Final Project Media Processing

Introduction

This final project will take the rest of the semester. As evaluation, every team will present its work in week 13. A team has 3 students, each having a specific subtask as final responsibility. All software updates need to be done by *gitlab.groept.be*. The list of commits of every team member will be used for evaluation. Every team should also have a planning on Gitlab, where we can see what each team member is working on per week. A video about this is available on Toledo.

The final goal is to create a **game-like application** where the protagonist needs to be able to navigate in a given 2-dimensional, grid-based world, to be able to attack different kinds of enemies and to gather the necessary health packs. Navigation of the protagonist can be done both manually (e.g. arrow keys) and automatically (through A* **pathfinding**). Additionally, there should be an **auto-play** mode where the protagonist tries to defeat all enemies automatically. A flexible strategy needs to be developed for this, to increase the chances of victory in an unknown world.

Moving in the world will take energy defined by the value of the tile you are moving to, defeating an enemy will need enough health (which can be increased by going to a tile with a health pack on it).

Your application needs to be able to switch between 2 different visualizations of the same world: a 2D graphical representation and a text-based representation. Make use of the **MV(C) design pattern** to help you in reaching this goal. Make your architecture suited to easily be extended with more types of enemies (each having different behavior) and more even more ways of visualization.

You do not start from scratch for this project. On Toledo you will find a header file (world_v5) with the definition of the World – Tile – Enemy – PEnemy[1] – XEnemy[2] objects you will use, together with a **shareable library** implementing the defined methods. All these types are part of the model of your application. All of them will play a role in our game, each having their own rules. The library also contains functionality to generate the world you are playing in based on an image. Each pixel in this image will represent a tile in your grid-based world[3].

[1] PEnemy is an enemy which is poisoned. It will lose its poison when attacked and finally it will die
[2] Xenemy is a yet unknown type of enemy which will have specific behavior that you may choose yourself
[3] The value of a tile is determined by the grey value of the corresponding pixel

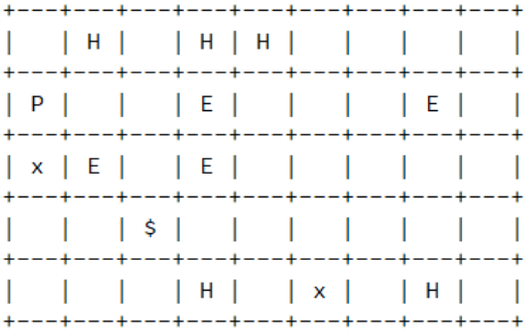
Detailed specifications

Each team member is responsible for one of the subtasks A, B and C. Up to you to organize D.

Subtask A. “Graphical” representation

- Look at QGraphicsView / QGraphicsScene / QGraphicsItem for this task
- Each tile has a fixed dimension in pixels and has a color defined by the value of the tile. In the simplest setting the world is visualized as a scaled version of the image used to create the world.
- All other objects (protagonist, enemies, health pack) get a specific visualization with the same dimensions as a tile. Each type of enemy has a different visualization. Also use a different visualization for non-defeated and defeated enemies. PEnemies need a more complex visualization. Once they get attacked, their poison level will gradually (but in a random time) go to zero while poisoning a wider area of the world surrounding it. Your visualization should animate also this effect.
- For the protagonist, at least the following actions should be visualized: attacking, using a health pack, getting poisoned, moving, and dying. Minimally these actions should result in a brief color change in the protagonist, but you can go further and include full animations if you wish.
- You can control the protagonist with simple movements (up, right, down, left)
- When you click on a valid position in the world, the pathfinder is used to find the optimal path and an animation is shown of the protagonist moving over the world (check out the QTimer class for animations)
- The path followed by the protagonist in “auto” mode should be visualized in some way
- The architecture of your application should make it easy to add new type of “actors” in the world with different behaviour.

Subtask B. “Text-based” representation



- The world itself should have an ASCII art-like, string-based visualization. A simple printout of the events that are happening is insufficient. We still need to be able to see the world.
- This text visualization should not simply be printed in the terminal, but should be integrated in the applications overarching UI. For example, there could be a central window in your UI that shows your world, and two buttons ‘Graphical view’ and ‘Text view’ that switch the representation shown in that window.
- The same protagonist actions should be visualized as in the graphical view (attacking, using a health pack, getting poisoned, moving and dying). Again, the minimal approach is to briefly change the protagonist’s color, but you can use a different approach if you wish.

- In this mode, all interaction with the protagonist is done by text commands instead of mouse clicks. The available commands should be easy extendable (we don't want to see a big if/else tree). Minimal required commands are:
    - *up, right, down, left*
    - *goto x y (this triggers the pathfinder!)*
    - *attack nearest enemy*
    - *take nearest health pack*
    - *help (this prints a list of all available commands)*
- Ideally a command should be understood as soon as it is unique (so typing a 'r' would be enough to go right), consider tab completion as a possible extension.
- The architecture of your application should make it easy to add new commands.

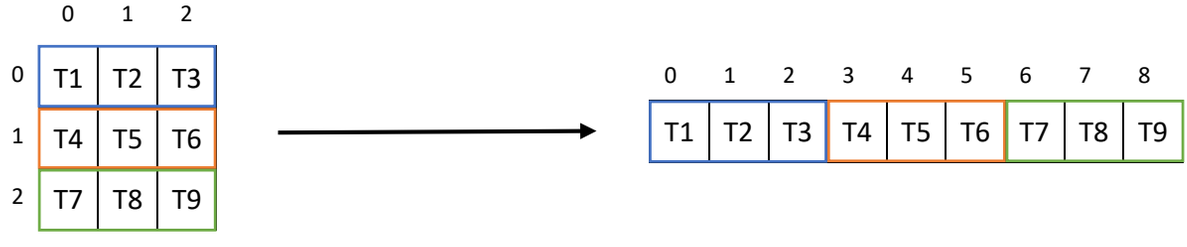
Subtask C. Pathfinding

The protagonist needs to have sufficient intelligence to navigate in an unknown world, populated with health packs and enemies. So you need to develop a path-finding algorithm to calculate the most appropriate way to get from position (xstart, ystart) to position (xend, yend). This algorithm will be called when you move the protagonist to a certain tile (either by clicking it in the graphical representation, or by using the goto command in the text-based representation), or during autoplay execution.

There are of course many ways to define what the most appropriate path is, but we will use a combination of distance and the difficulty to move to a certain position (e.g. you can see the tile value as the difficulty to traverse that tile). Tiles with a value of infinity must be seen as impassable.

A* is the most used path-finding algorithm. We will not use an existing version, but implement our own using the most appropriate data structures and generic algorithms available from STL and/or Qt. You need to be able to answer all detailed questions about this algorithm: choice of data structure, efficient implementation... Keep in mind that this is a quite demanding application, so use all possible ways to optimize its performance.

Note that the world tiles returned by the library are stored in a specific order (see below). Use this information to avoid unnecessary iteration.



As a benchmark you will need to find your way into a big maze (a picture of a maze with dimensions 2400 x 2380, so nearly six million tiles) in a "reasonable" time. Experience from the past shows that this is possible in less than 1 second. Make sure your project supports easily loading in a new map and finding a path to a certain position, so you can show the performance of your implementation during the defense.

Subtask D. Integration and extra features

User interface

You will need a basic UI which gives you the possibility to switch between the two views and control some settings. Your UI should support at least the following features:

- Show the protagonist's health and energy bars
- Allow switching between graphical and text-based representation
- Allow the input of commands in text-based representation
- Set the weight of the heuristic in your pathfinder
- Control the speed of your animations
- Zoom in/out on your world visualisation

Autoplay

The game should also have an autoplay mode. Minimally implement the following strategy: select nearest enemy, find out if you have enough health to defeat it, otherwise find first a suitable health pack. See if you can think of a more advanced strategy…

Of course navigating in the world has also a cost: your energy level will decrease with the cost (=value) of the tile you walk to (or the difference between the values). As a consequence you only have a limited field-of-view in which you can look for enemies and/or health packs. Once you have defeated an enemy, your energy level is restored to maximum. The visualization of the enemy needs to be changed after it is defeated and the tile becomes impassable .

You repeat this scenario until all enemies are defeated (you win) or you are not able to defeat one any more (you lose).

XEnemy

You should add a third type of enemy in your game, aside form regular Enemies and PEnemies (poison). Make sure your XEnemy has its own visualization. The behavior of your XEnemy is up to you. Some ideas:

- A transforming enemy, like a werewolf, who switches between two forms after a certain amount of time or turns
- A moving enemy
- An enemy who doesn't die when killed, but resurrects as a zombie
- An enemy that needs to be hit three times, but teleports after each hit
- An enemy that grows stronger the longer it is on the board
- …

Note that your XEnemy is part of your model layer, so you will have to extend the library for this. You cannot / are not allowed to change the source code of the library, so you will need to use inheritance here. Also, the library returns a vector of all enemies, but this will only contain regular enemies and PEnemies. You'll need to come up with a way to convert a certain number of regular enemies to XEnemies.

Bonus features

If you have time and energy left, you can try adding some bonus features. Use your imagination here. Some suggestions:

- A Save / Load system which stores the level's current state

- Adding healthpacks to specific tiles with drag-and-drop
- New enemy types
- More than one protagonist
- Multiplayer support
- More animations
- An additional visual representation (different 2D visualization? 3D? Virtual reality?)
- A more advanced pathfinding approach
- …

The library

On Toledo you will find a header file (world_v4) with the definition of the World class you will use. The implementation of all available functionality can be found in a shareable library (or a dll if you prefer that). So the first step is to adapt your project settings to be able to use these things. The given libworld.so is a 64 bit Linux library, if you need another version, you need to get the source code and build the project yourself. Keep in mind however, that this should be a "given" file: you are not allowed to change any functionality of the given code.

Use the method `World::createWorld(QString filename, unsigned int nrOfEnemies, unsigned int nrOfHealthpacks, float pRatio)`

to initialize your world and create a protagonist.

- `filename` is the name of an existing image. Use the **Qt Resource System** to add images to your project in a platform independent way.
- Every pixel of the image stored in filename will become a Tile in your world, the grey value of the pixel determines the value of the tile (and thus how difficult it is to traverse). This way, you can create new worlds by just drawing an image. We will also add a couple of images on Toledo that you can use to generate worlds.
- Your application should visualize all tiles using their position. The value attribute should be used to give a Tile a certain color.
- The 3 other parameters to `createWorld` define how many enemies and healthpacks you will have and what ratio of the enemies will be PEnemies.
- The method `getTiles()` returns a vector with unique-ptr to the tiles you just created.

Keep in mind that all data you get from the library are collections of unique_ptr. This is a 2nd version of a smart pointer available in the STL. Like the name suggests, with unique_ptr you can have only 1 pointer referring to the pointee. This means that you cannot assign unique_ptr's to each other, otherwise you would have 2 pointers pointing to the same pointee. The moment that the library returns the collection, it transfers ownership to your application.

Planning

You can make your own planning for this project, but we expect you to loosely adhere to the following structure:

Week 6: Experimentation with Qt, integrating the library and making a first graphical version

Week 7: Design architecture and create application UML. **Milestone 1**

Week 8-10: Subtask A-B-C implementation

Week 11-12: Integration and extra features

Week 13: Final presentation. **Milestone 2**

Be sure that every member can commit to Gitlab since the activity on git will also be used as an evaluation criterium. Each task has still a certain degree of freedom where you may use your own creativity.

Don't leave the integration all the way to the end. Be sure that you can do early integrations using a SCRUM like approach.

Milestones

Milestone 1: Architecture (08/11/21)

Based on all the info you have until here create a class diagram of your application. Again we will make use of Design Patterns to define our architecture. Here you should apply the Model-View (Controller) design pattern to make the visualizations as loosely coupled as possible with the underlying model.

Try again to be as detailed as possible, this class diagram will help you to divide the work among the different team members. Use smart pointers wherever possible. Use of ordinary pointers needs to be motivated!

This class diagram need to be uploaded by Monday, November 8th**. This UML is not graded**, but you will get feedback in the next lab session.

Milestone 2 : Final code (18/12/21)

Every team will get 40 min. to present its final project. All code, resources (images, other data...) + final UML class diagram (as image) + status report need to be final and on Gitlab by Saturday, December 18 2021 23.59 on gitlab.groept.be. Please send also an e-mail to your lab coach to mention that your team participates in (or postpones) the presentation. The detailed schedule will be presented later. The code will be evaluated on a 64 bit Linux environment.

If no working (compilable and runnable) project is available by that date, students are not allowed to present their project and need to rework it for the 3rd examination period in September.

Evaluation criteria

We will analyze your final uploaded code before the presentation and prepare questions beforehand. The final presentation consists of three phases:

Demo

You will start by giving a short demo of your project, demonstrating all features. Make sure you can show at least the following aspects:

- Switch between 2D and text-based view during program execution
- Move protagonist with arrow keys or by clicking / goto command (pathfinder)
- Autoplay feature
- XEnemy
- Easily switch to new map
- Pathfinding efficiency on the big demo maze
- Visual representation of path
- Show effect of heuristic weight on pathfinding

Of course, any bonus features you add will be factored into your score.

UML

Have a complete, accurate UML diagram of your code ready (you can just generate this from your code). Based on this, we will ask questions about your software architecture. We will focus on:

- Model-View(-Controller) architecture
- Correct use of Polymorphism
- OOP design principles (SOLID) think about extending with other actors, other commands, more complex visualisations…

Code

We will dive into the specifics of your code. Aside from the points mentioned above, we will also consider:

- Correctness
- Efficiency
- Avoiding unnecessary copies (pass by reference vs pass by value)
- Use of smart pointers