Maze Runner - Programmer's Documentation

Programmer Documentation for classes
Programming 2 (NPRG031) at MFF UK..

Revisions

Creation date:	25.9.2020
Document version:	01
Last updated:	25.9.2020

1. **Specification Breakdown**

   Maze Runner is a pseudo 3D game, where 3D effect is done via raycasting. The aim of the player is to successfully escape a maze he is set to. The game is based on this game (my semestral project for Programming 1). Original specification link.

   Functional requirements of the project:

   1. GUI
   2. Minimap
   3. Main menu
   4. Randomly generated maze
   5. Maze protector - via billboarding

2. **Architecture / Design**

   This program is split to several objects that either interact or complement each other. "Main wrapper" of all other classes (except for the Program.cs class) is Game.cs class. GUI, minimap and main menu are all included in the mentioned Game.cs class as this class handles Forms and Drawing. Randomly generated maze has its own class called Map.cs, so does Maze protector - class Enemy.cs.

   The Game.cs has several procedures to handle time, input, rendering (and wrapping all of these in a game loop). Each object has its own methods as well. There is not for example a specific function to draw the minimap (handled in Render procedure - as well as the scene or e.g sprite). The functions and procedures were made to wrap more events which are interrelated.

   For my program, I decided to define several data structures - objects. These are Vector, Map, Player, Enemy and GameMap. Map and Enemy map perfectly to fourth and fifth functional requirements. Others (1.-3.) are handled in a game loop.

   Global variables used in my program ale bool info about keys being pressed (WASD) to provide continuous movement, new x and y coordinates position of the player - to

check for wall collisions and time elapsed for checking when to spawn the enemy (timer for FPS is handled via WinForms timer object).

I have implemented a ray casting algorithm (RCA) that is invoked every time a new frame is rendered. This algorithm works with generated map a the player - according to players FOV and his direction, RCA checks for wall intersections in the whole FOV. I wanted to use my knowledge from Linear Algebra so I adapted the RCA algorithm that way (vectors, rotation matrices, determinants, etc.) At worst, this algorithm checks almost every horizontal and vertical line in a grid and does this for “each pixel” of the width of the screen (I went for 1200), so its time complexity is asymptotically O(n*w), where n is number of tiles in a row/column in a grid and w is the width of the used display screen rectangle. This algorithm stores information for each ray it casts, so its space complexity is asymptotically O(w).

Algorithm for random generation of the maze is closely described here, as I used it in my semestral project for programming 1.

Last but not least, I implemented a billboarding algorithm to display the sprite. This algorithm is called everytime new frame renders. It transforms x and y coordinates of the sprite according to the player's direction (and vector that is perpendicular to it) and renders each stripe of the sprite's texture on the screen. It draws each stripe of sprites texture separately so it’s time complexity is O(t) where t is the width of the sprite (after transformation).

Last important algorithm that was implemented is BFS. It’s used to find the shortest path from enemy’s to the player's location. This algorithm works with objects Player, Enemy and Map. These three objects are all components of GameMap class which this algorithm and procedure that implements it receives. At worst, implemented BFS works with O(n^2) time and O(n) space complexity, where n is the number of tiles in a row/column of the grid. Thanks to the maze generating algorithm, half of the squares of the grid (not counting the wall that encloses the whole maze) is approximately empty, so BFS’ time complexity seems to be worse than it actually is.

The user input is processed in Game class which handles KeyEvents and Events thanks to the fact the whole game is WinForm application. Info that is being checked about Key is whether the particular key is up or down.

3. Technical documentation

Data structures:

- *Vector*
  - attributes:
    double x - x coordinate of the vector in Cartesian system
    double y - y coordinate of the vector in Cartesian system

  - methods:
    Vector RotateCounterclockwise(double angle) - rotates Vector object counterclockwise by given angle, original vector will be rotated and saved

    Vector RotateClockwise(double angle) - same as previous function but clockwise

    Tuple<double, double> RotateClockwiseSym(double angle) - symbolic clockwise rotation of the vector, returns tuple with new coordinates, rotated vector is not saved

    Tuple<double, double> RotateCounterclockwiseSym(double angle) - same as previous function but counterclockwise

    double GetAngle() - returns angle in radians; angle that vector forms with (1,0)^T vector

- *Player*
  - attributes:
    Vector direction - current looking direction of the player
    int positionX - current position of the player - x coordinate
    int positionY - current position of the player - y coordinate
    int Step - size of player's step - how much he moves each frame if needed
    double FOV - field of view (in radians)
    int startLocationX - starting location - x coordinate
    int startLocationY - starting location - y coordinate
    double rotationSpeed - by how much degrees player rotates each frame if needed
    bool Killed - info whether player was caught by enemy to end the game properly

  - methods:

    void SetPosition(int x, int y) - sets player's position to given x,y

    Tuple<int,int> GetPosition() - returns player's current position

***void* SetDirection(*Vector* newDirection)** - sets player's direction

***Vector* GetDirection()** - returns player's current direction

- *Enemy*
  - attributes:
    int positionX - current enemy's position - x coordinate
    int positionY- current enemy's position - y coordinate
    int startLocationX - starting location - x coordinate
    int startLocationY - starting location - y coordinate
    int Step - size of enemy's step (in the same manner as Player's one)
    Bitmap texture - enemy's texture (square texture)
    long SpawnTime - millis to elapse before enemy spawns
    bool Spawned - info whether enemy spawned
    int StepsToMake - how much steps to make before computing next path (BFS)
    Tuple<int, int> nextStepDirection - which way to go next

  - methods:
    ***void* SetPosition(*int* x, *int* y)** - sets enemy's position to given x,y

    ***Tuple*<*int,int*> GetPosition()** - returns enemy's current position

    ***Tuple*<*int,int*> FindWay(*GameMap* gameMap)** - performs BFS and computes which way to go next to get to player's location by shortest path

    ***void* Move(*GameMap* gameMap)** - handle moving; moves if it has steps to make, if not, compute it's way and move that way

- *Map*
  - attributes:
    int SideSize - number of tiles on one side (width=height)
    int[,] Grid - grid where info about existence of wall on each tile is stored
    int TileSize - size of one tile in the grid (width=height)

  - methods (describer closely here):
    ***void* CreateFoundation()** - creation of template to build maze in

    ***int* FoundationsLeft()** - returns how many foundations are left in the grid

    ***Tuple*<*int,int*>? ChooseRandomFoundation(*Random* random)** - choose random foundation to build wall from

***void* BuildWall(*Random* random)** - build wall from returned random foundation

***void* BuildMaze()** - handle all building functions -> build maze

- *GameMap*
  - attributes:
    Map map - map game takes place in
    Player player - player set to the map
    Enemy enemy - enemy set to the map

  - methods:
    ***Tuple*<*double, int, double*> Raycast(*int* x, *int* y, *Vector* direction, *int* max)** - cast one ray and return info about it <size, wall it hit, angle it was cast from>, takes x, y coordinates of position of the player, hist direction and max length of the ray it can cast

    ***List*<*Tuple*<*double, int, double*>> CastRays(*int* screenWidth)** - cast all the rays to fill the screen width

Functions and procedures:
***void* Render(*GameMap* gameMap)** - renders everything, i.e scene, minimap, sprite

***void* timer1_Tick(*object* sender, *EventArgs* e)** - game loop, handles spawning, end of the game, viewing menu, input

***void* MazeRunner_KeyUp(*object* sender, *KeyEventArgs* e)** - detects whether particular keys are up

***void* MazeRunner_KeyDown(*object* sender, *KeyEventArgs* e)** - detects whether particular keys are down

***void* NewGame_Click(*object* sender, *EventArgs* e)** - detects whether user clicked on “New Game” button and starts the game if so

***void* Exit_Click(*object* sender, *EventArgs* e)** - detects whether user clicked on “Exit” button and exits the application if so