Game of Mazes

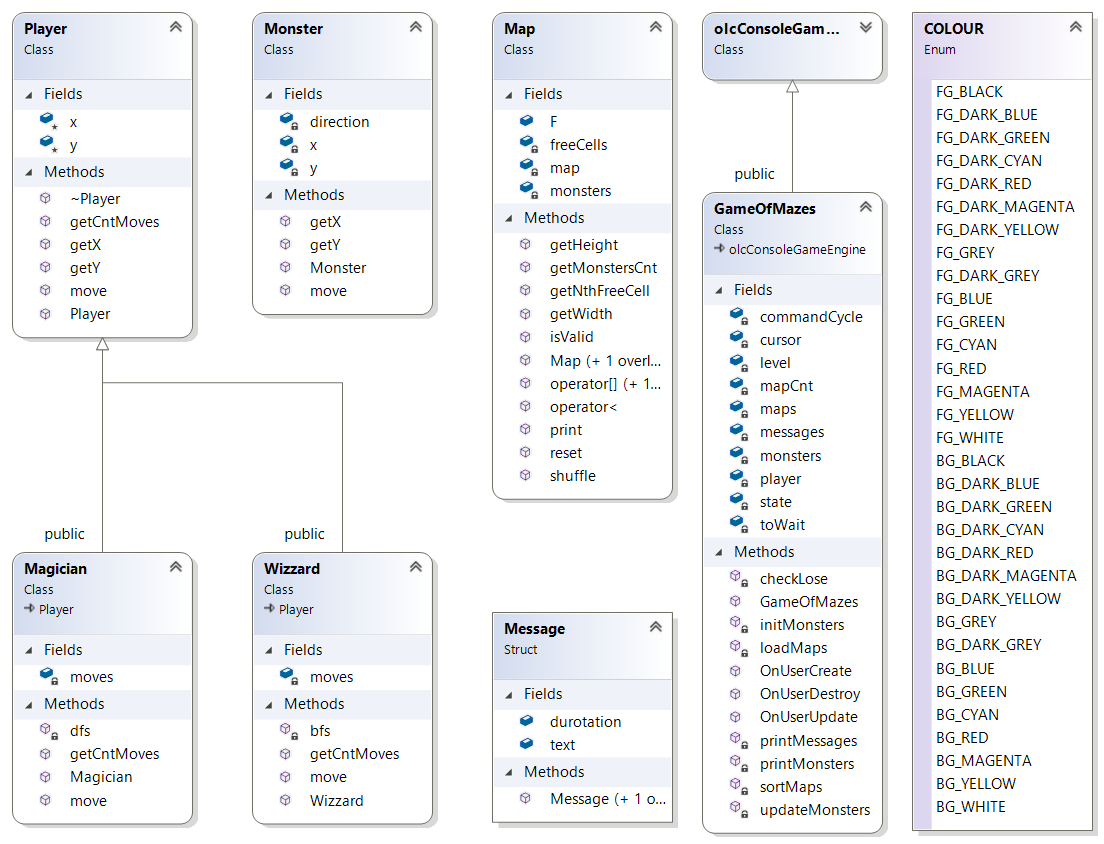
A brief description

Mazes... We see them all through our lives, from children to... programers. The concept is easy: you have to get from a point, called „starting point“ to another one, called „end point“ through some network of paths... and that‘s all. Even though the task is all clear and easy at first glace, the algorithms required for the computation of the problem are even easier – the well known DFS and BFS. Yay! The current project displays, in a fun and iteractive way, how DFS and BFS work.

Architecture

The architecture of the project is as simple as the task itself.

* We have a pure virtual multithreaded olcConsoleGameEngine class, which is being used for more user-friendly UI.
* We have a self-called GameOfMaze class, which inherits olcConsoleGameEngine and is the core class, connecting all others.
* We have a class Map, which stores the information for a single maze.
* We have a pure virtual Player class, whcih is being inherited by classes Magician and Wizzard - the representatives of the two traversal algorithms.
* We have a class Monster, representing the enemy of the user.



olcConsoleGameEngine

Data members (protected):

struct sKeyState

{

bool bPressed;

bool bReleased;

bool bHeld;

} m\_keys[256], m\_mouse[5];

int m\_mousePosX;

int m\_mousePosY;  
int m\_nScreenWidth;  
int m\_nScreenHeight;  
short m\_keyOldState[256] = { 0 };  
short m\_keyNewState[256] = { 0 };  
bool m\_mouseOldState[5] = { 0 };  
bool m\_mouseNewState[5] = { 0 };  
bool m\_bConsoleInFocus = true;

CHAR\_INFO \*m\_bufScreen; //the screen buffer  
std::wstring m\_sAppName; //the title of the console  
HANDLE m\_hOriginalConsole; //handle to the standart console  
CONSOLE\_SCREEN\_BUFFER\_INFO m\_OriginalConsoleInfo; //screen buffer to the standart console  
HANDLE m\_hConsole; //output handle to the current console  
HANDLE m\_hConsoleIn; //input handle to the current console  
SMALL\_RECT m\_rectWindow; //struct having left, top, right, bottom

Member functions (public):

int ConstructConsole(int width, int height, int fontw, int fonth);  
virtual void Draw(int x, int y, short c = 0x2588, short col = 0x000F);  
void Fill(int x1, int y1, int x2, int y2, short c = 0x2588, short col = 0x000F);  
void DrawString(int x, int y, std::wstring c, short col = 0x000F);  
int ScreenWidth();  
int ScreenHeight();  
sKeyState GetKey(int nKeyID);  
sKeyState GetMouse(int nMouseButtonID);  
int GetMouseX();

int GetMouseY();  
void Start(); //creates the game thread: thread(&olcConsoleGameEngine::GameThread, this);

// User MUST OVERRIDE THESE!!

virtual bool OnUserCreate() = 0;

virtual bool OnUserUpdate(float fElapsedTime) = 0;

// Optional for clean up

virtual bool OnUserDestroy();

GameOfMazes

Data Members (private):

Map maps[1024]; //holds the data of all maps

size\_t mapsCnt; //holds the count of the maps

size\_t level; //the current level of the player

Player \*player; //ptr to the player class (Magician or Wizard)

pair<size\_t, size\_t> cursor; //cursor for putting walls

string state; //what the main game thread loop should do

string commandCycle; //command for after the infinite cycle

float toWait; //variable, used to track time

list<Message> messages; //interactive list of messages

vector<Monster> monsters; //data for all the monsters of the current map

Members Functions (private):

void loadMaps(); //loads maps from a file DIR\_MAPS

void sortMaps(); //sorts them by level

void initMonsters(size\_t level); //sets initial random positions of the monsters

void updateMonsters(); //all monsters move if possible

void printMonsters(); //prints all monsters to the console

bool checkLose() const; //returns true if the player and monster overlap

void cycle(); //"pauses" the main game loop untill a button is pressed

bool play(float fElapsedTime); //the process of "walking" through the maze

void mapManagement(); //the process of putting walls

void printMessages(size\_t x, size\_t y, float fElapsedTime); //prints the messages and if needed, deletes them

Member Functions (public):

virtual bool OnUserCreate() override; //initializes the class

virtual bool OnUserUpdate(float fElapsedTime) override; //main game thread loop

virtual bool OnUserDestroy() override;

Map

Data Members (private):

vector<vector<char> > map;

vector<pair<size\_t, size\_t> > freeCells; //a vector, containing all cells of type '.'

size\_t monsters; //count of monsters

Data Members (public):

size\_t F; // F = freeCells.size() - monsters – 2

Member Functions (public):

Map(ifstream &iFile); //loads a map from a file. If failed to import, throws an exception

bool isValid(); //checks if the map is valid (contains only '.' and '#' and a path from (0, 0) to (m-1, n-1))

void print(olcConsoleGameEngine \*cge) const; //prints the map to the console ptr the modified (with 'X'-es)

void reset(); //undo all 'X' to '.'

void shuffle(); //shuffle freeCells to later place monsters freeCells[0], [1], [2], ...

size\_t getHeight() const { return map.size(); }

size\_t getWidth() const { return map[0].size(); }

size\_t getMonstersCnt() const { return monsters; }

pair<size\_t, size\_t> getNthFreeCell(size\_t n) const { return freeCells[n]; }

bool operator<(const Map &other) const;

vector<char>& operator[](size\_t pos);

const vector<char>& operator[](size\_t pos) const;

bool operator<(const Map &other) const; //predefined operator<. Returns true if \*this is lower level than other

vector<char>& operator[](size\_t pos); //predefined operator[] for non-const

const vector<char>& operator[](size\_t pos) const; //predefined operator[] for const

Player

Data Members (protected):

size\_t x;

size\_t y;

Member Functions (public):

virtual ~Player() {} //needed so that x and y destructors are called

virtual pair<size\_t, size\_t> move() = 0; //virtual func, which when inherited will return the next move of the player

size\_t getX()const { return x; }

size\_t getY()const { return y; }

virtual size\_t getCntMoves()const = 0;

Magician

Data Members (private):

queue<pair<size\_t, size\_t> > moves; //holds the path of the player

Member Functions (private):

bool dfs(Map &m, size\_t x = 0, size\_t y = 0); //initializes moves and returns true if path found

Member Functions (public):

Magician(const Map &m); //initializes and throws an exception if path not found

virtual pair<size\_t, size\_t> move() override; //returns the next move of the player + repositioning

virtual size\_t getCntMoves()const override { return moves.size(); }

Wizard

Data Members (private):

stack<pair<size\_t, size\_t> > moves; //holds the path of the player

Member Functions (private):

bool bfs(const Map &m, size\_t x = 0, size\_t y = 0); //initializes moves and returns true if path found

Member Functions (public):

Wizzard(const Map &m); //initializes and throws an exception if path not found

virtual pair<size\_t, size\_t> move() override; //returns the next move of the player + repositioning

virtual size\_t getCntMoves()const override { return moves.size(); }

Monster

Data Members (private):

size\_t x;

size\_t y;

string direction;

Member Functions (public):

Monster(size\_t x, size\_t y);

pair<size\_t, size\_t> move(const Map &map, const vector<Monster> &m); //returns the next move of the monster + repositioning

size\_t getX() const { return x; }

size\_t getY() const { return y; }

Message

The purpose of this class is only for better user-interface. It contains some simple wstring text and some float duration. While the duration is higher than 0, the text must be shown somewhere (by the user of the class). When the duration gets lower or equal to 0, the text must be removed from wherever it has been displayed (by the user of the class).

Data Members (public):

std::wstring text;

float duration;

Member Functions (public):

Message(const std::wstring &text, float duration): text(text), duration(duration) {}

Message(const std::string &text, float duration): text(converter.from\_bytes(text)), duration(duration)