Department of Informatics, Technical University of Munich Winter term 2019/20
Scientific Computing in Computer Science
G. Chourdakis, J.-M. Gallard, F. Simonis

Advanced Programming Tutorial
Project, Sprint X: Open-ended ideas

Sprint description

The game now includes all basic features. This open-ended phase provides some ideas to expand it and face other C++ challenges.

This worksheet provides a description of some possible recommended features. Together they form a nice package and will benefit from each other but you can implement only some of them independently if you prefer. The provided executable fulfills all previous user stories and implements these features. Of course the example executable is just one way of doing it and you are free to do it differently.

You can also try out other ideas and if they proved to be interesting to implement please tell us about it.

Assets

A recurring concept to expand many ideas is the use of assets and dynamic loading. An asset is a file containing information used by the game. It can be pictures, 3D models, sounds, or in our case text files containing some game informations.

An example of possible asset is one describing the enemies found in the game. This asset is dynamically loaded at the start of the game to set up an enemy manager that provides the enemy for the fight function or in quests. By using a human-readable format and with the right structure in the code, this asset can also be modified by a third party without needing to recompile the game and thus it simplifies the addition of new content (= modding the game), for example adding more enemies to encounter.

Using a standard data interchange format like JSON or XML for the assets simplifies their handling. The code loading the asset can use a factory pattern to initialize each elements described its corresponding asset file.

Improved file I/O: JSON

Goal: easier handling of I/O to files for the save/load and assets features

Currently the save and load feature write files using a custom format. To improve upon this a standard data interchange format like JSON or XML should be used. As they are not supported by the C++ standard library, it is recommended to use a third party library.

For simplicity we recommend JSON and the following JSON library: `https://github.com/nlohmann/json`. This library can be used with a single header include, provides enough features to fully utilize JSON and is well documented.

Refactor your serialize/deserialize methods to use a standard data interchange format. This should actually simplify your code and make the save file easily readable. It will also simplify the implementation of modding support in other features.

The following is an example of a save file written in JSON.

```
{
    "dayCount": 13,
    "player": {
        "HP": 165,
        "atk": 30,
        "def": 18,
        "level": 10,
        "mag": 60,
        "mana": 150,
        "maxHP": 165,
        "maxMana": 150,
        "name": "Bob",
        "type": "Archmage"
    }
}
```

More classes and the Diamond Problem

Goal: more RPG classes and dealing with the diamond problem.

Extend the amount of RPG classes available by allowing the “Warrior” and “Mage” classes (from sprint 3) to be further upgraded into new advanced classes that may or may not include new mechanics.

Also include a “Battlemage” class that both Mage and Warrior can upgrade into and that combines features of both: a Battlemage uses mana and magic and his attacks can critically strike. Thus this class inherits from both Warrior and Mage, C++ allows multiple inheritance. As both parent classes are child classes of the same “Trainee” class, this causes a Diamond Problem. For more information see `https://isocpp.org/wiki/faq/multiple-inheritance`.

This issue is solved using virtual inheritance. However this comes with some caveats to take into account like using `dynamic_cast` when downcasting. Furthermore the virtually inherited class will need to be initialized in all its child classes (even in “grandchild” classes), which can become annoying. There are ways to hide this by using an empty constructor in the virtually inherited class and initializing it in the constructor of the class that directly inherit from it.

Note that usually the base class virtually inherited in a multiple inheritance scenario is a pure virtual class (i.e. an interface) and thus most considerations with using virtual inheritance become irrelevant, making it easy to use to solve the Diamond Problem, our case here is a an especially nasty one.

Quests

Goal: richer game and dealing with complex data structures

One way to implement a quest mechanic is to implement quests as graphs to be traversed. The graph has a starting node and is then traversed from nodes to nodes following player actions, each node offering options to progress to another node, until the player dies or reach a node without links to further nodes (an end node). Each node contains some text to display and then optionally a fight and/or a reward and if it is not an end node one or multiple choices to continue the quest (i.e. links to other nodes). It is possible that different paths lead to the same outcome (for example multiple options to quit the quest all leading to a “you fled back to the village” end node) or lead back to a previous node, thus a graph is more appropriate than a tree.

To improve the feature, enable modding by making it so that a quest can be fully described in some text file format (e.g. JSON) and loaded at runtime by a quest engine, thus allowing users of your game to add their own quests to the game.

The following JSON shows an example of such implementation of a quest. Note that here enemies are described by an Id to be loaded by the `EnemyManager` (see Dynamic enemy loader) as well as the reward.

```
{
  "description": "Deal with the bandit camp. (Medium)",
  "startNodeId": 1,
  "nodes": [
    {
      "id": 1,
      "introText": "You reach the bandit camp.",
      "optionTexts": ["Charge!", "Wait for the night to sneak in."],
      "optionNodeIds": [10, 20]
    },
    {
      "id": 10,
      "introText": "A bandit reach for his sword, ready to fight!",
      "enemyId": 800,
      "postBattleText": "As you slay the bandit, you see a well
        equiped fighter coming toward you.",
      "optionTexts": ["Stand ready to fight.", "Flee!"],
      "optionNodeIds": [11, 101]
    },
    {
      "id": 11,
      "introText": "The bandit leader step forward to fight you!",
      "enemyId": 801,
      "postBattleText": "As the bandit leader falls, the rest flee.",
      "reward": 20,
      "optionTexts": ["Loot the camp and return home"],
      "optionNodeIds": [100]
    },
    {
      "id": 20,
      "introText": "A day passes. Under cover of night, you
        effortlessly slice the throat of the sleeping thieves.",
      "reward": 5,
      "optionTexts": ["Loot the camp and return home"],
      "optionNodeIds": [100]
    },
    {
      "id": 100,
      "introText": "You found some gold and a medium potion of
        vitalty. Your HP is increased by 5. You then return to the
        village",
      "reward": 31
    },
    {
      "id": 101,
      "introText": "You fled and returned to the village."
    }
  ]
}
```

Dynamic enemy loader

Goal: More enemy variety and creating a spawn table

Add a variety of enemies to the game, this is another case where you could use asset loading to facilitate the inclusion of new enemies and modding. Each enemy type should be identified by a unique id.

An `EnemyManager` class can be used to load up all the enemies. Having a factory to load up each enemy can be useful and you can reuse the code used to load the player data from a save file. The `EnemyManager` should offer a method to get an enemy by id (this feature can be very useful for quest). It should also provide a method to get an enemy to fight randomly depending on the player level for example, one way to facilitate this is to build a spawn table. A spawn table is a vector of vector where the inner vector represent the list of enemies that can appears (=spawn) level or group of levels. Some enemies might be marked to not appear in the spawn table and only be retrieved by their id (for example a quest's boss).

To increase diversity, enemies should use some advance classes too with mage enemies and warrior ones.

Further ideas

These features are not part of the provided example executable but would be nice game features if you want to go further.

Improved fight

Instead of having automatic fights, allow the player to chose attacks and/or use items during the fight. Each class would unlock different attacks, for example the mage might unlock different elemental magic attack (fireball, lightning, ice beam), and enemies might have weaknesses/resistances. Attacks might be locked behind a level requirement or stat requirement.

Also stats might be modified temporary during a fight by some attacks/items or effect.

Items and inventory

Implement an inventory for the player. Winning a fight or finishing a quest rewards some gold and possibly items. Gold can be spend at a merchant to buy items

Some items might be consumable while other are pieces of equipment. Pieces of equipment come with classes restrictions and equipment restriction (you can only wear one helmet at a time).

Localization / dialogue customization

Currently all text outputs are hard coded. This would make it difficult to offer multiple version of it, for example to support female player character or other languages. A good solution to it is to use asset loading and potentially a third party template engine (for example `https://mustache.github.io/`).

In the engine, instead of using `cout`, call a `localizationManager` that will load the string to output from one of many localization assets file (simplified by doing the JSON idea). The choice of which asset file to load depend on some dynamic parameters like the language or player gender. This way adding a new language to the game can easily be done.

Using a third party template engine like Mustache allows to put some logic in the assets themself. They are usually made to generate HTML web page but can be used to render customized string.