

**Department of Computer Science**

**Department of Multimedia**

Multimedia - Game Programming

**Dmytro Maretskyi**

S19340

**Turn-based multiplayer strategy game**

Engineering thesis

Thesis supervisor

Warsaw, PJATK, 06, 2022

Table of Contents

[1. Introduction 3](#_Toc75360009)

[2. Project description 4](#_Toc75360010)

[2.1. Definitions 4](#_Toc75360011)

[2.2. Game description 4](#_Toc75360012)

[2.3. Multiplayer aspects 5](#_Toc75360013)

[2.4. Game visuals 5](#_Toc75360014)

[3. Project architecture 6](#_Toc75360015)

[3.1. Overview 6](#_Toc75360016)

[3.2. C# build pipeline 6](#_Toc75360017)

[3.3. Unit testing 8](#_Toc75360018)

[3.4. Database schema 8](#_Toc75360019)

[3.5. Class diagram 8](#_Toc75360020)

[3.6. Deployment 8](#_Toc75360021)

[4. Game state 9](#_Toc75360022)

[4.1. Game state data structure 9](#_Toc75360023)

[4.2. Updating the game state with new move 9](#_Toc75360024)

[4.3. Cycle search algorithm 9](#_Toc75360025)

[4.4. Encircled unit search algorithm 9](#_Toc75360026)

[5. Game rendering 10](#_Toc75360027)

[5.1. Board rendering 10](#_Toc75360028)

[5.2. Captured area rendering 10](#_Toc75360029)

[6. Client-server interaction 11](#_Toc75360030)

[6.1. WebSockets API 11](#_Toc75360031)

[6.2. Making a move 11](#_Toc75360032)

[6.3. Backend server 11](#_Toc75360033)

[6.4. Database 11](#_Toc75360034)

[6.5. WebSockets API 11](#_Toc75360035)

[6.6. Authentication? 11](#_Toc75360036)

[6.7. ELO ranking 11](#_Toc75360037)

[6.8. Matchmaking 11](#_Toc75360038)

[7. Conclusion 12](#_Toc75360039)

[8. Bibliography 12](#_Toc75360040)

## Introduction

Inform about topic -> What is this thesis/diploma about ...

(Don't write it at the beginning (final version)

Explain …

List of goals:

(THIS part is written BEFORE You will start...) -> It defines how You wish to proceed...

(Do not give any answers to your problems)....

#KEYWORDS!!!!

List of technologies

## Project description

### Definitions

Game - …

Turn based game - …

Strategy game - ..

Multiplayer - …

Matchmaking - …

Player ranking - ...

### Game description

Chart, scatter chart

Description automatically generatedThe game is played between 2 players on a rectangular grid, usually 25 by 25 cells, but other sizes are also supported. Each turn a player places his unit (also called “dot”) on a vacant grid cell. All units are the identical and only differentiated by the color, signifying to which player they belong. If, after placement, by connecting player’s adjacent dots (either orthogonally or diagonally) a cycle can be formed that also surrounds one or more opponent’s dots, the player captures the opponent’s units.

Figure 1. Red units captured a blue unit.

Each captured dot grants the player 1 point, as well as effectively “disables” the enemy uint - it can no longer participate in forming cycles needed for capture. The goal of the game is to get more points then your opponent. The game is played until either one of the players resigns or there are no legal moves left.

### Multiplayer aspects

The game should also provide rich multiplayer aspects by having an automatic matchmaking system, where you can queue up to have a game with the next available player.

There also will be a ranking system to give each player a score that would describe their skill level. Ranking would be calculated based on the results of game’s played and updated according to the ELO system[1], similar to the ones used in online chess games. The rankings would also be used to match opponents with the closest skill levels during matchmaking.

### Game visuals

The two players are identified by their color: player 1 - red, player 2 - blue. The dots are rendered on cell grid intersections and match the color of the player who placed them.

Captured areas are denoted by connecting the dots that participated in the enclosure with a line and adding a semi-transparent fill over the enclosed area with a matching color.

Q: Should I include the game screenshots here?

## Project architecture

### Overview

The project can be split among two parts: the game client that uses Unity and is written in C#, and a backend server that maintains the game state. Between them there are several shared classes such as data models describing the game state and the moves made by players. The high level overview of the project structure is presented on Figure 2.

The backend server hosts a WebSocket API that game clients connect to. Choosing WebSockets as a transport layer provides us with an ability for two-way communication: game server can push updates to the clients and clients can send moves made by players. Also this increases the flexibility for the backend deployment as almost all hosting providers support WebSockets.

The backend server has a PostgreSQL database associated with it. It was chosen because relational databases are well supported by mainstream C# data-persistence frameworks, such as Entity Framework.

There’s also an additional test module hosting automated unit-tests for the core data models and algorithms.

Diagram

Description automatically generated

Figure 2. Project overview. TODO: Add the test module.

### C# build pipeline

One of the difficulties when sharing code with a Unity project is that Unity does not support multi-assembly builds. Meaning that all C# sources must be a part of a single C# project when compiling the game client. Also, C# project and solution files are auto generated, which means making any manual changes are unfeasible, as they would be lost next time Unity Editor is refreshed.

This poses a challenge, because a usual way to share code between the backend server and a client would be to introduce a third class-library project with shared classes, that two other projects depend upon. Since that’s not possible the only solution is two have two parallel project structures: one for the backend and the second one for the Unity build. The shared sources would be included in both structures and compiled separately.

Unity requires all C# sources to be placed inside “Assets” directory. This also includes the shared classes. This would force us to move away from then canonical project structure where each module resides in a separate directory on the top level. There were numerous attempts to go back to the flat structure utilizing filesystem symlinks. Unfortunately, Unity as well as C# build tools do not have good enough support for them, and often they were not recognized as a link to another directory.

Ultimately the following project structure was settled on:

* Backend project directory.
  + Backend “csproj” file.
  + Backend sources.
* Test project directory.
  + Test “csproj” file.
  + Test sources.
* Unity project directory.
  + Assets directory.
    - Unity assests.
    - Core project directory (shared sources).
      * Core “csproj” file.
  + Unity autogenerated “csproj” file.
  + Unity autogenerated solution file.
* Solution file for the backend and core projects.

### Dependency management

It wouldn’t make sense for every project to re-implement from scratch common standardized components such as serialization, networking and others. So, to speed-up and ease development we often rely on existing solutions and already developed libraries and frameworks. Initially, and often in older programming languages like C and C++, developers would copy the library into the source tree of the project.

This approach poses many problems, among which is difficulty of installation, as it must be performed manually. And the difficulty of configuration as there’s no enforced standard for how the reusable library code should be structured. Upgradability is also an issue: when a new version of the dependency gets released, developers would need to perform the same steps of copying the library code into their project’s codebase and configuring the compilation pipeline. The process gets even more difficult when transitive dependencies come into play. Transitive dependencies are dependencies if the library or package your project depends on directly.

To solve those issues package managers were created. Package manager is a tool responsible for installing and upgrading project dependencies and their transitive dependencies. Code is typically structured into packages that are identified by their unique name. Packages are versioned and the project’s manifest specifies what version of the package it requests to be installed. The most common versioning schema is “SemVer” (semantic versioning) [add reference].

SemVer versions consist of three integer parts: major, minor, and patch number. For example, 1.0.12 – would be a valid SemVer version. Each part has its own meaning to represent the type of changes that were introduced. The major number is incremented when breaking changes were introduced – changes that, after upgrade, would break existing code that depended on the previous version of that package. The minor version is incremented when new features were introduced, but in a backwards-compatible way, where it wouldn’t break the existing code after an upgrade. The patch number is reserved for bugfixes.

This type of versioning scheme allows developers to specify not concrete versions of their dependencies, but whole ranges of versions they are compatible with. Most commonly, developers allow minor and patch numbers to change up to the next major release. 2.4.12 <= x < 3.0.0 would be an example of such a range.

Such standards allow packages to be distributed easily and often package managers come with their own package repositories, where developers can publish their packages and others can download and install dependencies with a package manager. The installation usually is as easy as issuing a command to the package manager, to install a dependency, by specifying its name and the requested version. In such a setup, only the manifest file is committed to the VCS, and package files are excluded from source control, as every developer can easily recover them from the manifest using a package manager.

The de-facto package manager for C# (and the whole .NET ecosystem) is NuGet [add reference]. It comes with its own package repository hosted on <https://www.nuget.org/>. As of the time of writing, the repository contains over 280 thousand packages. Packages are versioned using SemVer.

This project uses NuGet to manage dependencies for the backend server. Among which is the JSON [add reference] serialization library and web-socket networking library. For Unity, a different package manager is used: Unity Package Manager (UPM for short).

UPM has a package distribution method. NuGet distributes packages in their compiled form: as a DLL (Dynamically Linked Library) [add reference]. This has the advantage of the source code already being compiled, which avoids any compilation errors in packages and reduces the size of installed dependencies. There’s however a disadvantage: assets including textures, models and prefabs cannot be distributed in such a way. UPM installs packages with their C# source code and assets. The package code is typically included in VCS.

The UPM ecosystem is quite young and there are often packages missing. While there’s no official support for NuGet in unity, there’s an open-source unofficial plugin to replicate NuGet package manager behaviour for Unity projects: https://github.com/GlitchEnzo/NuGetForUnity.

### Unit testing

The game state and associated algorithms outlined in the section 4 of the work can get quite complex and have many edge cases. Testing them manually is a very time-consuming and error-prone process. For those reasons an automated test suite is used to verify the correctness of the algorithms. A unit test suite is contained in the “CoreTest” submodule. It consists of a series of functions each executing a piece of game logic following a specific scenario and asserting that the output matches the expectation. The tests might contain general scenarios such as modifying game state by making moves or test concrete functions such as querying game state for cycles in the graph of units.

Since unit tests are set-up as individual methods, a test-runner is needed to orchestrate the test execution. It should execute each test methods, gathering the results or handling any thrown exception and then report the test execution result. NUnit is used for that purpose. It is widely adopted in the .NET ecosystem. Has IDE integrations and supports running subsets of tests or running individual tests with debugger enabled.

[TODO: Add illutaration of tests results in Rider]

Every unit test usually consists of three sections:

* Assemble – setting up the initial state of the tested module. For testing state transitions it might be the state before making a mutation.
* Act – run the code which is being tested. For example: make a move on the game board.
* Assert – assert that the result of the executions matches your expectations.

Graphical user interface, text, application, email

Description automatically generated

Figure 3. Example of a unit test.

In the unit test displayed on the Figure 3 the three stages are apparent. The “assemble” stage sets up the game state from a predefined scenario. In this case it’s a state just before a capture. Next step is to make a move by calling the API. The move is for the Red Player to place the final unit completing the cycle. This will mutate the board state in reaction to the move. In the final stage we assert that the game state after the move has recorded that capture.

To achieve good impact from unit-tests it’s important to have them cover edge cases and different branches in code. This might involve

### Diagram Description automatically generatedDatabase schema

Figure 4. Database schema.

### Class diagram

### Deployment

TODO: digital ocean or heroku

## Game state

### Game state data structure

The game state consists of a 2-dimensional array describing each grid cell position as well as a list of captures that were performed during the game.

Each cell state consists of:

* Vacancy flag - whether this cell was claimed by the player.
* Their player that claimed this cell - only in case this cell is claimed.
* Captured flag - whether the dot placed in this cell is captured by the opponent.

Each capture entry in the list includes a player that made the capture as well as the list of grid coordinates of dots that participated in the enclosure of enemy units.

TODO: C# code sninppets.

### Updating the game state with new move

With every move we perform a cycle search to find new enclosed areas. Dots placed on the square grid form an effective graph with their neighbors. A depth-first search algorithm is used to find cycles in this graph. Cycles are also validated to not be composed from multiple larger cycles.

After the cycle is found a variant of flood-fill algorithm is used to select all grid coordinates that are enclosed by this cycle. Then, if any of those contain enemy units, those units are marked captured, and the cycle is added to “captures” list.

### Cycle search algorithm

Depth first search is used to find cycles in the graph. Search is always started from the dot placed on the current move. This optimization can be made because all cycles that were formed without the dot placed on the current move were already discovered when running the algorithm on previous moves.

To implement the algorithm a recursive approach is used with a stack to store a sequence of coordinates being currently processed. First a recursive function is executed on the starting point - the dot placed on the current move. It pushes the coordinates to the stack and then executes itself on all neighbors that can participate in capture. If a valid cycle is found, i.e. we reached the starting point, the sequence of points is stored as one of the cycle candidates. After all of the neighbors have been processed the current position is popped from the stack, and the execution flow is returned to the caller.

If during the execution we reach a neighbor of any point that already in the stack, but not the initial one, we terminate early, because that cycle could have been formed without the initial dot. And hence, was already processed on previous moves.

TODO: Union find algorithm: <https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf>

### Encircled unit search algorithm

A modified version of flood-fill algorithm

## Game rendering

### Board rendering

### Captured area rendering

With sprite-shape controller

## Client-server interaction

### WebSockets API

Rationale for using web sockets.

API reference: JSON RPC

### Making a move

Describe what path a move makes, from being created on the client, registered on the backend and then game state synchronized to the client.

### Backend server

### Database

### WebSockets API

### Authentication?

### ELO ranking

### Matchmaking

Show Your Struggle... and knowledge You had to get...

Theory - There is a knowledge You will have to accumulate... Explain it on Your OWN... (give good statement from where You got this knowledge... footnote.. ...page 47-85)

(You got all the pieces together here)...

Practice

You invoke knowledge form thery presenting Your SOLUTION...

(Always ask 2 questions... Why... and ... where.....)....

You do think about something.... Is it important for my work... for it's explanation...

Choose something for structure... Start general... then go to details/ or do opposite...

------------------------------------------------------------------------------------

## Conclusion

Answer Introduction...

- You where defining problem.. You where defining possible approaches You wish to follow...

Look at ALL YOUR SOLUTION (Main body)... tell what is the result... Summarize effects...

What Do You think about it....

Future improvements

------------------------------------------------------------------------------------

## Bibliography

1. [*"About the USCF"*](http://www.uschess.org/about/about.php). United States Chess Federation http://www.uschess.org/index.php/About-USCF/*. Retrieved 2008-11-10*.