, and one for the stones on the cell.

After that, each player taking part in the process level up by one level.

As the conditions are static, we are using a list of integers to verify those conditions

```
static const int s_required[7][6] = {{1, 0, 0, 0, 0, 0}, {1, 1, 1, 0, 0, 0},
    {2, 0, 1, 0, 2, 0}, {1, 1, 2, 0, 1, 0}, {1, 2, 1, 3, 0, 0},
    {1, 2, 3, 0, 1, 0}, {2, 2, 2, 2, 2, 1}};
static const int p_required[7] = {1, 2, 2, 4, 4, 6, 6};
```

```c
bool ai_elevation(server_t *serv, client_t *cli, UNUSED const char *obj)
{
    bool val;
    int n = 0;
    int *elevated = NULL;

    if (check_stone(serv, cli) && check_nb_players(serv, cli)) {
        n = mark_player_elevating(serv, cli);
        server_send_data(cli, "Elevation underway\n");
        val = true;
        elevated = getelevated(serv, n);
        event_start_incantation(serv, cli, elevated, n);
        free(elevated);
        list_add_elem_at_back(&cli->cmds, strdup("EndIncantationServer\n"));
    } else {
        server_send_data(cli, "ko\n");
        val = false;
    }
    cli->cmd_duration = 300;
    gettimeofday(&cli->last_cmd_time, NULL);
    return val;
}
```

This process is tied to an event with the GUI, as well as the level up process.

Vision

The vision mechanism allows the player to know what is on the cell in front of it. This allows the player to base its decision on the resources needed to level up.

The vision is in correspondence with the level of the player, meaning that after each level goes up the player can see farther.

At level 1, the player can see from 0 to 3, at level 2 it can see from 0 to 8, and so on.

On the server, this process is separated into two steps; the first step is to get the cells that are in the field of view of the player.

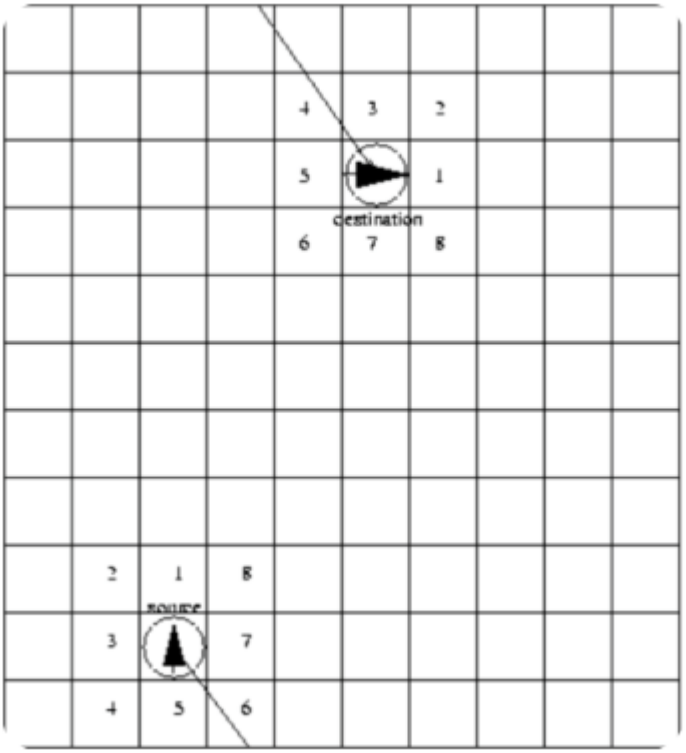
```c
static cell_t *get_case(server_t *serv, int x, int y)
{
    x = x < 0 ? serv->resX + x : x;
    y = y < 0 ? serv->resY + y : y;
    return &serv->map[x % serv->resY][y % serv->resX];
}

static cell_t *get_current_case(server_t *serv, const client_t *cli, int i, int j)
{
    if (cli->player.orient == NORTH)
        return get_case(serv, cli->player.x + j, cli->player.y - i);
    if (cli->player.orient == EAST)
        return get_case(serv, cli->player.x + i, cli->player.y + j);
    if (cli->player.orient == SOUTH)
        return get_case(serv, cli->player.x - j, cli->player.y + i);
    if (cli->player.orient == WEST)
        return get_case(serv, cli->player.x - i, cli->player.y - j);
    return NULL;
}
```

We then create a string containing the contents of each cell in the FOV.

Sound transmission

The sound transmission allows a player to send a message to the other player. This message can be anything and is used especially in the elevation process.



To do this process, the server must determine the shortest path to another player and a direction.

```c
static int map_to_local_player_direction(const ivect2D_t *pos, const player_t *player)
{
    int dir = 0;
    fvect2d_t vect = {(float) pos->x - (float) player->x,
        (float) pos->y - (float) player->y};
    fvect2d_t orient = get_multiplier(player->orient);
    float angle = atan2f(orient.y, orient.x) - atan2f(vect.y, vect.x);

    angle = angle * 360.0f / (2.0f * M_PI);
    if (angle < 0.0f)
        angle += 360;
    dir = roundf(angle / 45.0f) + 1;
    return (dir);
}
```

It’s then sent to the closest player

```c
bool ai_broadcast(server_t *serv, client_t *cli, const char *obj)
{
    size_t players = list_get_size(serv->client);
    client_t *client = NULL;

    for (size_t i = 0; i != players; i++) {
        client = list_get_elem_at_position(serv->client, i);
        if (client == NULL || client->state != AI || client == cli)
            continue;
        if (client->player.x == cli->player.x
            && client->player.y == cli->player.y) {
            send_broadcast(client, obj, 0);
            continue;
        }
        send_broadcast(client, obj, get_tile_orient(serv, cli, client));
    }
    event_broadcast(serv, cli, obj);
    cli->cmd_duration = 7;
    gettimeofday(&cli->last_cmd_time, NULL);
    server_send_data(cli, "ok\n");
    return (true);
}
```

Time

The time unit on the game is the second. Each command from the player has an execution time, meaning that you cannot do anything during this time.

An action’s execution time is calculated with the following formula: action / f, where f is an integer representing the reciprocal (multiplicative inverse) of the time unit.

The time is set by the user when starting the server and can be modified by the GUI during the game.

For example, the action “Forward” has a 7-unit execution time, meaning that if the time is set to default (f=100), the action will take 0.07 second to be executed.

On the server, this time is managed by various clocks. One for the game globally and one for each player, that holds the time when a player started an action. This clock is checked against the global clock to know if a player can do an action.

```c
static bool is_client_ready(const server_t *serv, client_t *client)
{
    struct timeval current_time;
    double ready_time = timeval_get_milliseconds(&client->last_cmd_time)
        + (((double) client->cmd_duration / serv->freq) * 1000.0);

    if (client->state == GRAPHICAL)
        return (true);
    gettimeofday(&current_time, NULL);
    return (ready_time < timeval_get_milliseconds(&current_time));
}
```

Player Reproduction

A player is allowed to reproduce thanks to the “fork” command. This command will create an egg on the map, which then allows a new player to connect to the game.

The servers keep a list with all the eggs, allowing it to correctly position the new player connecting through the egg.

```c
struct egg_s {
    int x;
    int y;
    team_t *team;
    int number;
};
```

Ejection

A player can eject all other players from a shared unit of terrain by pushing them in the direction it is looking. When a client sends the eject command to the server, all the clients that are sharing the tile will receive the following line:

“eject: K \ n”

where K is the direction of the tile where the pushed player is coming from.

When a player ejects all other beings from the unit of terrain, it also destroys any eggs played on that unit.

```
bool ai_eject(server_t *serv, client_t *cli, UNUSED const char *obj)
{
    int len_team = list_get_size(serv->teams);
    int len_eggs = 0;
    team_t *tteam = NULL;

    sub_eject(serv, cli);
    for (int i = 0; i != len_team; i++) {
        tteam = list_get_elem_at_position(serv->teams, i);
        if (tteam == NULL)
            continue;
        len_eggs = list_get_size(tteam->eggs);
        for (int y = 0; y != len_eggs; y++)
            eject_destroy_eggs(serv, cli, tteam, &((ivect2D_t){i, y}));
    }
    cli->cmd_duration = 7;
    gettimeofday(&cli->last_cmd_time, NULL);
    return true;
}
```

GUI

The graphical user interface (GUI) allows a user to watch the progression of the game. It also allows the user to send various commands to the server to retrieve data, such as the map size, content of a tile, player position etc …

The GUI was made in C++, in 3D, using the RayLib library.

Server command

The GUI parses the commands sent by the server and then calls the correct function using a queue to store the commands sent by the server.

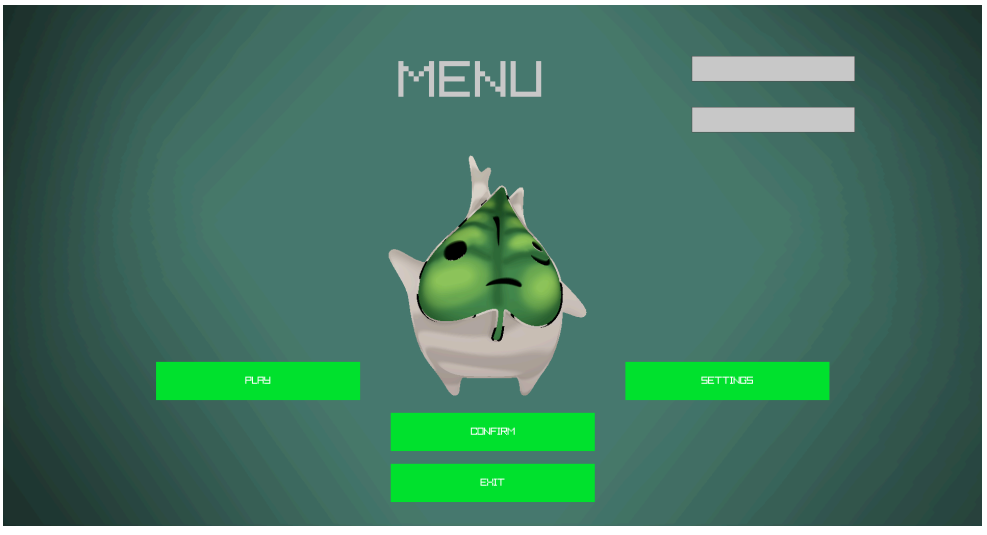
Scene management

The GUI uses an enum to determine what's the current scene. It's used to display different parts of the game

```
namespace Zappy {
    enum Scene {
        MENU,
        GAME,
        SETTINGS,
    };
};
```

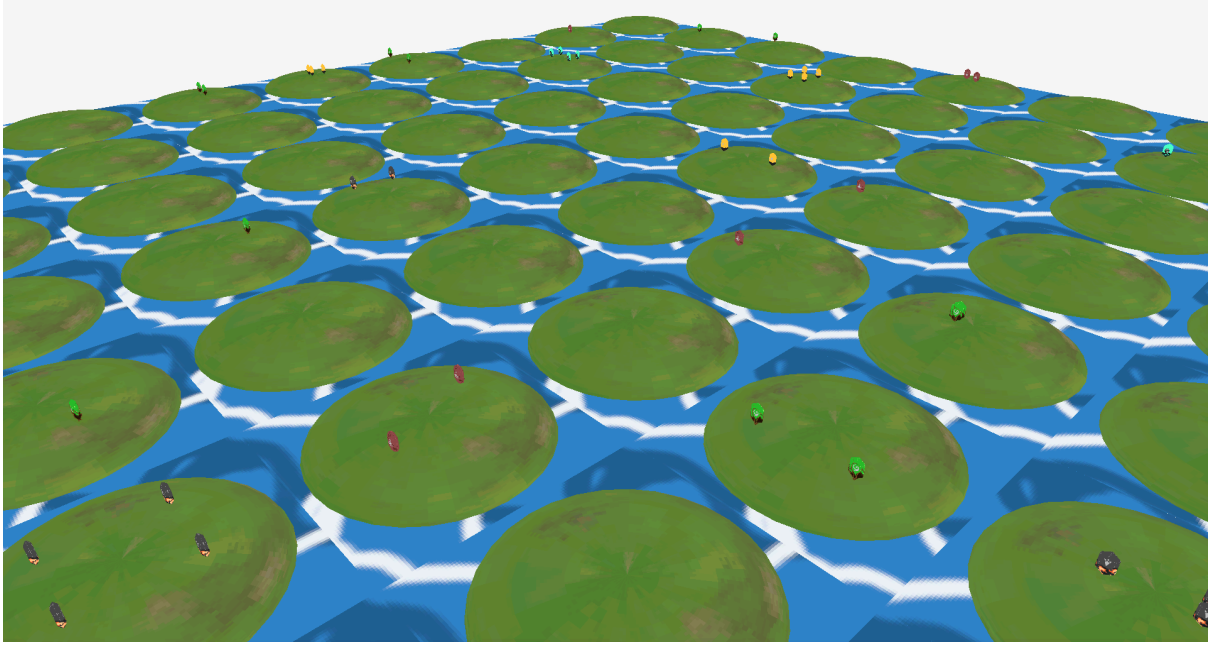
Menu

The menu allows the user to enter the server ip and port, as well as different settings such as the resolution, and the music volume.



Game Scene

The game scene is the main scene of the GUI. It's used to display the game in progress. We can find the cell, represented by islands. There are also the stones and the players.



The items are represented by the little rubies, each of them representing a specific resource.

```
void Zappy::Bloc::setItems(std::vector<Zappy::items> items)
{
    for (size_t i = 0; i != items.size(); i++)
        addItem(items[i], 1);
}
```

AI

The AI is the part of this project that controls every player. The goal of the AI is to win the game, meaning having 6 players on the same team reach level 8.

To do so the AI uses the different game mechanics as seen previously.

The AI was developed in python to allow an easier, and more permissive developpement.

Server connection

The AI uses a class to handle the connection to the server

```python
class Server:
    def __init__(self, hostname, port):
            self.hostname = hostname
            self.port = port
            self.socket = None
            self.selectors = selectors.DefaultSelector()
```

Player

The AI also has a player class.

This class is used to manage the inventory, as well as many other things. It stores the data locally to avoid having to send commands to the server and therefore losing time.

```python
class Player:
    def __init__(self, team_name: str) -> None:
        self.team = team_name
        self.logged = False
        self.data_to_send = f"{team_name}\n"
        self.inventory = {}
        self.map = []
        self.look_arround: str = ""
        self.level = 1
        self.step = 8
        self.action = []
        self.object_to_go = ""
        self.ready_to_level_up = False
        self.broadcast_receive = ""
        self.player_incantation = 1
        self.team_slot = 0
        self.verbose = False
        self.can_fork = True
        self.client_id = uuid.uuid4()
```

Algorithm

The algorithm part of the AI is responsible for all the movements and actions of a player. The algorithm is separated in multiple steps, with each one of them corresponding to a specific state of a player.

```python
def take_action(self):
        if self.step == 0:
            if self.action:
                self.data_to_send = self.action[0]
                self.action = self.action[1:]
            else:
                self.data_to_send = "Inventory\n"
                self.step += 1
            print_verbose(self.verbose, f"[SEND] {self.data_to_send}")
        elif self.step == 1:
            if (self.incantation_possible()):
                if self.player_incantation < PLAYER_MANDATORY[self.level - 1]:
                    message = f"{self.team};incantation;{self.level};{self.client_id}"
                    self.data_to_send = f"Broadcast {message}\n"
                print_verbose(self.verbose, f"[SEND] {self.data_to_send}")
                self.player_incantation = 1
                self.ready_to_level_up = True
                self.step = 4
            else:
                self.step += 1
        elif self.step == 2:
            self.data_to_send = "Look\n"
            self.step += 1
```