Programming II – Semestral project

Gabriela Suchopárová

26. září 2017

1 User documentation

The goal of this game is to kill as many enemies possible. The enemies spawn more frequently as the game goes. Player can gain hit points with collecting dead enemies. There are multiple enemy, weapon and obstacle kinds.

1.1 Controls

- WASD – movement
- mouse – rotation
- spacebar – shooting
- number keys – weapon change
- F3 and F12 – soft and hard restart
- F5 – fullscreen
- Esc – exit

1.2 Weapons

There are different weapon types which differ in some parameters: shooting speed, maximum number of bullets shot at a time, minimum distance between two bullets.

1.3 Game map

The game map consists of walls, free space and map border. There may be enemies appearing at any random place, but no items are supported yet.

1.4 Enemy spawns

Enemies spawn at randomized places on the map. Once spawned, they move towards the player. If the distance between an enemy and the player is small, the enemy starts damaging both the player and itself. An enemy who dies this way does not count as a kill. At any time, an enemy has a chance to teleport on a random place to avoid large enemy clusters (as this could lead to huge hit points gains and to repetitiveness).

2 Programmer documentation

2.1 Algorithm description

Raycasting The environment is created as a projection of a 2D map into the screen plane using the raycasting algorithm. For every column of pixels on the screen, a single ray is casted and the distance of crossed walls is computed. Then, we determine how many pixels of the column a particular wall takes.

As the walls take up whole grid cells, we need to find grid intersections of the ray in order to be able to check if it crosses a wall. This can be achieved with a variation of DDA (differential digital analyzer) algorithm, which is also used for line rasterizing. The difference between the original DDA and our algorithm is that we move in both x and y direction, whereas in the DDA only one direction is chosen. Other than that, all invariants remain the same.

First we compute the distance to the nearest gridline that is parallel with x-axis and y-axis. Then, using trigonometry, the distance between two intersections (x or y lines) is computed. Afterwards the algorithm starts; with each loop the current position of the ray moves by one grid cell either in the x or y direction. Every step the grid cell is checked for a wall; if a wall is discovered, its distance from the start position and the exact intersection position is saved. The raycasting ends after a certain condition is met (e.g. number of walls, ray distance).

After the data from raycasting is gathered, the game scene can be drawn. The height of a wall is computed from its maximum height and its current distance. As there are multiple wall sizes allowed in this implementation, the overall maximum height of all walls has to be taken in account, so all smaller walls are drawn in relation to the biggest one.

As a ray can cross multiple objects, the walls which are further away need to be drawn first. Also, to avoid drawing a pixel more than one time, walls that are completely hidden are excluded and only the higher part of visible walls is drawn.

Raycasting is also used for object movement, as it can easily spot obstacles. The difference between the wall raycasting and movement raycasting lies in the raycasting distance; the latter ends just as the maximum cast distance has been overcome.

Sprites For drawing enemy sprites another technique is used. Although the enemy could be crossed by a ray just like a wall, ti would be harder to determine the exact position on the screen, as unlike walls its position coordinates can be floating point numbers. Furthermore, it would be necessary to check every tile for enemies, so either the game map would be extended to contain objects instead of integers, or there would be a look-up table for enemy cell positions. Thus, the sprites are instead projected on the screen plane taking their relative position to the player. This is achieved by solving the equation $v' = Av$, where v is player position subtracted from enemy position and A is a matrix of direction

and screen plane vector:

$$A = \begin{pmatrix} dir.X & dir.Y \\ plane.X & plane.Y \end{pmatrix}$$

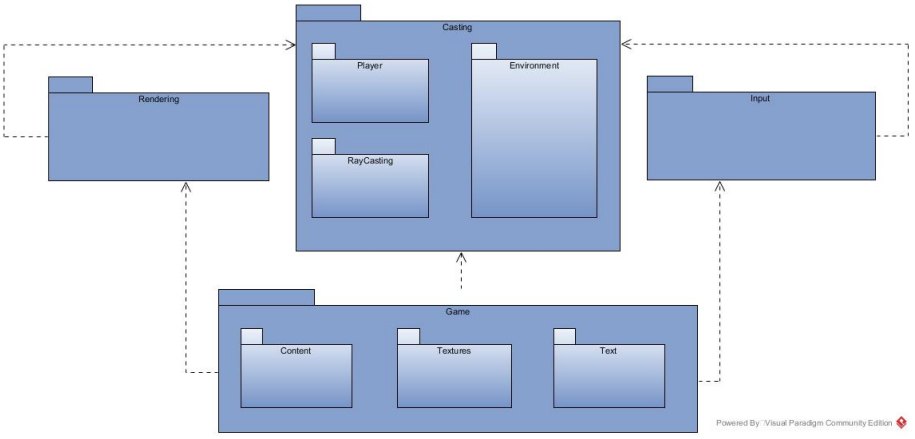
Then the x coordinate of the vector v' determines the horizontal position on the screen and the y coordinate represents enemy's distance from the screen plane (negative value of y means the enemy is behind the player). Lastly, as the enemy could be partially hidden by a wall, it is sorted into wall list of corresponding rays and rendered column by column alongside the walls.

Bullet shooting There are two common algorithms for bullet shooting in game development. The first one is shooting with raycasting. When shooting, a ray is cast, so the hit happens at the same time as the bullet is fired. The advantage is that the bullet is guaranteed to hit if it flies straight towards the target.

The second possibility is to project it as a moving element on the game plane and with every update check if it hits an enemy. The drawback of this approach is that it is possible that a bullet passes through an enemy, because it moves in a discrete way and therefore can skip the hitbox. An option to avoid this is to have hitbox sizes greater than the bullet movement step or to do a short raycast like in the first algorithm.

In this implementation, the second possibility is used, as it is easier to render the bullets this way (in the same manner as drawing the enemies).

3 Architecture



The program is separated into 4 modules: Casting, Rendering, Input and Game. Casting is the largest modul with not only classes which are necessary for raycasting, but also enemy, weapon and other similar classes. Module Rendering converts data provided by raycasters to raw texture data. Input is a small module which contains classes for loading settings. It also references implementations of Casting interfaces and passes them to the Game module using Dependency injection. Lastly, the modul Game contains the actual game logic and graphics.

3.1 Input

User input Player can move on the game map using WASD keys and rotating the view with the mouse. When one of the keys is moved, the new position is computed as follows:

$$W/S : nextPosition = position \pm speedMultiplier * direction \quad (1)$$
$$A/D : nextposition = position \pm speedMultiplier * screenPlane \quad (2)$$

Settings The game settings can be set by input text files. The main settings file must be in the same folder as the Game1.cs file and be named *settings.txt*. Input strings of the main settings files are separated be newlines; all other files have on line of comma separated input values.

The **main settings file** must have the following format (input strings are separated by newlines):

(Line 1) : "wall file path"
"map file path"
"enemy file path"
"weapon file path"
"player file path"
"sky texture path"
"floor texture fath"
enemy spawn time in seconds (int)
horizontal resolution (int)
vertical resolution (int)

The **wall file** contains all possible wall types, including -1, which is the map border wall; 0 should not be used for any wall:

wall index *(int)*, x side texture, y side texture, x alt color *(hex)*, y alt color *(hex)*, wall height *(int)*

The **map file** contains multiple lines of comma separated non-negative integer numbers, which represent walls. The value for free space is 0, all other values must match the wall input file. There must not be any blank lines. All lines should have the same length.

The **player input file** contains one line of initial player values. Starting position should correspond to map environment:

position x *(float)*, position y *(float)*, direction x *(float)*, direction y *(float)*, hitpoints *(int)*, name

The **enemy input file** contains multiple lines of enemy input data:
type id *(int)*, hitpoint *(int)*, position x *(float)*, position y *(float)*, living texture path, killed texture path, height *(int)*, width *(int)*, movement speed *(float)*, hitbox size *(float)*, spawn time interval in seconds *(float)*

The **weapon file** can have up to 10 input lines of weapon data. No error occurs if there are more weapons provided, but they won't get mapped to numbered keys and cannot be used in the game:

max ammo *(int)*, weapon texture path, shooting speed *(float)*, bullet texture path, exploded bullet texture path, bullet size *(int)*, min bullet distance *(int)*

4 Performance

4.1 Rendering

Although the raycasting algorithm is quite fast for relatively small maps and a low maximum walls-per-ray count, the performance bottleneck is the rendering. The number of pixels is too high even for smaller resolutions, because the column data is copied pixel by pixel and includes accesses to other textures.

A solution might be using Cudafy, a library for C# which enables to use the GPU for some computations. If the project is developed further, this approach could be worth trying.

4.2 Enemies

The performance also drops with greater numbers of enemies, because with every update we need to check the enemies' distance from the player and from bullets. Also, each enemy in front of the player needs to be added to crossed object list of multiple rays, as we do not know yet if it is covered by a wall or not. Therefore, the number of enemies should be reasonably limited.

5 Possible extensions

There are various playability, graphics and other extensions that could be included to provide more variability:

- separate healing items
- limited ammo
- better mouse movement responsibility
- going through walls – doors (new movement condition)
- bullets hitting more enemies
- more bullet types – freeze,...
- more intelligent enemies