Space Invaders Report

Kristof De Middelaer
UA 2BINF

December 2013

1 Introduction

As an exercise in the correct use of certain design patterns we had to make game that resembled Space Invaders. In the following sections I'll explain and motivate my design. In particular my choice for a data driven system together with the use of the abstract factory and MVC/Observer pattern.

2 Usage

1. Navigate to the build folder: cd build
2. Use the command: cmake ..
3. Use the command: make install
4. Navigate to the bin folder: cd ../bin
5. Run the program with optional parameter: ./Space Invaders game-XML-FILE

The optional parameter is an xml file that describes a game. Such as Data/game1.xml and Data/game2.xml.

3 Implementation

3.1 Data driven system

Instead of immediately starting to program I chose to take the time to think about what I wanted to achieve with my version of space invaders. I read the articles provided in the assignment and found the "The Guerrilla Guide to Game Code" to be very informative. (`http://www.gamasutra.com/view/feature/2280/the_guerrilla_guide_to_game_code.php`)

I hadn't thought of the idea of a data driven system before. It opened up entire new possibilities that I thought were very nice. By using data instead of code to determine the properties of models the flexibility of my program increased enormously. I was able to build entirely different Space Invader games and gameplay experiences without touching my code. New types of enemy ships, playerships and bullets etc were easily made without recompiling or anything. Different ship formations, increased level difficulty,... all modifiable by creating new files/editing files.

This gives the developer (and players!) an easy way to create new levels and experiences.

3.1.1 Concrete

To achieve this I used XML files in a certain hierarchical way and parsed them using the TinyXML library.

There are two levels in the file hierarchy:

1. The XML gamefile that describes the actual game. Things like enemyship formation, amount of shields, level difficulty are in here. We also define what type of enemy ships, spaceships etc. should use. These are defined in the form of an XML filename.

2. The XML file that describes a certain model. e.g.: a file that describes the spaceship: its speed, hp, sprite,...

The DataParser and GameParser classes in the factories namespace both take care of parsing these XML files. They store the extracted information which can then be used to construct our game/entities.

3.2 Abstract Factory

I first read up about the Abstract Factory on the internet. I found a lot of different interpretations and implementations of this pattern but ended up with something I'm happy with.

AbstractFactory is the abstract class from which my Factory classes (e.g.: BulletFactory) are derived from. See figure 2.2.1 for an UML diagram.

The reason why I have three getController() methods is that for the constructing of certain MVC triples I need other parameters. This in itself is no problem, the getController() function gets overloaded and the correct method will automatically be chosen.

3.3 MVC-Observer

3.3.1 MVC

I read up about the MVC pattern and Observer pattern. Again, as expected, there were numerous interpretations and implementations. I ended up basing my MVC-Observer pattern off of the one described here: `http://www.laputan.org/pub/papers/POSA-MVC.pdf`

The MVC model:

- The model component encapsulates core data and functionality. It's independant of specific output representations or input behavior.
- The view component displays information to the user. It obtains the data from the model.
- The controller component recieves input. This input is translated to requests for the model or the view.

The only way the user can interact with the program is through the controllers. In my particular implementation controllers are driven by the user as well as the program itself. e.g.: the enemy ship controllers are driven by the program itself.

The most important class in my program is the Game class. It basically controls the game and holds information about the game such as the level etc. Important about this is that it contains no models or modelviews, it solely contains controllers as these should be the only way we can interact with our models and views.

One thing I would like to add is that my implementation isn't 100% 'clean'. I decided to add the sprites (and the mandatory textures) in the Model instead of the View class. The reason I chose for this is because editing the sprite's information was way more handy (and there were some very handy methods in sf::Sprite). if I would have put it in my View class I should've either: a) Made my View class accessible to my Models or b) keep information about the Model in a seperate format and change the sprite's info every time a change happens in our Model. (This is pretty much already implemented with the Observer pattern, it's used for other changes.)

In hindsight it might have been better to do it as said in b). This way I could've made SFMLViews and for example OpenGLViews without messing with my Models.

3.3.2 Observer

Both my controller and ModelView classes are derived from the Observer class. All models have a registry of Observers. Concretely, they have a vector of Observer pointers. Every time something in the model changes its notify() method will be called. This will iterate over the Observer registry and call the Observer's update() method. This way the Observer has been notified of a change in the model and can handle accordingly.

4 Game class

The Game class is the main class in the game. It holds information about the game such as the level etc. It holds a vector with smart pointers to all of our model controllers as well as a matrix of additional smart pointers to our enemy ships. The matrix is just used to easily determine what enemy ships are allowed to shoot. (Which should only be the bottom ones of each column.)

Its cycle() method performs one game cycle. This means that it will take care of the shooting/moving of all the modelControllers, checks for collision detection (and lets the models react when they collide with eachother), check if the game has ended or a new level needs to be started, delete irrelevant models that are either out of our gamefield or dead.

5 Documentation

Documentation has been generated through doxygen. By adjusting some settings UML diagrams are also generated and added to the documentation. In other words: all inheritance diagrams together with the documentation can be found here.