

ISLAND DEFENDERS

Kacper Lutomski

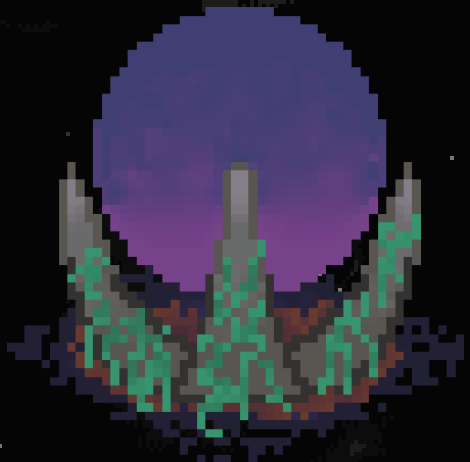
02 Stycznia 2023

Zasady gry



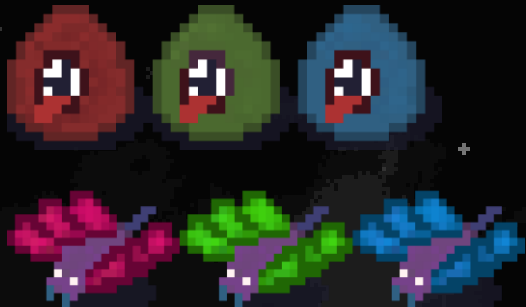
- Island Defenders to gra typu **Tower Defense**, gdzie ze wszystkich stron mapy nacierają stworki, a zadaniem gracza jest obrona znajdującego się na samym środku Dark Altaru.

Zasady Gry – Opis jednostek



Dark Altar

Znajduje się na środku mapy. Posiada 5 żyć. Gdy zostanie zaatakowany przez przeciwnika, traci jedno życie oraz zabija atakującą go jednostkę. Jeśli jego HP spadnie do zera, gra się kończy.



Przeciwnicy

Spawnią się na obrzeżach mapy i podążają w kierunku jej środka. Z biegiem gry rośnie ich HP oraz prędkość ataku/poruszania. Jeśli w zasięgu jednej kratki znajdzie się jakiegokolwiek budynek (Dark Altar, wieżyczka, kopalnia), atakują, zadając jeden punkt obrażeń. Za unicestwienie jednego wroga, gracz otrzymuje 5 kryształów.



Podstawowa wieża

Ma 4 życia. Zadaje 1 punkt obrażeń najbliższemu wrogowi w promieniu 3 kratek.



Magiczna wieża

Ma 3 życia. Jednorazowo unieruchamia wroga na 3 sekundy.



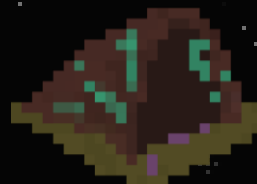
Ognista wieża

Ma 2 życia. Zadaje 2 punkty obrażeń (kosztem wyższej ceny i mniejszej ilości HP).



Lodowa wieża

Ma 3 życia. Spowalnia ruchy wroga o połowę.



Kopalnia

Ma 3 życia. Generuje 20 kryształów co 5 sekund. Może znajdować się jedynie na specjalnie wyznaczonych polach.

Struktura Kodu

- ↺ AssetManager.cpp
- ↺ Entity.cpp
- ↺ Game.cpp
- ↺ Hud.cpp
- ↺ main.cpp
- ↺ Particle.cpp
- ↺ Projectile.cpp
- ↺ Settings.cpp
- ↺ World.cpp

Gra jest podzielona na 7 plików, obejmujących **7 głównych klas + 2 Podklasy** oraz 2 dodatkowe pliki (main.cpp oraz Settings.cpp, w którym znajdują się funkcje odpowiedzialne za tworzenie i wczytywanie configu oraz zapisywanie oraz odczytywanie stanu gry z pliku).

Klasa 'Game'

Zmienne

```
class Game
{
private:
    // Const Variables & Enums
    sf::Clock clock;
    sf::Event sfEvent;
    bool fullscreenMode;
    AssetManager am;
    Hud *hud;
    std::vector<Particle> particles;
    std::vector<turretParticle> turretParticles;
    std::vector<Projectile> projectiles;

    // Gameplay Variables
    double deltaTime;

    bool debugMode;
    double screenTransition[4] = {0}; // {isActive [bool], value - coordinate [int], transition state (1-out, 2-waiting, 3-in) [in], end GameState [int]}
```

```
// Game states
enum gameStates
{
    MAINMENU,
    NEWGAME,
    GAME,
    SCORE,
    CREDITS
};
```

```
public:
    Game(int width, int height, float tileScale, bool fullScreenMode);
    virtual ~Game();

    // Public Variables
    sf::RenderWindow *window;
    float tileSize, tileScale = 3.f;
    long double timePassed = 0;
    double spawnTimer = 0;
    int screenWidth, screenHeight;
    int mapXOffset, mapYOffset;
    int gameState = MAINMENU;
    int selectedItem = 0;
    bool isPaused = false;
    int hoveredTileX = -1, hoveredTileY = -1;
    int selectedTileX = -1, selectedTileY = -1;
    bool buildMode = false;

    bool isStarted = false;
    int hearts = 5;
    int crystals = 1000;
    int crystalsEarned = 0;
    int monstersKilled = 0;
```

Najważniejsze zmienne:

- **gameState** – kontroluje w jakiej „fazie” znajduje się gra (MAINMENU, NEWGAME, GAME, SCORE, CREDITS).
- **isPaused** – jeśli równa się true to znaczy że gra jest zapauzowana, a wszystkie obliczenia są zatrzymane.
- Vektory **particles**, **turretParticles** oraz **projectiles** - w nich przetrzymywane są cząsteczki oraz pociski, które następnie są rysowane na ekran.

Klasa 'Game'

Metody prywatne

```
// Game Functions
void createWindow();

// Update Functions
void mouseTohoveredTile();
void callNewWave();
sf::Vector2i randomizeSpawnTile();

// Draw Functions
void drawMap();
void drawEntities();
void drawParticles();
void drawProjectiles();
void drawDebugInfo();
void drawScreenTransition();
```

createWindow() – tworzy okno gry o podanych parametrach – szerokości, wysokości oraz zmiennej isFullScreen decydującej czy gra ma otworzyć się w oknie czy w trybie pełnoekranowym.

mouseTohoveredTile() – przelicza pozycję myszy na ekranie na współrzędne kratki na mapie.

drawMap() – wyświetla mapę.

drawEntities() – rysuje jednostki na mapie.

drawParticles() – rysuje animowane cząsteczki.

drawProjectiles() – rysuje pociski.

drawDebugInfo() – wypisuje dodatkowe informacje o grze (podczas gdy użytkownik trzyma przycisk SHIFT).

drawScreenTransition() – tworzy animowane przejście między różnymi gameState'ami.

Klasa 'Game'

Funckja mouseToHoveredTile() i callNewWave()

```
// Calculates hovered tile.
void Game::mouseToHoveredTile()
{
    int x1 = sf::Mouse::getPosition(*window).x - mapXOffset;
    int y1 = (sf::Mouse::getPosition(*window).y - mapYOffset) * -2;
    double xr = cos(M_PI / 4) * x1 - sin(M_PI / 4) * y1;
    double yr = sin(M_PI / 4) * x1 + cos(M_PI / 4) * y1;
    double diag = tileSize * sqrt(2);
    hoveredTileX = floor(xr / diag);
    hoveredTileY = floor(yr * -1 / diag);
}
```

mouseToHoveredTile() – pobiera pozycję myszki z ekranu, a następnie oblicza współrzędne kratki, na której się ona znajduje.

```
// Runs every 1 second.
if (spawnTimer > 2 - int(monstersKilled / 20) / 10)
{
    spawnTimer = 0;

    // Spawns new wave of enemies.

    std::string enemyTypes[] = {"slime", "wasp"};
    std::string enemyColors[] = {"red", "green", "blue"};
    std::string enemyType = enemyTypes[rand() % 2];
    sf::Vector2i coords = randomizeSpawnTile();
    Enemy *r_wasp = new Enemy();
    if (enemyType == "wasp")
        r_wasp->createEntity(coords.x, coords.y, enemyType, "wasp_" + enemyColors[rand() % 3], 5, 7, 5 + world->difficulty + int(monstersKilled / 10), world);
    else
        r_wasp->createEntity(coords.x, coords.y, enemyType, "slime_" + enemyColors[rand() % 3], 9, 3, 7 + world->difficulty + int(monstersKilled / 10), world);
    world->entities[coords.x * MAPSIZE + coords.y] = r_wasp;
    r_wasp->setTimeToNextMove(double(rand() % 50 / 100));
    r_wasp->findPath();
    // sf::Thread thread(std::bind(&Entity::findPath, &(*r_wasp)));
    // thread.launch();
}
```

callNewWave() – co 2 sekundy (czas zmniejszający się w zależności od zabitych stworków) tworzy nowego, losowego przeciwnika na obrzeżach mapy.

Klasa 'Game'

Metody Publiczne

```
// Main Functions
void handleEvents();
void update();
void draw();
void run();

// Public Draw Functions
sf::Sprite drawSprite(float x, float y, std::string spriteName, float scaleX = 1.f, float scaleY = 1.f, bool draw = true, bool reversed = false);
sf::Text drawText(int x, int y, std::string text, int size, sf::Color color = sf::Color::White, bool centered = false, std::string font = "pixelmix", bool draw = true);
void addParticle(Particle particle);
void addTurretParticle(Particle particle, int x, int y);
void destroyTurretParticle(int x, int y);
void addProjectile(Projectile projectile);

// Public GameState functions
void changeGameState(int newGameState);

bool checkIfValidTileSelected();
bool checkIfValidTileHovered();
void clearParticlesAndProjectiles();
};
```

```
void Game::run()
{
    while (window->isOpen())
    {
        handleEvents();
        update();
        draw();
    }
}
```

run() – zawiera w sobie pętlę while, która jest główną pętlą gry.

handleEvents() – zajmuje się wykonywaniem operacji, gdy wykryje interakcje użytkownika (kliknięcie myszki, wciśnięcie przycisku).

update() – wywołuje pomniejsze funkcje aktualizujące jednostki, cząsteczki oraz pociski.

draw() – wywołuje pomniejsze funkcje rysujące poszczególne elementy na ekranie w zależności od wartości gameState

drawSprite() – funkcja rysująca sprite o danej teksturze na danych współrzędnych o danej skali.

drawText() - funkcja rysująca tekst na danych współrzędnych o danej wielkości.

Klasa 'AssetManager'

```
AssetManager::AssetManager()
{
    // Loads hud sprites.
    addSprite("bg", true);
    addSprite("cursor", true);
    addSprite("info_hud", true);
    addSprite("build_hud", true);
    addSprite("build_hud_selected", true);
    addSprite("build_hud_disabled", true);
    addSprite("overlay", true);
    addSprite("gem", true);
    addSprite("heart", true);

    // Loads building sprites
    addSprite("main_base", true);
    addSprite("mine", true);
}
```

Zajmuje się wczytywaniem tekstur oraz czcionek, które następnie przechowuje w odpowiednich mapach. Funkcje **getSprite()** oraz **getFont()** zwracają wskaźniki do odpowiednich sprite'ów/czcionek.

```
class AssetManager
{
private:
    sf::Texture txt;
    std::map<std::string, sf::Texture> textures;
    std::map<std::string, sf::Sprite> sprites;
    std::map<std::string, sf::Font> fonts;

public:
    AssetManager();
    virtual ~AssetManager();

    // Functions
    void addSprite(std::string path, bool full = true, std::string variant = "", int x = 0, int y = 0, int width = 0, int height = 0);
    void addParticleTexture();
    sf::Sprite *getSprite(std::string id);
    sf::Font *getFont(std::string fontName);
};
```

Klasa 'World'

Przechowuje dwie główne tablice:

- **tilemap**[][] odpowiedzialną za przechowywanie id tekstur poszczególnych kratek.
- **entities**[] – przechowującą wskaźniki do obiektów typu Entity.

Funkcja **createNewWorld()** generuje nową losową mapę.

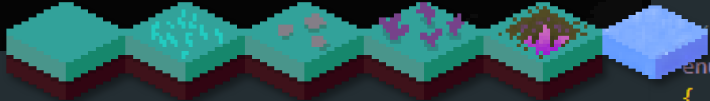
funkcja **placeNewBuilding()** tworzy nowy obiekt typu Building i umieszcza wskaźnik do niego w w entities[].

funkcja **getEntity(x, y)** zwraca wskaźnik do obiektu znajdującego się na korydnatach (x, y) z tablicy entities[].

Funkcja **destroyEntity()** usuwa obiekt i wskaźnik z tablicy entities[].

```
class World
{
private:
public:
    Game *game;
    Entity *entities[MAPSIZE * MAPSIZE];
    int tilemap[MAPSIZE][MAPSIZE];
    World(Game *game);
    ~World();
    void createNewWorld();
    Entity *getEntity(int x, int y);
    bool placeNewBuilding(int x, int y, std::string type, std::string spriteName, int xOffset, int yOffset, int health);
    void destroyEntity(int x, int y);

    int difficulty = 0;
};
```



```
enum tiles
{
    REGULARTILE,
    GRASSTILE,
    ROCKTILE,
    PLANTTILE,
    MINETILE,
    WATERTILE,
    TREETILE,
};
```

Klasa 'World'

Funkcja createNewWorld()

```
void World::createNewWorld()
{
    srand((unsigned)time(NULL));

    for (int i = 0; i < MAPSIZE * MAPSIZE; i++)
        entities[i] = nullptr; // NULLPTR

    // Generates tilemap.
    for (int i = 0; i < MAPSIZE; i++)
        for (int j = 0; j < MAPSIZE; j++)
            this->tilemap[i][j] = 0;

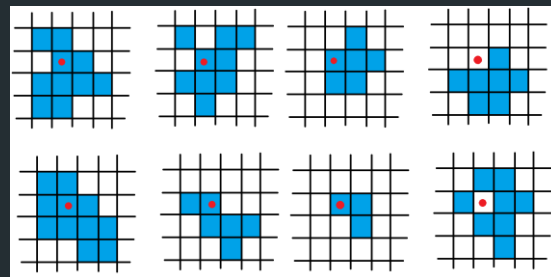
    // Adds tiles with grass and rocks.
    for (int i = 40 + rand() % 80; i > 0; i--)
    {
        this->tilemap[rand() % MAPSIZE][rand() % MAPSIZE] = rand() % 3 + 1; // Sets array to REGULARTILE, GRASSTILE, ROCKTILE or PLANTTILE
    }

    for (int i = 0; i < 8; i++)
    {
        this->tilemap[rand() % MAPSIZE][rand() % MAPSIZE] = MINETILE; // Sets array to REGULARTILE, GRASSTILE, ROCKTILE, PLANTTILE or MINETILE.
    }
}
```

Funkcja **createNewWorld()** wypełnia losowymi wartościami tablicę tilemap[[]]. Następnie generuje 8 pól o losowych współrzędnych, na których gracz będzie mógł umieścić kopalnię. Umieszcza losową liczbę jezior (nie mniejszą niż 4 i nie większą niż 8), korzystając z predefiniowanych schematów. Na koniec w losowych miejscach dodaje TREETILE oraz tworzy drzewa - obiekty typu Building.

```
// Adds water pools in available spaces.
for (int i = 4 + rand() % 5; i > 0; i--) // Amount of pools
{
    int x = rand() % MAPSIZE;
    int y = rand() % MAPSIZE;
    int direction = rand() % 4; // top, right, bottom, left
    std::string waterSchematics[8] = {
        "1100011011111100",
        "1011011011100100",
        "0010011101100000",
        "0000001011110110",
        "1100111011110011",
        "0000110001110010",
        "0000011000100000",
        "0110101101100010";
    int schematic = rand() % 8;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
        {
            if (waterSchematics[schematic][i * 4 + j] == '1')
                this->tilemap[std::max(std::min(x - 1 + i, MAPSIZE - 1), 0)][std::max(std::min(y - 1 + j, MAPSIZE - 1), 0)] = WATERTILE;
        }
}
```

Na rysunku poniżej znajdują się wzorce do generacji jezior.
Każda z kombinacji 0 i 1 odpowiada kolejnemu ułożeniu watertile'i.



```
for (int i = 7 + rand() % 5; i > 0; i--)
{
    int x, y;
    do
    {
        x = rand() % MAPSIZE;
        y = rand() % MAPSIZE;
    } while (getEntity(x, y) || tilemap[x][y] == WATERTILE || tilemap[x][y] == MINETILE);

    Building *tree = new Building();
    tree->createEntity(x, y, "tree", "tree", 8, 15, 9999, this);
    entities[x * MAPSIZE + y] = tree;
    tilemap[x][y] = TREETILE;
}
```

Klasa 'Entity'

```
class Entity
{
protected:
    int x, y;
    int xOffset, yOffset;
    int damage;
    int health;
    double timeToNextMove = 0;
    std::string type;
    std::string spriteName;
    World *world;
    int actionTimeMultiplier = 1;
    bool wasStunned = false;

public:
    Entity();
    ~Entity();
    void createEntity(int x, int y, std::string type, std::string spriteName, int xOffset, int yOffset, int health, World *world);
    std::string getSpriteName();
    std::string getType();
    virtual void performAction(double time);
    int getX();
    int getY();
    int getXOffset();
    int getYOffset();
    World *getWorld();
    virtual float getMoveX();
    virtual float getMoveY();
    int getHealth();
    virtual double getTimeToNextMove();
    virtual int getIsMoving();
    virtual int getDirection();
    virtual bool checkIfPathEmpty();
    virtual void findPath();
    void decreaseHealth(int dmg);
    void setTimeToNextMove(double time);
    void setActionMultiplier(int multiplier);
    virtual void setIsMoving(bool set);
    void stun();
};
```

Zajmuje się przechowywaniem informacji o każdym obiekcie znajdującym się na mapie. Jest szkieletem do dwóch podklas – **Enemy** oraz **Building**, z tego powodu zawiera dużo metod wirtualnych.

Klasa 'Entity'

Podklasy 'Building' i 'Enemy'



```
class Building : public Entity
{
private:
public:
    Building();
    ~Building();
    void createProjectile(int target_x, int target_y, Entity *enemy);
    void performAction(double time);
};
```

Klasa **Building** zawiera funkcję odpowiedzialną za tworzenie obiektów typu Projectile (pocisków wystrzeliwanych w przeciwników) oraz funkcję performAction() odpowiedzialną za atakowanie wrogów/dodawanie kryształów (w przypadku kopalni).



```
class Enemy : public Entity
{
private:
    bool isMoving = false;

    int direction = 0; // 0 - gora, 1 - prawo, 2 - dol, 3 - lewo
    float moveX = 0.f, moveY = 0.f;
    std::vector<sf::Vector2i> pathBFS;
    bool checkForNearBuildings();

public:
    Enemy();

    ~Enemy();
    void performAction(double time);
    float getMoveX();
    float getMoveY();
    int getDirection();
    int getIsMoving();
    void findPath();
    bool checkIfPathEmpty();
    void setIsMoving(bool set);
};
```

Klasa **Enemy** zawiera dodatkowe funkcje odpowiadające za płynny ruch przeciwników między kratkami, a także funkcję **findPath()**, wykorzystującą algorytm Breadth-First-Search do pathfindingu.

Klasa 'Entity'

Podklasa 'Building' – funkcja performAction()

```
void Building::performAction(double deltaTime)
{
    if (type == "base" || type == "wall" || type == "tree")
        return;

    timeToNextMove += deltaTime;

    if (type == "mine")
    {
        if (timeToNextMove > 5)
        {
            float scale = 1.f;
            int mapX = world->game->tileSize * x - world->game->tileSize * y - world->game->tileSize + world->game->mapOffset + 0.5 * world->game->tileSize * world->game->tileScale - world->game->tileScale * 3 * scale;
            int mapY = (world->game->tileSize * y + world->game->tileSize * x) / 2 + world->game->mapOffset - world->game->tileScale * 15 * scale;
            Particle particle(mapX, mapY, "particles/gem", 7, scale);
            world->game->addParticle(particle);
            world->game->crystals += 20;
            world->game->crystalsFarmed += 20;
            timeToNextMove = 0;
        }
        return;
    }
}
```

```
if (timeToNextMove > 0.3)
{
    std::vector<Entity*> enemies;

    // Puts nearest enemies into a vector.
    for (int i = 0; i < 3; i++)
    {
        for (int target_y = std::max(y - 1 - i, 0); target_y <= std::min(y + 1 + i, MAPSIZE - 1); target_y++) // height
            for (int target_x = std::max(x - 1 - i, 0); target_x <= std::min(x + 1 + i, MAPSIZE - 1); target_x++) // width
            {
                if (target_x == x && target_y == y)
                    continue;
                if (world->getEntity(target_x, target_y))
                {
                    if (world->getEntity(target_x, target_y)->getType() == "wasp" || world->getEntity(target_x, target_y)->getType() == "slime")
                    {
                        enemies.push_back(world->getEntity(target_x, target_y));
                    }
                }
            }
        if (enemies.size() > 0)
            break;
    }

    // Searches for an enemy with the shortest distance.
    int shortestDistance = 100000000;
    Entity *closestEnemy = nullptr;
    for (auto enemy = begin(enemies); enemy != end(enemies); ++enemy)
    {
        double distance = (x - (*enemy)->getX()) * (x - (*enemy)->getX()) + (y - (*enemy)->getY()) * (y - (*enemy)->getY());
        if (distance < shortestDistance)
        {
            shortestDistance = distance;
            closestEnemy = *enemy;
        }
    }

    if (closestEnemy)
        createProjectile(closestEnemy->getX(), closestEnemy->getY(), closestEnemy);

    return;
}
}
```

Funkcja **performAction()** w obydwu klasach jest bardzo rozbudowana.

W klasie **Building** wykonuje ona kolejne operacje:

- Sprawdza czy budynek powinien w ogóle wykonywać akcję (czy nie jest drzewem lub altarem).
- Sprawdza czy budynek nie jest kopalnią – w tym przypadku przyznaje kryształy co każde 5 sekund.
- W tym momencie wiadomo, że budynek jest wieżyczką, więc rozpoczyna się szukanie najbliższej jednostki do zaatakowania.

Klasa 'Entity'

Podklasa 'Enemy' – funkcja performAction()

```
void Enemy::performAction(double deltaTime)
{
    timeToNextMove += deltaTime;

    double moveSpeed = 0;
    if (type == "wasp")
        moveSpeed = 0.5 * actionTimeMultiplier;
    else if (type == "slime")
        moveSpeed = 0.8 * actionTimeMultiplier;

    if (!isMoving)
    {
        if (timeToNextMove > moveSpeed)
            if (checkForNearBuildings())
            {
                for (int target_y = std::max(y - 1, 0); target_y <= std::min(y + 1, MAPSIZE - 1); target_y++) // height
                    for (int target_x = std::max(x - 1, 0); target_x <= std::min(x + 1, MAPSIZE - 1); target_x++) // width
                    {
                        if (target_x == x && target_y == y)
                            continue;
                        if (world->getEntity(target_x, target_y))
                        {
                            if (world->getEntity(target_x, target_y)->getType() == "turret" || world->getEntity(target_x, target_y)->getType() == "base" || world->getEntity(target_x, target_y)->getType() == "mine")
                            {
                                std::string targetType = world->getEntity(target_x, target_y)->getType();
                                world->getEntity(target_x, target_y)->decreaseHealth(1);

                                float scale = 1.5f;
                                int mapX = world->game->tileSize * target_x - world->game->tileSize * target_y - world->game->mapXOffset + 3 * world->game->tileScale * scale;
                                int mapY = (world->game->tileSize * target_y + world->game->tileSize * target_x) / 2 + world->game->mapYOffset - world->game->tileScale * scale - 4 * world->game->tileScale;
                                Particle particle(mapX, mapY, "particles/hit", 6, scale);
                                world->game->addParticle(particle);

                                timeToNextMove = 0;
                                if (targetType == "base")
                                {
                                    this->decreaseHealth(5);
                                }
                                return;
                            }
                        }
                        else
                            continue;
                    }
            }
        else
        {
            // Chooses most optimal direction.
            if (pathBFS.empty())
                return;
            // Plan path.
            int new_x = pathBFS.back().x, new_y = pathBFS.back().y;

            if (new_x < 0)
                direction = 1;
            else if (new_x < x)
                direction = 3;
            else if (new_y > y)
                direction = 2;
            else if (new_x > x)
                direction = 4;
            else
                direction = 0;

            // =====
            if (world->isEntity(new_x * MAPSIZE + new_y))
            {
                pathBFS.pop_back();
                world->isEntity(new_x * MAPSIZE + new_y) = world->isEntity(x * MAPSIZE + y);
                world->isEntity(x * MAPSIZE + y) = false;
                x = new_x;
                y = new_y;
                timeToNextMove = 0;
                isMoving = true;

                switch (direction)
                {
                    case 0:
                        moveX = -16.f;
                        moveY = 8.f;
                        break;
                    case 1:
                        moveX = -16.f;
                        moveY = -8.f;
                        break;
                    case 2:
                        moveX = 16.f;
                        moveY = -8.f;
                        break;
                    case 3:
                        moveX = 16.f;
                        moveY = 8.f;
                        break;
                }
            }
        }
    }
    else
    {
        deltaTime *= 4; // speed of animation
        switch (direction)
        {
            case 0:
                moveX += 16 * deltaTime;
                moveY -= 8 * deltaTime;
                break;
            case 1:
                moveX += 15 * deltaTime;
                moveY += 8 * deltaTime;
                break;
            case 2:
                moveX -= 16 * deltaTime;
                moveY += 8 * deltaTime;
                break;
            case 3:
                moveX -= 15 * deltaTime;
                moveY -= 8 * deltaTime;
                break;
        }
        if (timeToNextMove > 0.25)
        {
            timeToNextMove = 0;
            isMoving = false;
            moveX = 0.f;
            moveY = 0.f;
        }
    }
}
```

Natomiast w klasie **Enemy** wykonuje ona następujące operacje:

- Sprawdza czy obiekt jest w bezruchu.
- Jeśli tak, to:
 - Sprawdza czy w pobliżu 1 kratki znajduje się budynek
 - Jeśli tak to:
 - Atakuje go
 - Jeśli nie, to:
 - wykonuje ruch według ruchów zawartych w wektorze **PathBFS**
- Jeśli nie, to:
 - Dalej wykonuje ruch, aż do końca.

Klasa 'Entity'

Podklasa 'Enemy': funkcja findPath() - Pathfinding

```
void Enemy::findPath()
{
    dest_x = 11;
    dest_y = 11;
    std::vector<visitedTile> queue;
    std::vector<visitedTile> visited;
    int maxsteps = 5000;
    int currentX = x, currentY = y;
    visited.push_back(sf::Vector2i(currentX, currentY), sf::Vector2i(-1, -1));
    while (maxsteps > 0)
    {
        if (checkIfNotInVector(queue, sf::Vector2i(currentX + 1, currentY)) && currentX + 1 < MAPSIZE)
            if (world->tilemap[currentX + 1][currentY] != WATERTILE && world->tilemap[currentX + 1][currentY] != TREETILE)
                queue.push_back(sf::Vector2i(currentX + 1, currentY), sf::Vector2i(currentX, currentY));

        if (checkIfNotInVector(queue, sf::Vector2i(currentX - 1, currentY)) && currentX - 1 >= 0)
            if (world->tilemap[currentX - 1][currentY] != WATERTILE && world->tilemap[currentX - 1][currentY] != TREETILE)
                queue.push_back(sf::Vector2i(currentX - 1, currentY), sf::Vector2i(currentX, currentY));

        if (checkIfNotInVector(queue, sf::Vector2i(currentX, currentY + 1)) && currentY + 1 < MAPSIZE)
            if (world->tilemap[currentX][currentY + 1] != WATERTILE && world->tilemap[currentX][currentY + 1] != TREETILE)
                queue.push_back(sf::Vector2i(currentX, currentY + 1), sf::Vector2i(currentX, currentY));

        if (checkIfNotInVector(queue, sf::Vector2i(currentX, currentY - 1)) && currentY - 1 >= 0)
            if (world->tilemap[currentX][currentY - 1] != WATERTILE && world->tilemap[currentX][currentY - 1] != TREETILE)
                queue.push_back(sf::Vector2i(currentX, currentY - 1), sf::Vector2i(currentX, currentY));

        if (queue.size() == 0)
            break;

        currentX = queue.front().vertex.x;
        currentY = queue.front().vertex.y;
        visited.push_back(queue.front());

        queue.erase(queue.begin());
        if (currentX == dest_x && currentY == dest_y)
            break;

        maxsteps--;
    }
}
```

Funkcja **findPath()** jest implementacją algorytmu Breadth-First-Search. Tworzy 2 vectory (**queue** oraz **visited**), a następnie zapętlą się przez kolejne pola z queue. Gdy **currentX** i **currentY** będą obie równe 11, pętla zrywa się i poniższy skrypt odczytuje właściwą drogę, która zapisuje do vectora **PathBFS**:

```
while (getFromVector(visited, sf::Vector2i(currentX, currentY)).x != -1)
{
    pathBFS.push_back(sf::Vector2i(currentX, currentY));
    sf::Vector2i currentVector = getFromVector(visited, sf::Vector2i(currentX, currentY));
    currentX = currentVector.x;
    currentY = currentVector.y;
}

sf::Vector2i getFromVector(std::vector<visitedTile> visited, sf::Vector2i vertex)
{
    for (auto i = visited.begin(); i != visited.end(); ++i)
    {
        if ((*i).vertex == vertex)
            return (*i).origin;
    }
}

bool checkIfNotInVector(std::vector<visitedTile> queue, sf::Vector2i newVertex)
{
    for (auto tile = begin(queue); tile != end(queue); ++tile)
        if (tile->vertex == newVertex)
            return false;
    return true;
}
```


Klasa 'Particle'

```
class Particle
{
private:
    int x, y;
    double speed = 0.25, time = 0;
    float scale;
    int frame = 0, maxframe;
    std::string spriteName;

public:
    Particle(int x, int y, std::string spriteName, int maxframe, float scale);
    ~Particle();
    bool update(double deltaTime);
    sf::Vector2i getPosition();
    float getScale();
    std::string getSpriteName();
};
```

Klasa ta przechowuje informacje na temat cząsteczek takich jak wybuchy po zniszczeniu, animacje ataku wrogów czy animacje kryształów przy kopalniach. Główna funkcja, która wywołuje się co każdą klatkę, sprawdza czas wyświetlania animacji i w zależności od warunków, zmienia klatkę animacji na następną. Gdy wartość **frame** osiągnie wartość zmiennej **maxframe**, funkcja zwraca wartość **true**, a cząsteczka zostaje zniszczona.

```
bool Particle::update(double deltaTime)
{
    if (time > 0.05) // Animation speed.
    {
        if (frame == maxframe)
        {
            if (spriteName == "particles/fire" || spriteName == "particles/magic" || spriteName == "particles/poison" || spriteName == "particles/ice")
            {
                frame = 1;
                time = 0;
            }
            else
                return true;
        }
        else
        {
            frame++;
            time = 0;
        }
    }
    else
    {
        time += deltaTime;
        return false;
    }
}
```

Klasa 'Projectile'

```
class Projectile
{
private:
    double x, y, travelTime;
    std::string spriteName;
    double start_x,
           start_y,
           dest_x,
           dest_y;
    int enemy_x, enemy_y;
    World *world;

public:
    Projectile(int x, int y, int dest_x, int dest_y, std::string spriteName, Entity *entity);
    ~Projectile();
    bool update(double deltaTime);
    sf::Vector2f getPosition();
    std::string getSpriteName();
};
```

Klasa ta zajmuje się pociskami. Główna funkcja oblicza właściwą pozycję pocisku w danej klatce, a także sprawdza, czy pocisk nie osiągnął celu. Jeśli tak się stanie, pocisk nałoży na docelowego wroga odpowiednie efekty.

Instrukcja **if (travelTime > 5) return true;** zapobiega nieusuwaniu pocisków, które nie trafią w cel.

```
bool Projectile::update(double deltaTime)
{
    travelTime += deltaTime;
    float tx = dest_x - start_x;
    float ty = dest_y - start_y;
    float dist = sqrt(tx * tx + ty * ty);

    x += tx / dist * 1000 * deltaTime;
    y += ty / dist * 1000 * deltaTime;

    if (fabs(sqrt(pow((dest_x - x), 2) + pow((dest_y - y), 2))) < 20)
    {
        if (world->getEntity(enemy_x, enemy_y))
        {
            if (spriteName == "projectile_ice")
                world->getEntity(enemy_x, enemy_y)->setActionMultiplier(2);
            else if (spriteName == "projectile_magic")
            {
                world->getEntity(enemy_x, enemy_y)->stun();
            }
            else if (spriteName == "projectile_fire")
                world->getEntity(enemy_x, enemy_y)->decreaseHealth(2);
            else
                world->getEntity(enemy_x, enemy_y)->decreaseHealth(1);
        }
        return true;
    }

    if (travelTime > 5)
        return true;

    return false;
}
```

Klasa 'Hud'

```
class Hud
{
private:
    Game *game;
    struct hudElement
    {
        sf::FloatRect bounds;
        int id;
    };
    std::vector<hudElement> hudElements;
    void addHudElement(sf::FloatRect bounds, int id);
    int size, smallSize, offset;
    int difficultyChoice = 0; // 0 - easy, 1 - normal, 2 - hard
    const std::map<int, std::string> difficulties = {{0, "Easy"},
                                                    {1, "Normal"},
                                                    {2, "Hard"}};
    // int elementChoice = 0; // 0 - water, 1 - fire, 2 - ground, 3

public:
    Hud(Game *game);
    virtual ~Hud();

    void updateHudElements(int newGameState);
    void executeMenuAction();
    int checkMouse(sf::Vector2f);
    void drawMainMenu();
    void drawNewGameScreen();
    void drawGameHud(int wave);
    void drawScore();
    void drawCredits();
    void drawPauseOverlay();
};
```

Głównym zadaniem tej klasy jest wyświetlanie elementów interfejsu użytkownika, a także wykonywanie operacji w momencie, gdy użytkownik zdecyduje się z nim wejść w interakcję. Vector **hudElements** przechowuje pary prostokątów oraz odpowiednich id opcji. Gdy myszka znajdzie się w jednym z tych prostokątów możemy wywnioskować, że użytkownik właśnie najechał na opcję o odpowiednim id.

- Funkcja **updateHudElements()** aktualizuje wyżej opisane pary w vectorze hudElements.
- Funkcja **executeMenuAction()** wykonuje odpowiednie polecenia w zależności od kliniętej opcji.
- Funkcja **checkMouse()** sprawdza, w którym prostokącie znajduje się mysz (czyli która opcja jest aktualnie wybrana).

Pozostałe funkcje wyświetlają interfejs użytkownika w zależności od **gameState**.

Klasa 'Hud'

Funkcja drawGameHud()

```
void Hud::drawGameHud(int wave)
{
    // Draws crystals.
    game->drawSprite(game->tileScale * 8, game->tileScale * 8, "gem", 2 * game->tileScale, 2 * game->tileScale);
    game->drawText(game->tileScale * 8 + 12 * 2 * game->tileScale, game->tileScale * 9, std::to_string(game->crystals), game->tileScale * 8 * 2);

    // Draws hearts.
    for (int i = 0; i < game->hearts; i++)
        game->drawSprite(game->screenWidth - (game->tileScale * 8) - (2 * game->tileScale * 11) - (i * (game->tileScale * (4 + 2 * 11))), game->tileScale * 8, "heart", 2 * game->tileScale, 2 * game->tileScale);

    // Draws Info Panel.
    if (game->checkIfValidTileHovered())
    {
        // Selects item to display.
        if (game->world->entities[game->hoveredTileX * MAPSIZE + game->hoveredTileY])
        {
            std::string type = game->world->entities[game->hoveredTileX * MAPSIZE + game->hoveredTileY]->getType();
            if (type == "base" || type == "turret" || type == "mine" || type == "tree")
            {
                // Draws Back Panel.
                int ix = game->tileScale * 8;
                int iy = game->screenHeight - (game->tileScale * (8 + 1.5 * 32));
                game->drawSprite(ix, iy, "info_hud", 1.5 * game->tileScale, 1.5 * game->tileScale);

                // Draws Entity Info.
                Entity *ent = game->world->getEntity(game->hoveredTileX, game->hoveredTileY);
                if (type == "base")--
                else if (type == "turret")--
                else if (type == "mine")
                {
                    game->drawSprite(ix + 6 * game->tileScale, iy + 10 * game->tileScale, "mine", 1.5 * game->tileScale, 1.5 * game->tileScale);
                    game->drawText(ix + (8 + 40) * game->tileScale, iy + (6 + 1.5 * game->tileScale) * 0.75, "MINE", 0 * game->tileScale);
                    game->drawText(ix + (8 + 40) * game->tileScale, iy + (6 + 1.5 * game->tileScale) * 1.75, "Mines 20 crystals every 5 seconds.", 6 * game->tileScale);
                    game->drawText(ix + (8 + 40) * game->tileScale, iy + (6 + 1.5 * game->tileScale) * 2.75, "Next delivery in " + std::to_string(5 - int(ent->getTimeToNextMove())) + " seconds...", 6 * game->tileScale);
                    game->drawText(ix + (8 + 40) * game->tileScale, iy + (6 + 1.5 * game->tileScale) * 3.75, "HP: " + std::to_string(ent->getHealth()) + "/" + std::to_string(MINEHEALTH), 6 * game->tileScale);
                }
                else if (type == "tree")--
            }
        }
    }

    // Draws Build Menu.
    if (game->checkIfValidTileSelected() && game->buildMode && game->world->tilemap[game->selectedTileX][game->selectedTileY] != MATERTILE && !game->world->entities[game->selectedTileX * MAPSIZE + game->selectedTileY])
    {
        int x = game->tileSize * game->selectedTileX - game->tileSize * game->selectedTileY - game->tileSize + game->mapXOffset;
        int y = (game->tileSize * game->selectedTileY + game->tileSize * game->selectedTileX) / 2 + game->mapYOffset;
        x += 16 * game->tileScale - 184 / 2 * game->tileScale;
        if (game->selectedTileX > MAPSIZE - 5 && game->selectedTileY > MAPSIZE - 5)
            y -= (40 + 2) * game->tileScale;
        else
            y += (16 + 4) * game->tileScale;
        game->drawSprite(x, y, "build_hud", game->tileScale, game->tileScale);

        // Highlights hovered option.
        if (game->selectedItem > 0 && game->selectedItem < 6)
        {
            game->drawSprite(x + (4 + (game->selectedItem - 1) * 24 + (game->selectedItem - 1) * 6) * game->tileScale, y + 4 * game->tileScale, "build_hud_selected", game->tileScale, game->tileScale);

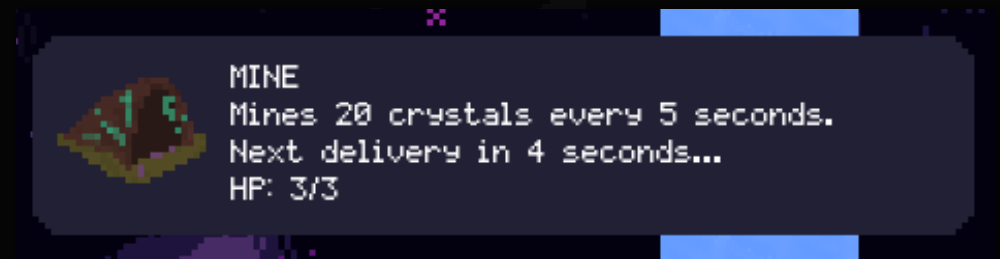
            // Shows price.
            int price = 0;
            switch (game->selectedItem)--
            if (game->crystals >= price)
                game->drawText(x + 24 * game->tileScale, y + 31 * game->tileScale, std::to_string(price), 5 * game->tileScale);
            else
                game->drawText(x + 24 * game->tileScale, y + 31 * game->tileScale, std::to_string(price), 5 * game->tileScale, sf::Color::Red);
        }

        if (game->world->tilemap[game->selectedTileX][game->selectedTileY] != MINETILE)
            game->drawSprite(x + (4 + 4 * 24 + 4 * 6) * game->tileScale, y + 4 * game->tileScale, "build_hud_disabled", game->tileScale, game->tileScale);

        // Draws buildings relatively to build hud.
        game->drawSprite(x + (7 + 0 * 30) * game->tileScale, y + 5 * game->tileScale, "turret", game->tileScale, game->tileScale);
        game->drawSprite(x + (7 + 1 * 30) * game->tileScale, y + 5 * game->tileScale, "turret_ice", game->tileScale, game->tileScale);
        game->drawSprite(x + (7 + 2 * 30) * game->tileScale, y + 5 * game->tileScale, "turret_fire", game->tileScale, game->tileScale);
        game->drawSprite(x + (7 + 3 * 30) * game->tileScale, y + 5 * game->tileScale, "turret_magic", game->tileScale, game->tileScale);
        game->drawSprite(x + ((5 + 4 * 30) - 1) * game->tileScale, y + 7 * game->tileScale, "mine", game->tileScale, game->tileScale);
    }
}
```

Funkcja **drawGameHud()** kolejno:

- Rysuje ilość posiadanych przez gracza kryształów.
- Rysuje serduszka odpowiadające ilości pozostałych żyć.
- Sprawdza czy gracz nie najechał myszką na obiekt w grze - wtedy wyświetla panel informacyjny.
- Sprawdza czy gracz nie jest w trybie budowania – wtedy wyświetla menu z dostępnymi budynkami.



Zapisywanie i odczyt z pliku

```
// Creates save.cfg file.
void createSaveFile(int tilemap[MAPSIZE][MAPSIZE], Entity *entities[MAPSIZE * MAPSIZE], long double timePassed, int hearts, int crystals, int monstersKilled, int difficulty)
{
    std::string gameSave, stilemap = "map:", sentities;

    for (int i = 0; i < MAPSIZE; i++)
        for (int j = 0; j < MAPSIZE; j++)
            stilemap = stilemap + std::to_string(tilemap[i][j]);

    for (int i = 0; i < MAPSIZE; i++)
        for (int j = 0; j < MAPSIZE; j++)
            if (entities[i * MAPSIZE + j])
            {
                Entity *ent = entities[i * MAPSIZE + j];
                std::string entity = "e:" + std::to_string(ent->getX()) + " " + std::to_string(ent->getY()) + " " + ent->getType() + " " + ent->getSpriteName() + " " + std::to_string(ent->getXOff());
                sentities = sentities + entity + "\n";
            }

    gameSave = "#Map\n" + stilemap + "\n\n#Entities\n" + sentities + "\n\n#Game Variables\ntime:" + std::to_string(int(timePassed)) + "\nhearts:" + std::to_string(hearts) + "\ncrystals:" + std::to_string(crystals) + "\nmonsterskilled:" + std::to_string(monstersKilled) + "\ndifficulty:" + std::to_string(difficulty);

    ofstream file("save.cfg");
    file << gameSave;
    file.close();
}
```

```
1 #Map
2 map:000300000000003005510000003100011000032051
3
4 #Entities
5 e:2 21 slime slime_blue 8 9 3
6 e:3 10 tree tree 9999 8 15
7 e:6 0 wasp wasp_blue 6 5 7
8 e:7 9 tree tree 9999 8 15
9 e:8 8 turret turret_fire 2 7 8
10 e:9 11 turret turret_magic 3 7 8
11 e:9 16 tree tree 9999 8 15
12 e:9 20 tree tree 9999 8 15
13 e:10 10 base main_base 9999 -11 25
14 e:10 11 base 9999 0 0
15 e:11 5 wasp wasp_green 8 5 7
16 e:12 10 turret turret_magic 3 7 8
17 e:12 12 turret turret_fire 2 7 8
18 e:13 13 mine mine 3 3 2
19 e:14 3 slime slime_red 12 9 3
20 e:16 3 wasp wasp_green 6 5 7
21 e:17 3 wasp wasp_blue 11 5 7
22 e:21 5 tree tree 9999 8 15
23 e:23 4 tree tree 9999 8 15
24 e:23 6 tree tree 9999 8 15
25
26
27 #Game Variables
28 time:7
29 hearts:5
30 crystals:165
31 monsterskilled:2
32 difficulty:1
33
```

Przykładowa zawartość pliku save.cfg

Zapisywanie i odczyt z pliku

```
// Loads save from file.
bool loadSaveFile(World *world, long double *timePassed, int *hearts, int *crystals, int *monstersKilled)
{
    string text;
    ifstream file("save.cfg");

    // Check if file exists.
    if (file.fail())
    {
        return false;
    }

    // Clears entities array.
    for (int i = 0; i < MAPSIZE * MAPSIZE; i++)
    {
        if (world->entities[i])
        {
            delete world->entities[i];
            world->entities[i] = nullptr;
        }
    }

    // Loops through all lines of the file.
    while (getline(file, text))
    {
        // Ignores blank lines and comments starting with '#' symbol.
        if (text[0] != '#' && text != "")
        {
            if (text.rfind("e:", 0)
            {
                int propertyIndex = 0;
                std::string entityProperties[7];
                text.replace(0, 2, "");

                for (int i = 0; i < text.length(); i++)
                {
                    if (text[i] != ' ')
                        entityProperties[propertyIndex] = entityProperties[propertyIndex] + text[i];
                    else
                        propertyIndex++;
                }

                int x = stoi(entityProperties[0]), y = stoi(entityProperties[1]);
                if (entityProperties[2] == "wasp" || entityProperties[2] == "slime")
                {
                    Enemy *ent = new Enemy();
                    ent->createEntity(stoi(entityProperties[0]), stoi(entityProperties[1]), entityProperties[2], entityProperties[3], stoi(entityProperties[5]), stoi(entityProperties[6]), stoi(entityProperties[4]), world);
                    world->entities[x * MAPSIZE + y] = ent;
                    ent->setTimeToNextMove(double(rand() % 50 / 100));
                }
                else
                {
                    Building *ent = new Building();
                    ent->createEntity(stoi(entityProperties[0]), stoi(entityProperties[1]), entityProperties[2], entityProperties[3], stoi(entityProperties[5]), stoi(entityProperties[6]), stoi(entityProperties[4]), world);
                    world->entities[x * MAPSIZE + y] = ent;
                    if (entityProperties[2] == "turret")
                    {
                        std::string particleType;
                        if (entityProperties[3] == "turret_ice")
                            particleType = "particles/ice";
                        else if (entityProperties[3] == "turret_fire")
                            particleType = "particles/fire";
                        else if (entityProperties[3] == "turret_magic")
                            particleType = "particles/magic";
                        else
                            particleType = "particles/poison";

                        int mapX = world->game->tileSize * x - world->game->tileSize * y - world->game->tileSize + world->game->mapXOffset + 13 * world->game->tileScale;
                        int mapY = (world->game->tileSize * y + world->game->tileSize * x) / 2 + world->game->mapYOffset - world->game->tileScale - 9 * world->game->tileScale;
                        Particle particle(mapX, mapY, particleType, 5, 1);
                        world->game->addTurretParticle(particle, x, y);
                    }
                }
            }
            else if (text.rfind("map:", 0)
            {
                text.replace(0, 4, "");
                for (int i = 0; i < MAPSIZE; i++)
                    for (int j = 0; j < MAPSIZE; j++)
                        world->tilemap[i][j] = text[i * MAPSIZE + j] - '0';
            }
            else if (text.rfind("time:", 0)...)
            else if (text.rfind("hearts:", 0)...)
            else if (text.rfind("crystals:", 0)...)
            else if (text.rfind("monstersKilled:", 0)...)
            else if (text.rfind("difficulty:", 0)...)
        }
    }

    file.close();

    return true;
}
```

PS. W grze znajdują się jeszcze 3 poziomy trudności, od których zależy prędkość ruchu i hp przeciwników, ale nie zmieściło się to w prezentacji heh