The Battle For Pass

Marcos Vázquez Rey

Programación Avanzada, ESAT 2015

Table of Contents

1. Intro to the Game Project

The Battle for Pass is an RPG game in which the player takes control of a valiant hero that must free the *Village of Pass* from an evil monster and its minions.
This game contains a total of 6 concatenated maps. The player must travel from first to last, and defeat the monster at the last map in order to beat the game (with the exception of map nº2, a house that can be skipped).

The source code for this project contains no propietary functions whatsoever, following the standards for C++11, and it has been tested to work without major problems in both Windows 7 and OS X 10.9 Mavericks.

There are some elements, both in-game and in the code, that serve as an example of how the game could be further expanded, but are not currently in use. For instance, the main menu contains buttons for game loading and options menu, but they lead nowhere. And the code contains several classes for Items (armor, weapon… etc.), that currently serve no purpose.

In adittion to the features strictly specified for this project, it incorporates, among others, the following improvements:

- **An Intro screen**
- **A dynamic player creation screen** -> Images are reloaded on button click
- **A loading screen**
- **Integration with maps created with the editor *Tiled***
- **A total of 6 different maps**
- **Passageways between maps**
- **Interactable NPCs**
- **Enemies on fixed positions**, that can also be edited using *Tiled*
- **Map collisions**, setting which tiles can be walked on. This can also be edited with *Tiled*
- A total of **9 different enemies**
- **Battle Backgrounds** that vary from map to map

- **Player Sprites** -> A total of 33 sprites per race/job combination (8 per every movement direction, plus one for dead character), and 2 additional images per race/job combination (face and bust/battler). This sums up to a grand total of 528 sprites and 32 images used for playable characters.

- **Continuous movement** -> The player movement on the map is made smoothly using animations and updated every single frame.

- **Map persistency** -> When entering a previously visited map, the player is at the correct spot, and not at a fixed point on the map.

- **Healing potions** -> They can be bought from merchants and used in combat

- A couple of **easter eggs** and references to other games ;)

- Unnecesarily **detailed maps**

- **Enemy recognition on map** -> When resting, the player is attacked only by enemy types that are present in the current map

- **Magic** -> Different classes have different spells available that suit them. Magic can be used in combat spending mana (magic points).

- A **spells interface** during battle

- A **battle log** detailing what's going on

- **An incremental difficulty system** -> Every map contains thougher enemies

- A **final boss** and an **end game** screen

- A **Game Over** screen

2. Data Structures Specification

2.1 Folder Structure

assets

Contains all images, map files, sounds and fonts used by the game, and it's divided into several sub-folders:

- *raw* -> Map files and images used by those map files.
- *character* -> Character images. The root holds images representing generic jobs. The following hold images for every combination of race/job.
- *background* -> Backgrounds for game screens and battles.
- *enemy* -> Enemy battlers organized by the enemy name.
- *UI* -> User interface elements (fonts and images)

include

Contains c++ header files organized semantically. These are the files to be included in each other.

lib

Contains source code for third party software and precompiled libraries. At this stage, it makes use of a modified TMX parser based on tinyxml2, in which support for base64 maps has been removed due to compatibility issues.

src

Contains source code for the project, method specification, extensive code commenting and it's organized following the same semantic logic used for header files.

docs

A few documents about the development, including this very same document.

compile.bat

A simple script to automatize the compiling and linking process in Windows environments.

2.2 Logical Structures

Source code is organized semantically. Folders in both *include* and *src* follow the same structure and hold files that are closely related in terms of functionality, often inheriting from and including each other.

Character

Character is the base class that is inherited by all playable, neutral and enemy avatars.

Friend represents a neutral NPC.

Ally represents a playable character that has both a *Race* and a *Job*:

Race is an abstract class inherited by *Human*, *Dwarf*, *Elf* and *Orc*, each one of them having their own initial stats regarding health, mana, attack and defense.

Job is also an abstract class that is inherited by *Boss*, *Hunter*, *Warrior* and *Wizard*. These apply modifications on the base stats set by the player race.

Enemy

This folder contains the classes for different types of enemies.

Foe is the abstract base class that inherits from Character and is inherited by all of the specific enemies.

All of the other classes representing enemies have the exact same methods inherited from *Foe*, and only differ in their stats.

Notice that all the enemies have MP=0, since they cannot cast spells and that field is, therefore, irrelevant.

These enemies originally appear on the map on groups of two per map. Here's a list of the implemented enemies from weakest to strongest:
Brown_Asp, *White_Asp*, *Harpy*, *Skeleton*, *Torturer*, *Troll*, *Soldier1*, *Soldier2*, *Black_Dragon*.

That last enemy acts as a final boss, and a *Game Won* screen will be shown every time one of those is defeated.

Item

These classes are currently unused, although they broadly showcase how an inventory system could be implemented.

Main

This contains classes that have no direct relation with a specific group of other classes, are general-purpose or unclassifiable.

Misc is a static class containing general-purpose methods.

Grid is a bidimensional linked list of void pointers. It is mostly used for holding information about several data layers of a *Map*.

Map holds information about every one of the in-game maps, as well as a TMX parser and some code for displaying a loading screen.

Animation holds a collection of sprites and methods for extracting and displaying them in order. It is currently used for playable characters, altough it can be easily expanded to show enemy, npcs or map animations, among others.

Manager is a singleton class that holds information about the current state of the game and is called by the majority of other classes.

Screen

These are the different screens, or scenes, or game stages, that exist in the project. Let's list them in order of appearance:

Screen is the initial class inherited by all the others. It contains default method specifications and perform the GameLoop (input, update, draw).

Intro is the first screen displayed when running the game. It acts as a somewhat nice introduction.

MainMenu is where the player decides to start a new game or leave to desktop. This screen contains placeholders for "options" and "load game" buttons, altough they are currently not in use.

OptionsMenu is a class placeholder for a hypotetical future options screen. Currently unused.

NewGame is the character creation screen, where a name can be intruduced and a race and job can be selected. When starting the game, the player will be presented with a loading screen. This doesn't have its own class, as it is a sub-process located in the *Map* class.

Game is the screen that shows maps, the character avatar and where the player is more likely to spend time in. It contains a bunch of boolean variables for storing the on-map status.

Battle is the screen that shows and processes battles between the player and an enemy character.

GameOver is a simple screen shown when the player is defeated in battle, and leads to the *MainMenu*.

GameWon works like *GameOver*, but is shown when the *Black Dragon* is defeated, and leads to the *Game* screen for the player to continue playing for as long as they like.

Spell

These are spells that the player can cast during a battle. Casting a spell in battle, unlike using a potion, will cause the enemy to attack in response.
Every spell showcases different possibilities.

Spell is the base class. All of the remaining classes in this folder simply set their own attribute values, and override the *Cast* method specifying what the spell does.

Blizzard directly damages an enemy, inflicting 20 damage regardless of defense.

FireSword temporarily increments the attack power of the player, making them accumulatively inflict more melee damage until the battle ends.

Heal restores a máximum of 15 health to the player, but never exceeding the player's max_HP.

config.h

This single file acts as configuration for testing purposes. Note that a responsive game is not fully implemented and even though changing the resolution will not cause it to crash, some areas may not be properly displayed.

kNumMaps is used for testing and debugging purposes, when only the first N maps must be loaded in order to skip the whole loading process. Use this carefully as it may cause a game crash if the character steps on a *portal* leading to an undefined *Map*.

3. Strategies and Solutions to the Development Problems

Coming up next there's a list of some of the most remarkable development problems encountered and how they have been solved:

Working with different OSs

The development of this game has been almost entirely done on MacOs X, working with the g++ compiler. That ensures a fair share of compatibility problems and duplicated work, so the code was completely freed from propietary functions and, in return, it is now easily portable to other systems.

TMX Parsing

Perhaps the main problem faced by this development was the integration with maps created via the *Tiled* editor. Details for how this has been implemented can be found at the Map::LoadFromFile method.

Map loading time VS loading between maps

Taking into account that the Map::LoadFromFile method is considerably time-consuming, a decisión had to be made: either we load all the maps at once before starting the game, or we load the new map every time it is entered.
Finally, the first option was chosen, and a loading screen was coded within the same Map::LoadFromFile method, so the player knows that the game is working on it.

Extracting sub-images

Most of the images come in the form of spritesheets that need to be extracted and shown individually, so the Misc::GetSubImage method was created.
This method scans a portion of an image pixel by pixel and them creates a new image from memory with the stored information.

Cross-referenced headers

C++ tends to be very picky about the headers and their inclusion order. So the classes structure had to be carefully redesigned to avoid cross-referenced classes.

Grid structure

A Grid is used to store collisions, enemies, NPCs and portal on maps. This grid was originally defined as nodes of integers. It was later redesigned to store void pointers that directly indicate the object they store. In the case of collisions, all we need is a boolean indicating if there's a collision or not, so a pointer to the map itself is stored if there's a collision. All of the other nodes would be set to NULL.

Random enemies by map

In order to maintain a progressive difficulty system, player should only be attacked by enemy types that exist in the current map, so the enemy_id's are stored in *Map::enemies_pool_* while the map is initially scanned and then randomly selected from the pool.

References to objects as class variables

Many class variables are actually pointers to objects. This facilitates freeing memory as long as the class destructors are properly defined, as the successive destructors are called in cascade.

4. User Manual & Game Example

After starting the game and past the intro, the player will be presented with the main menú. From there, the application can be left or a new game can be started by clicking on the appropiate buttons.

When clicking on *New Game*, the player will be presented with the player creation screen.

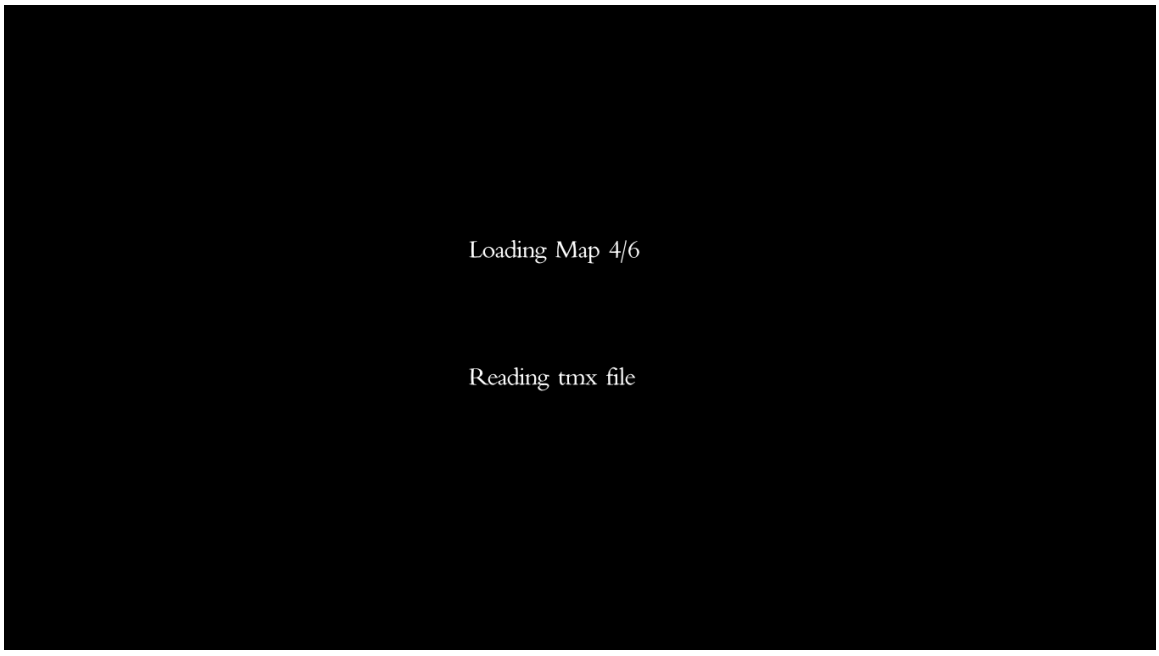
First, you must select a race by clicking on any of the faces on the left panel representing, from left to right: dwarves, elves, humans and orcs.

After doing that, the right panel will fill in with images portraying characters of that race,with different classes. From left to right: aristocrat, hunter, warrior and wizard.

Also, a few colored bars will appear on the left that indicate the base stats for the selected race. From top to bottom: health, mana, attack and defense.

Click on any of the right panel faces to select a character. Introduce a username and click *Start Game*.

The game now needs to load all of the maps from disk. Please be patient while this happens.



Now you're presented with the main game and the first map. You can use the arrow keys to move around. The movement on this map is discrete, meaning that a single tap on an arrow key will move the player a full tile on the grid.

Some NPCs can be talked to by stading around them. Do this if you want to learn more about the game story or interact with merchants.

There are some special tiles that lead to different maps. For example, by stepping onto any house door on this first map, you will be taken to a map where you can buy potions from a merchant. There are a couple of potions merchants throughout the game, so look around!
To trade with a merchant, stand in front of them and press 'B'. You will buy a healing potion until you run out of money. The conversation window will let you know when that happens.
Notice that you will exit the map to the same tile you entered through.

Also, some tiles contain enemies that must be fought when walked upon. This will trigger the Battle screen.

In this new screen, there are a few different actions that can be performed.

Attack causes both you and your foe to attack each other and cause meleee damage. The damage caused is proportional to your attack level minus the rival's defense level, and the other way aorund. This can lead to the situation where someone inflicts 0 damage to a foe. You need to level up if that happens!

Cast Spell opens/closes the spells window. This action can also be performed by pressing 'S'.
Each character has a range of available spells depending on their class. Wizards, of course, have them all. Hover the spell icon to read its description and its mana cost (bottom right).
Casting a spell will cause the enemy to attack, as if the player had attacked.
To close the spell window, click again on *Cast Spell* or press 'S'.

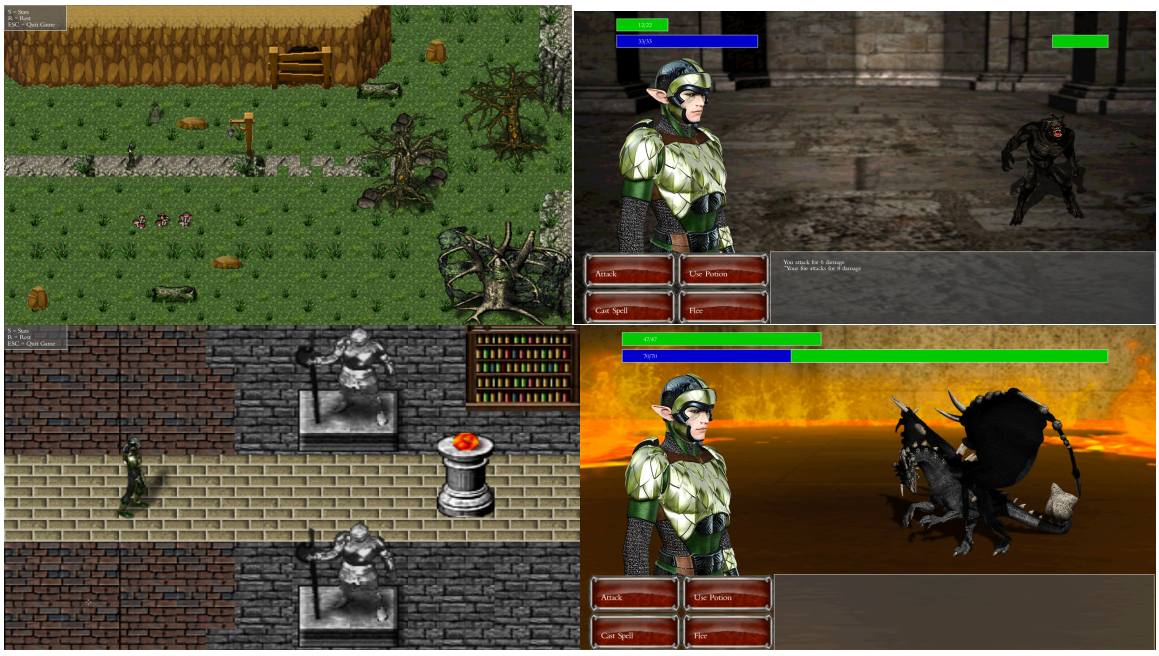
Use Potion will cause the player to use a healing potion and recover 15HP, given that the player has any potions left.
Unlike casting a spell, this will not cause the enemy to attack and can be done several times in a row.

The battle will continue until one of the contendants have an HP<=0, or the player flees.
Flee makes the player try to run away from battle. There's a 30% chance that you fail to flee and are attacked instead, though.

Back to the main screen and the map view, you can press 'S' to view your character stats.

Your can also press 'R' to rest. This can cause an enemy from the present map to attack you while you rest. Or you can be lucky and fully restore your HP and MP.

The objective of the game is to fight your way through the maps and reach the final battle with the oppresive monster terrorizing the Village of Pass.



The game can end in two ways: either the player is defeated in battle, or the final boss is defeated in battle. If the latter happens, the player will be allowed to continue playing and roam free for as long as he wishes.

Thanks for playing and good luck in your journeys!