Gauntlet project planning

This is the plan of action for the Gauntlet project. The reason why I'm making this is because I am having trouble with foreseeing a project. Much of my trouble comes from oversights, requiring me to change many classes. This results into hacks in the classes, having function calls for one other class. This is not how OOP should look in my eyes. OOP should behave like a real object. Every function that a class has must make sense from the class's point of view.
To keep it simple, this is my attempt to make myself better in planning my code.

Table of contents

Project structure

I will be using this video as a reference

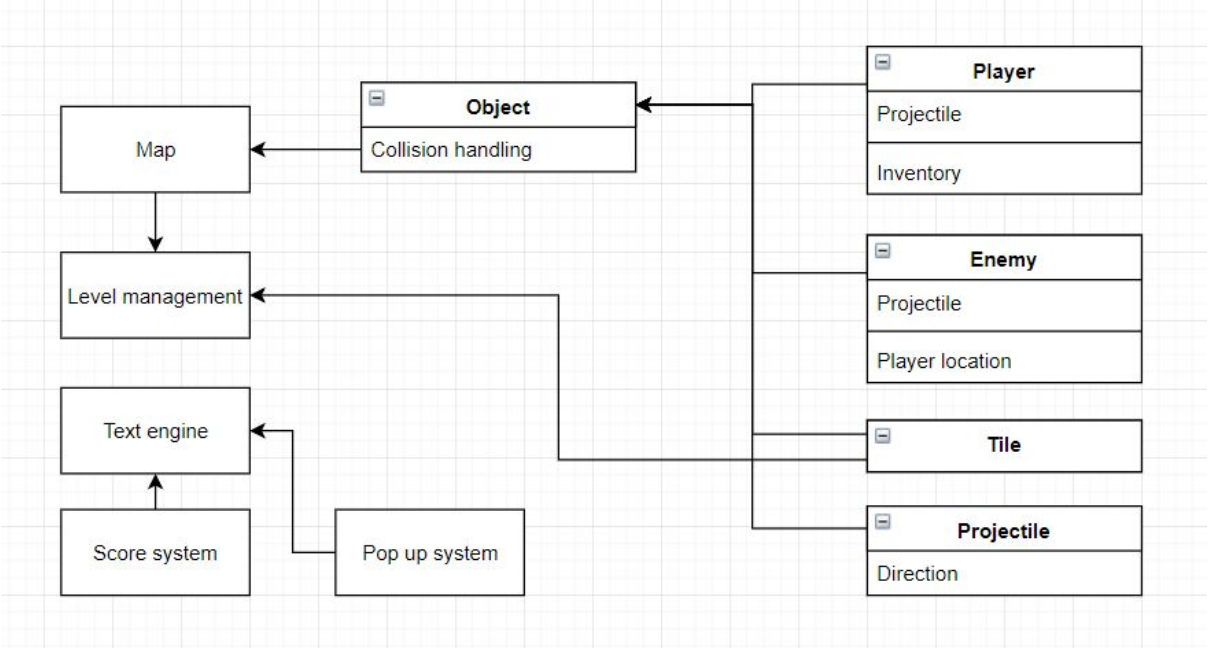
List of necessities

- A map
- A level manager
- Different kinds of player classes
- Different kinds of enemies
- Different kinds of pick up items
- Different kinds of tiles
- Different kinds of projectiles
- An inventory
- A score system
- A text engine
- Basic AI
- Collision handling
- Animation

These are the things we have to implement. Let's clean it up a bit.

- A map
- A level manager
- Objects
    - Collision handling
    - Entities
        - Animation
        - Player
            - An inventory
        - Enemies
            - Basic AI
    - Items
    - Tiles
    - Projectiles
        - Animation
- Object manager
- A score system
- A text engine
- A pop up system

Code structure



This is a broad overview, we will get a bit more specific but let me first elaborate on the choices. Tile is dependent upon level management because the exit tile needs to access new levels. Collision handling uses map to get surrounding objects. This model seems about right.
There is one thing however that we will change. Instead of allowing objects to make themselves immediately, we do it via a special constructor class. By doing this, the level can first decide the level of the enemies and then apply the right enemies. This way, we don't need to define 4 numbers for just ghost. We assign a certain ghost and duplicate it. That is what the spawner is for.

Code implementation

Maps

Maps represent the objects. They manage the lifetime and index of the objects. The index is used for collision checking and the AI.
The first map is the TileMap. This tile loads static objects like the background. They can also be called immutable solid objects, but static sounds simpler. It can be seen as a constant map.
The second map is the ObjectMap. The ObjectMap is responsible for all the objects and their position. The ObjectMap depends on the TileMap for collision related calculations. The ObjectMap is updated every frame and has collision checking for all objects that contain it.

The reason why we split these two maps is because it spares us cpu cycles. But the most important reason is that it makes programming easier. What happens when a spawner is destroyed? Do we have to spawn a new tile on it's position? This way, we can just remove the object when the collision happens. This makes logic more consistent across objects.

Objects

Object spawning

Objects represent everything on the screen with a position and texture (which brings a size with it). Objects are dynamically allocated while we load a map. The only thing that objects know about are themselves, except for doors and exits, they need to know more.
Because of this the creation of objects is inconsistent, so we will be applying an **object factory**. The object factory hides the making of objects behind a function. This not only makes creating objects consistent, is also allows us to always change the way objects are allocated.
This will be a special object factory. We have 5 types of monsters, so the factory will hold 5 types internally. We can do this because the monsters all are the same level on each level. When we want to spawn a new monster, we just clone what is already in the factory.
We can take this one step further and define 5 spawners. Every spawner is updated with a new monster type whenever a new level is loaded. If the level requires spawners, we just copy them to the game itself.

Object loop

Back in maps we wrote that maps keep track of the object's lifetime. We will go more in depth in how it will happen.
In Gauntlet, only the objects on screen are updated (I'm not sure about projectiles, so I will probably despawn them if they exit the screen). This means that maps will also only update and draw what's on screen.

How do we iterate over all the elements currently on screen? We use a bit of simple maths. We first of all we need a center position. In Gauntlet we always track the player position (singleplayer). We also need to define how much we want to see of the map. In Gauntlet this seems to be around 15 tiles in width. At least, we need the sizes of all textures, which in Gauntlet appears to be 32 x 32. Now let's do some maths.
We calculate the upper left and lower right indexes as follows.

Pos = center position
Subtract half screen size from pos
Pos2 = pos + screenSize
Check if both values aren't out of bounds
Divide pos and pos2 by textureSize
Save pos as upperleft and pos2 as lowerright index

Object collision

Object collision will not be hard to implement. We have a special class called **Collision Manager**. The reason why will be explained soon, but first I'll tell you how collision will be detected and resolved.
Every object will first request a move to a certain place. When this is all handled, we will go to the Collision Manager. In the collision manager, we have filled an vector with requested movements. While iterating over this vector, we will first check if there is any collision with an object in the ObjectMap. If there is, we already know there can't be a collision with the TileMap. If there is a collision, we go to resolving it.
While resolving the collision, we will first call the objects onCollision function. In this function the object is passed to the other object and the same for the other one.
After that, the positions for both will be resolved. Both sprites will get a position in between each other. This will make them stand against each other. After that, the other object can't move anymore because it's requested movement has already been moved. This makes sure that no double collisions of the same object happens.

So why do we make a class called Collision Manager? The map class itself could handle it, because of this class, we have to pass an additional reference to the Collision Handler. Well, that is because **objects aren't the only one who communicate with Collision Handler**. While out of the scope of this subject, we will have a class called achievement manager. The achievement manager listens to the collision handler. Every collision will also be pushed to the listeners of collision handler. This way, we can implement the pop up system that Gauntlet has whenever a player interacts with something for the first time.

Level manager

The level manager is responsible for loading all the levels. **The exit tile has to communicate with this class.** This will be handled by the object factory/spawner.
Gauntlet has a system where the player has to arrive at an exit tile to continue to the next level. If the map is responsible for the mapping of objects, level is responsible for loading the map with the right objects.

Format

The levels will be data driven, made by Tiled. Tiled makes tilemaps in the xml or json format. The framework already supplies a library to parse xml files. We will be using that. The format of an level file will contain the following:

- Monster types/levels
- The tilemap
- The object map
- The level message (marked in red)
- The level name

The map class is actually an instance of the current level. We will save the levels as their number to easily import them into the game. The level name will be saved internally.

Loading of levels

When we finish a level we can call the moveLevel function. This function offsets the current level by a given value and loads the new level number. We do this because this way exit tiles don't need to know about the current level number. Only the offset they give.
To load the level we simply convert the whole map into an vector. We define the level width and height in the save file so that we are sure that the vector is of right size. Then it's just a matter of loading the right objects into the map with the assistance of the object factory.