World War 3 Game Documentation

Table of Contents:

1) Project Description

The project is a 2D multiplayer, map-based, turn-based strategy game resembling a more complex version of Risk in the style of DEFCON (to minimize the work that must go into art assets). This game takes the form of a desktop application targetting the .NET Core (but actually using a downstream fork due to compatibility issues in the original version of the .NET core. Unlike in Risk, the regions to occupy are not countries, but smaller individual provinces. Some of these provinces contain cities, which are used for score computation and impact the resources the player gets from the province. In order to take control of one of these provinces, a player moves their army onto the space and ends their turn there. Players gather resources from their owned provinces at the end of each turn. Opposing armies that enter each other's zone of control attack each other, and the player must strategically maneuver their troops to secure victory. Each army has health, which is depleted in combat any time two armies end their turns close to each other. Armies that run out of health are removed from the game immediately. Like in Risk, the player's goal is to use their military to control as much territory as possible. A player's score is determined by the total value of the cities under their control, (the cities' real world population determines the score the city is worth). After 100 turns, the game will end, and the player with more provinces currently under their control will win.

2) Software Process

Our team used a modified version of the Extreme Programming model of software development. Before any coding was done, two initial design meetings took place. In the first of these meetings, we developed a unified vision of the general nature of the product and the work that would be required. To fit in with the project requirements, we turned all of these work items into user stories and sorted them by priority. Then, we divided these user stories into 6 work iterations, with the goal of having a usable deliverable at the end of each iteration with noticeable improvements over the last iteration. Another main goal was to have a relatively constant and realistically achievable amount of work for each iteration.

Although some of the specifics were changed over the course of development, these user stories helped provide a clear picture of where the project should be at each 2-week interval and what would be necessary to make that happen. At the start of each iteration, we reviewed any leftover work from the past iteration, added it to the list of the new iteration's work, and reprioritized based on the past iteration's velocity. Then, we divided team members into two pairs of 2 people and one group of 3. Each group was assigned an appropriate number of stories, with attempts made to assign work to the people best prepared to do that specific kind of development. Within each group, pair programming was used to help catch any small bugs early on, as well as so that at least two people would know where and how any given feature had been implemented.

This iterative development allowed for the team to effectively coordinate and parallelize work. In addition to that new development, each team was expected to do some amount of refactoring. The refactoring would take place in whatever areas of the code the team was already working on. Since the project did not include a huge number of files, this generally meant that every file would have some amount of refactoring performed on it. Also, on every iteration, one team would be assigned the task of general refactoring on the new code from the previous iteration. They would receive a smaller share of the new development work to keep development man hours generally consistent across groups.

While doing this refactoring and development, each group would also be responsible for, if applicable, writing tests. The tests would be done on the group's newly written code. To do so, we used C# and Visual Studio's built-in capabilities for unit testing. Ideally, each group would write at least one test for all non-GUI methods written, with more tests for more complicated methods. All tests were stored in a general Tests folder in the base of the project directory, with each set of related features having its own test class. These served as regression tests. All of the tests could be run as a package, helping to check that any new additions had not introduced any unforseen bugs into major features.

These test packages were run before each pull request was made to the upstream master repository. We used Git for version control, with one upstream master repository as the 'official' version of the project. Each team member also had their own forked version of this repository. To do work, a team member would update their repository to the latest version of the upstream master, then create a new work branch. They would complete their work on that branch, run all tests, then make a pull request to the master. Two team members were designated as code reviewers. These code reviewers would check over the submitted code, and either send it back with comments or approve it to be merged into the upstream master. Before the end-of-iteration demo, each team member would pull down the newest version of the upstream master to present their completed user stories.

3) Requirements & Specifications

a) User Stories

Iteration 1:

Expected Points	Actual Points	Description
2	2	Dump map to console
4	5	Create API between major game components
6	2	Create descriptive map with expected points of interest

Iteration 2:

Expected Points	Actual Points	Description
4	4	2 Players can now take turns making changes to the game state.
3	3	Users can now enter their moves via text input.
3	3	Let users view map via text interface.
3	3	Add persistent zone of control.

Iteration 3:

Expected Points	Actual Points	Description
4	3	Players own multiple armies and can move them across the map within their range every turn
1	1	Players can now end their turn and pass it to the next player
1	2	Armies are visible on map text interface
2	2	Cities generate resources and add them to their owner's stockpile
3	3	Armies that move onto a province can capture it for their owner

Iteration 4:

Expected Points	Actual Points	Description
3	4	Provinces are displayed as squares colored by their owner
1	1	Cities are displayed as smaller squares within each province
2	2	Armies are displayed as triangles of the same color as their owner
2	2	Armies can be made to capture a province, switching the color of that province to its owner
1	1	Resources are added passively to player stockpiles
1	1	Resources can be actively gathered by armies
1	1	Armies can be healed by consuming some resources

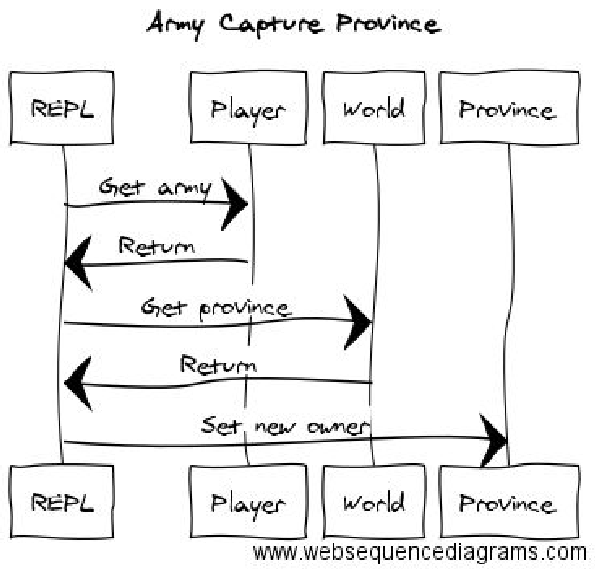
Iteration 5:

Expected Points	Actual Points	Description
2	2	Player gets points from controlled cities
3	2	Ability to create new armies
4	5	User keyboard and mouse input
2	4	Implement army combat
3	3	Render army information and action capabilities

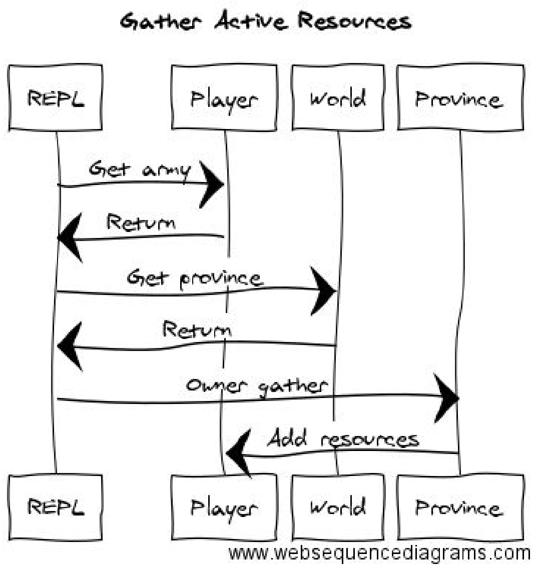
Iteration 6:

Expected Points	Actual Points	Description
2	2	Player starting conditions
2	2	Player victory conditions
5	6	Front-end for all actions and data
4	2	Text rendering
3	3	Saving and loading

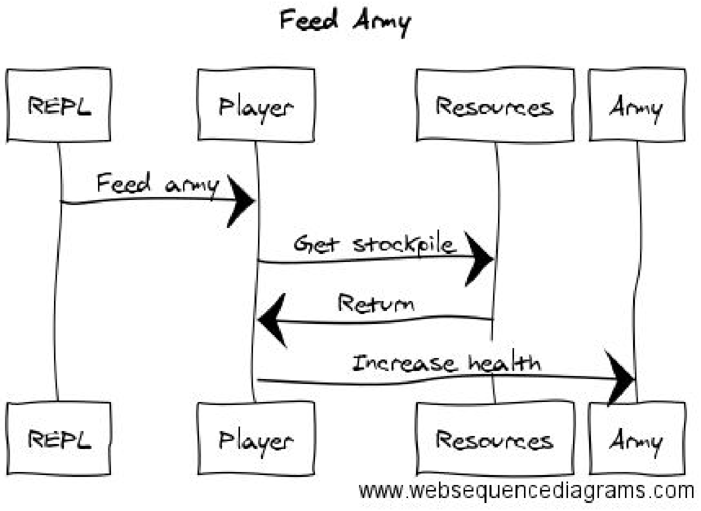
b) Use Cases



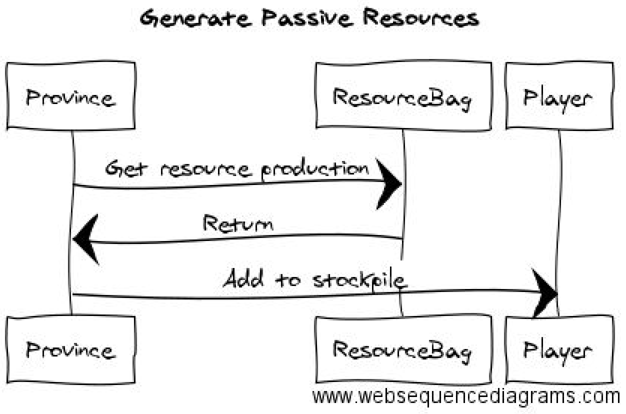
When an `Army` moves onto a new `Province` position, the game receives both objects and the new owner of the `Province` is set.



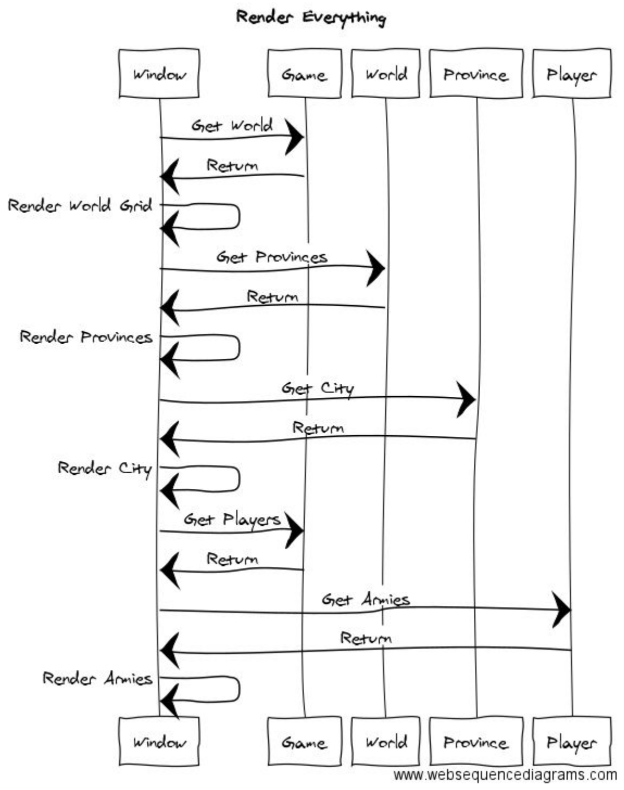
When the `REPL` gets a command to gather active resources, it gets each `Army` and the `Province` the `Army` is occupying. The `Player` object then receives resources from the `Province` object.



When the `REPL` receives a command to feed an `Army`, the `Player` object gets their `Resources`' stockpile and increase an `Army`'s health.



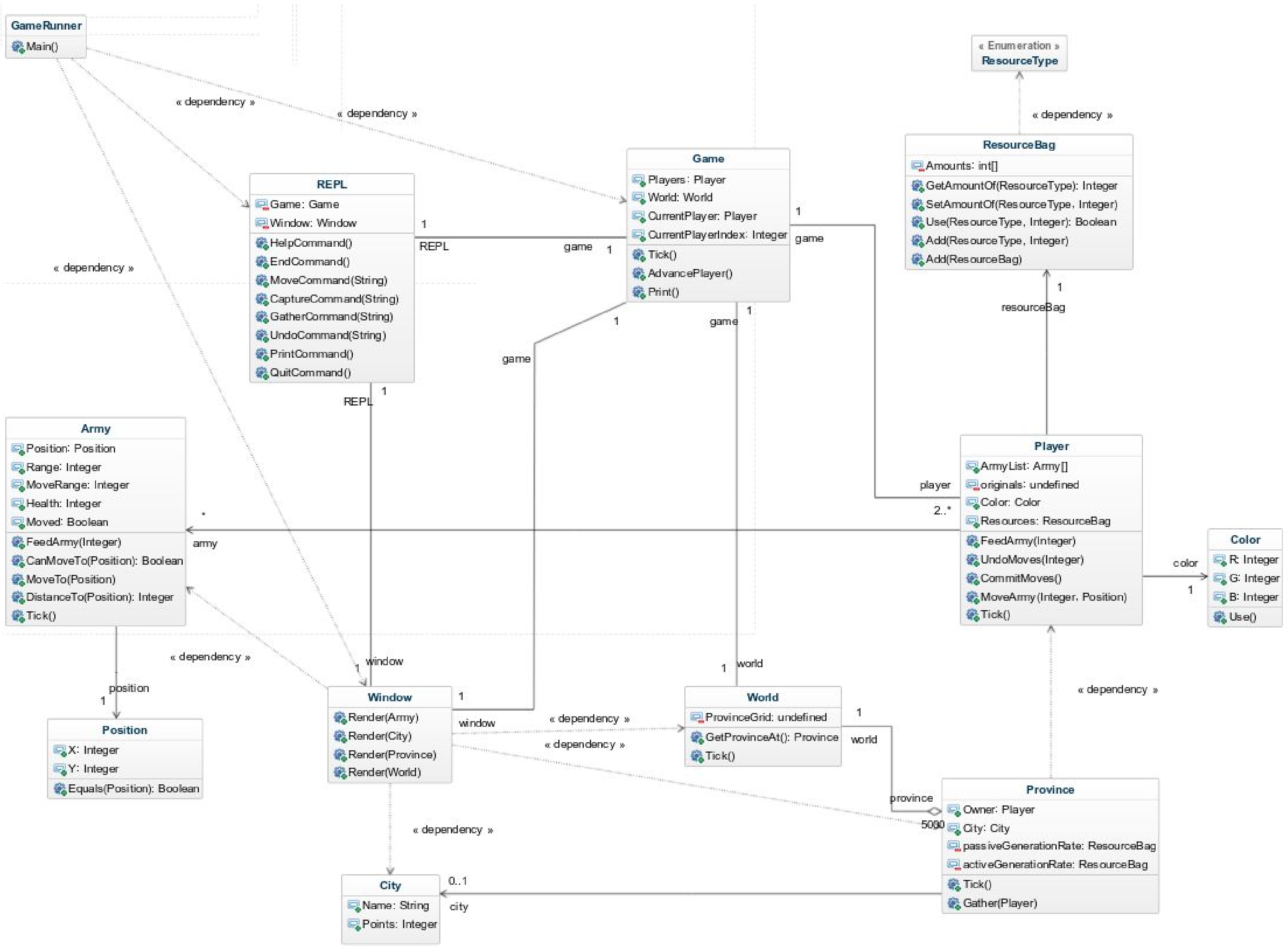
Passive resources are generated automatically every turn. Each `Province` owned by a `Player` gets its resource production from the `ResourceBag` and adds it to the `Player` object's stockpile.



The `Window` object gets all information on the `World`, `Provinces`, `Cities` and `Armies` and renders them to the screen.

4) Architecture and Design

a) UML Diagram



This is the general UML diagram showing all of the major classes within the game's structure. It also demonstrates which classes contain and interact with each other. Specifics are covered in the next section.

b) System Architecture

The game is launched through a `GameRunner` class, which starts a thread for the command line interface and a thread for the game window. The game loop is managed through the `Game` class. That class holds a `World` object, the `Player` objects, an `ArmyManager` object, as well as the logic for general functions like, saving, loading, and winning the game. A `World` represents the game map and contains an array of `Province` objects, each representing a square on the game board. Each `Province` object can contain cities, which provide resources and points when captured, armies, which can be used by players to fight each other and capture provinces, or may be empty.

The `Player` object represents an individual person playing the game, and has references to that player's controlled armies as well as their resources. It manages armies via the `ArmyManager` class, which contains the controlled armies as well as related logic. Resources are managed via the `ResourceBag` class, which likewise contains the player's available numbers of resources along with associated logic.

The game's GUI is managed via the `Window` class. On each frame, it renders the game's graphics, and checks for mouse input. It contains a reference to the main `Game` object, so that when input is received, the input can be passed along to the relevant components. Finally, the `REPL` class is the command-line analogue to the `Window` class. It accepts, parses and processes text input from the command line. Its main functions are to save the game and quit the game when the relevant commands are entered.

c) Influence of Frameworks

The two main frameworks used in the project's development were the C# language and the OpenTK graphical library. C# is a very object-oriented language, making our class structure well-defined. Because of this, we spent more time on the design process up-front before beginning the actual coding. While it made the initial iteration a bit slower, the clearer structure made coordinating the actual implementation easier than in other languages. We focused our design on making each class contain a single logical part of the program whose function could be generally understood via the class name. C# also helped in creation of the GUI due to easy multi-threading, and as noted above, its built-in unit testing functionality let us make one unit testing class for each logical piece of the program.

For the GUI, OpenTK served as a wrapper for the commonly-used OpenGL framework. It gave native low-level access from our C# code to OpenGL functionality. The low-level nature of the framework had several consequences for the system's design. Generally speaking, it made development initially take longer than it would have with a higher-level framework, but it allowed for time-saving later on due to customizability that would have been lost otherwise. For example, when rendering the game, we initially had internal confusion over whether the pixel size would be static or flexible. When we decided on static, changing over the flexible systems just took some manipulation of the existing pixel coordinate code, rather than a complete restructuring of the project. However, OpenTK did influence us towards keeping all of the rendering functionality in a single class, making the `Window` class overly long and hard to find specific code in.

5) Reflections

miranti2: Jumping right into coding without doing much design up front allowed us to get started and making visible progress immediately but I feel our design ended up hurting for it in the end. If I were to do this project again, I'd require us to do a few more design meetings.

mladeno2: I learned that .NET Core and .NET Framework are binary incompatible, and that using downstream forks of .NET Framework NuGet packages patched for .NET Core can lead to all sorts of headaches.

aghosh11: I learned that it is difficult to reconcile OpenGL's global state API with our game's object-oriented structure as well as a really useful but complicated way to maintain a linear git history by squashing commits and rebasing

raberns2: I learned quite a bit about the way that version control should be used for coordinating a small team project. Specifically, having each team member fork their own version of the repository and create a branch for each feature made the project much more error-resistant. Similarly, I learned the value of having dedicated code reviewers and style guidelines. While annoying at first, they kept the codebase from becoming an unreadable mess as tends to happen by the end of large class projects. Overall, our focus on following stricter software process guidelines made for a much better group coding experience than most of my other classes.

weislow2: I learned about eventHandlers through having to program interactions with the game GUI as well as openGL, which led to a new respect for video game/engine development. I spent ages trying to find different methods and solutions to make user interaction work with the game library we were using on multiple platforms.

jlee164: I learned a lot about Git flow, C# and .NET core. I learned that C# is very similar to Java, one of my favorite programming languages. I also learned that OpenTK was not very compatible with Mac's because I could not get the right coordinates of my clicks. Another thing I learned was how to handle mouse clicks and keyboard presses.