

**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](#_Toc90918592)

[2. Project description 4](#_Toc90918593)

[2.1. Definitions 4](#_Toc90918594)

[2.2. Game description 4](#_Toc90918595)

[2.3. Multiplayer aspects 5](#_Toc90918596)

[2.4. Game visuals 5](#_Toc90918597)

[3. Project architecture 6](#_Toc90918598)

[3.1. Overview 6](#_Toc90918599)

[3.2. C# build pipeline 6](#_Toc90918600)

[3.3. Dependency management 7](#_Toc90918601)

[3.4. Unit testing 8](#_Toc90918602)

[3.5. Class diagram 9](#_Toc90918603)

[3.6. Deployment and hosting 10](#_Toc90918604)

[4. Game state 11](#_Toc90918605)

[4.1. Game state data structure 11](#_Toc90918606)

[4.2. Updating the game state with new move 11](#_Toc90918607)

[4.3. Representing cycles 12](#_Toc90918608)

[4.4. Cycle search algorithm 14](#_Toc90918609)

[4.5. Encircled unit search algorithm 16](#_Toc90918610)

[4.6. Iterator methods in C# 16](#_Toc90918611)

[5. Game rendering 17](#_Toc90918612)

[5.1. Board rendering 17](#_Toc90918613)

[5.2. Captured area rendering 17](#_Toc90918614)

[6. Client-server interaction 19](#_Toc90918615)

[6.1. Overview 19](#_Toc90918616)

[6.2. Web-sockets 19](#_Toc90918617)

[6.3. JSON-RPC 19](#_Toc90918618)

[6.4. Backend server 23](#_Toc90918619)

[6.5. WebSockets API 24](#_Toc90918620)

[6.6. Making a move 24](#_Toc90918621)

[7. Conclusion 24](#_Toc90918622)

[8. Bibliography 25](#_Toc90918623)

## 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 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 describing many different test cases, which increases the amount of code used for testing. Maintaining a large amount of test code can cause problems, so it’s important to optimize the test code size as much as possible.

In the example above a utility function is introduced – “ParseBoardState”. It is used to create an initial game state with a minimal amount of code. The way it is achieved is by parsing board state for a string. This also has a benefit of providing good visualization of the board state. The string format consists of a rectangular grid of characters separated by spaces. Each character corresponds to a cell on the grid. Grid size is determined automatically from the input string: by measuring the number of lines, and the number of characters in the first line. Only “.”, “R”, and “B” characters are allowed, specifying empty cell, cell occupied by the red player, and a cell occupied by the blue player respectively.

“TestUtils” class is used to host aforementioned and other utility functions. A different function can be used to generate a sequence of grid coordinates which is used for testing algorithms related to cycle search. In that case the similar string format is used, but instead grid cells have numbers which represent indexes of grid cells in the resulting array.

### Class diagram

### Deployment and hosting

A couple of factors were taking into account when choosing a hosting platform:

* Support for the chosen architecture – ultimately the hosting platform would need to be able to run a .NET Core binary which backend server is compiled to.
* Flexibility – hosting platforms vary in the amount of control they give to the users. It can range from hosting lambda-functions [add reference] where users define individual request handlers and don’t have any much control, to dedicated servers where a bare-metal machine is provide and users can configure anything starting from the installed OS.
* Cost
* Ease of use – the platform should be easy to manage and not pose any roadblocks. This usually correlates inversely with flexibility as the most flexible systems require the most configuration. Although trying to fit a system into a hosting platform full of incompatible restriction could be a challenging or even impossible task.

With that in mind, three hosting providers were examined:

* Hetzner – they provide dedicated servers. A wide selection of operating systems can be installed. Installation can take upwards of 24 hours after an order is made. An installation fee is also taken on every order. This is by far the most flexible hosting solution, although it comes with a price of management overhead and higher cost.
* Digital ocean – they provide VPS (Virtual Private Server) where users get a virtual machine on a shared dedicated server. Users can pick the amount of resources they need allocated to their VPS: CPU, RAM, disk space. A machine can be ordered from their web UI. Installation is automatic and the server is usually ready within minutes. Costs start at $5 per month for the most basic configuration.
* Heroku – they provide servers but rather provide hosting for apps. There are a number of tech stacks they support, including Node.JS, .NET Core, and Ruby on Rails. Users don’t have control over the machine used to run the app. The app is provided with the configuration parameters to use, such as the port number for it to listen. A number of optional services can be also added, for example a PostgreSQL database. Interestingly, they offer a free plan where the hosting is free-of-charge but the app networking throughput is limited and the app is put to sleep when it is not used continuously. The paid plans start at $5 per month.

While Heroku was considered as a strong candidate, the amount of flexibility having a separate server offers was a deal-breaker, especially considering that the pricing was similar. Using a free plan on Heroku would significantly hinder the game performance, as the backend server would be put to sleep from time to time.

In the end, Digital Ocean was chosen as a hosting platform, running an Ubuntu operating system. Provisioning can be done through SSH. The server’s firewall is configured to accept TCP connections on a provided port. And a .NET Core runtime is installed to run the game server binary.

## 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.

A picture containing graphical user interface

Description automatically generatedEach 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.

Figure 4. Board state

### 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.

### Representing cycles

One of the core data-structures used by the algorithm is a cycle. Cycle is closed sequence of distinct points on a grid. Each point must be adjacent to the previous and next points in the cycle either horizontally or diagonally. In-memory cycles are represented as arrays of grid coordinates (row and column).

There are many ways that a single set of points can be ordered in an array: different points can be chosen as the begging of the cycle, and the enumeration direction can be either clockwise or counter-clockwise. This makes it difficult to perform equality comparisons, as the algorithm would need to account for those variations. In the same way, it makes it difficult to build a stable hash function, which would be useful for performing efficient lookups in “Set” or “Map” data structures.

For those reasons cycles are always stored normalized. Normalized cycles must have their points ordered clockwise. And the starting point must be the one with the smallest row coordinate. If there are multiple points on the lowest row, the one with the smallest column coordinate is taken. As a reminder, rows and columns are counted from the bottom-left corner of the game board, starting from (0; 0). Figure 6 displays a normalized cycle. Numbers in points depict the index of said point in the internal array representing the cycle.

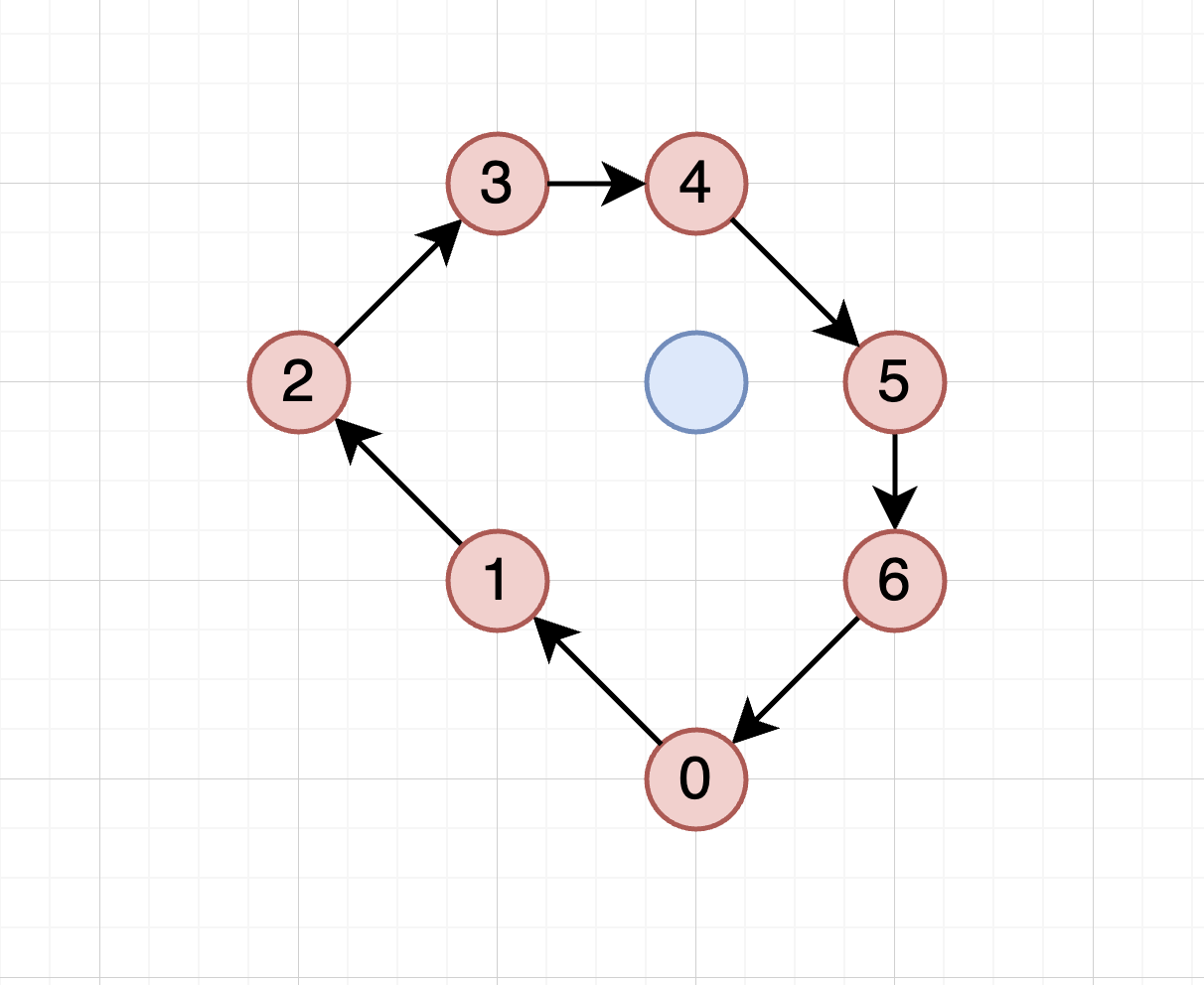


Figure 5. A cycle of 7 red points around a single blue one.

Normalization is performed using “Normalize” method in the “Cycle” class. It will mutate the instance it is called upon to have it be normalized. Normalization is done in two steps:

* Determining the current enumeration direction (clockwise or counter-clockwise) and reversing the array if the direction is counter-clockwise.
* Searching for the minimal point in the array (using aforementioned comparison rules) and rotating array elements so that the minimal point is at the index 0.

For the first stage, while reversing an array is easy using the standard library function, determining an enumeration direction can prove tricky. An operation similar to vector cross-product is used. Algorithm is as follows:

1. For each point in cycle, consider a vector pointing from the previous point in the cycle to the current one, and a vector pointing the current point to the next one in the cycle.
2. Normalize the two vectors so their lengths are equal to 1. This is required because diagonal connections will have a length of , while horizontal ones will have a length of 1. For the next steps it is important that all vectors are of equal length.
3. Extending the vector space to the third dimension, take a cross product of two vectors: . This vector is guaranteed to be parallel to the third-dimension basis.
4. The z coordinate (third dimension) of that vector will be positive if the curvature in that point of the cycle is clockwise, negative if it is counter-clockwise, and 0 if the line connecting cycle points is straight in that spot. Refer to Figure 7 for more details.
5. Summing up z coordinates calculated for each point in the cycle, we get a total metric for the entire cycle. It will be a positive number if ordering in the cycle is done counter-clockwise, and negative otherwise. This algorithm will also be resistant to any local fluctuations in the cycle boundary curvature, such as when the boundary shape is concave.

Chart, bubble chart

Description automatically generated

Figure 6. Examples of different curvatures at a given point in cycle boundary.

After obtaining the metric for the whole cycle it is just the matter of reversing the array if the current ordering is counter-clockwise. Next step in normalization is positioning the minimum point at the index 0. This is achieved by first, finding the current index of the minimum point by linearly searching the whole array. And second, copying all of the points to a newly allocated array of the same size. Copying is performed with an offset equal to the index if the minimum point, with indexes being calculated modulo array size. This could be done more efficiently by performing a series of swaps in-place, which would allow to get rid of extra array allocation, but performance boost gained is not significant in the context of this application.

There’s another, implicit, assumption: all cycles must not be self-intersecting. Meaning the only points allowed to be adjacent are the ones directly following each other in the cycle. The cycles that are self-intersecting can be split into two or more different cycles with some common points. This is not enforced in the “Cycle” class directly, but the following algorithms only operate on non-self-intersecting cycles. And self-intersecting ones are discarded immediately.

### 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.

Text

Description automatically generated

Figure 7. Cycle search algorithm

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

### Encircled unit search algorithm

When we detect a cycle, we can search all points inside that cycle to find out whether a capture was performed. To make the search a way to enumerated all points contained in the area bounded by the cycle is need. First a single point known to be inside that area is taken, and then a flood-fill algorithm is used to enumerate all points within that area. Each point is then examined to determine if it’s valid for capture.

The first step, calculating the starting point for flood-fill in the area bounded by the cycle, can be achieved using the same reasoning as for cycle normalization. We already defined a function to examine relative direction between neighbouring points. Assuming all cycles are normalized, their points must be enumerated clockwise. This means that, relative to the direction of the boundary, points inside the region would be to the right side, and outside the boundary – to the left. Picking a point on the cycle’s boundary and examining all of its neighbours to satisfy above conditions allows us to calculate the starting point for the flood-fill algorithm.

The flood-fill itself is an algorithm that recursively visits all neighbouring points from a given starting position. [add flood-fill reference] It is implemented using a queue and an array to mark visited points. Queue contains a list of points to be visited. It is bootstrapped with the initial point. The algorithm goes as follows:

1. Pop the point from the front of the queue. If the queue is empty algorithm is finished.
2. Set the visited flag for that point.
3. Perform any operations meant for visited points. In this case the point is checked for any potentially capturable units.
4. Enumerate all neighbours of the point. If they weren’t visited already, and satisfy conditions for flood-fill (not part of cycle boundary), push them to the back of the queue.
5. Go back to step 1 and repeat the algorithm until the queue is empty.

Figure 9 shows the C# code for the algorithm.

### Iterator methods in C#

In C# programming language there’s a feature called iterator methods. [add link to iterator methods docs] This is a special case of a method that instead of returning a single value, can generate a possibly infinite sequence of values. This is done by specifying the return type of that method as “IEnumerable<T>” and using “yield return” keyword. In this case the execution of the method will continue past the return statement, potentially allowing it to return multiple values. In practice this is achieved by returning a generator object, containing a suspendible version of the method code. This object can be repeatedly queried, for example using a “foreach” loop, to produce the next value in the sequence. Each query will run the portion of the method code up to the next “yield return” keyword. Then the execution will be suspended and method scope variables will be serialized.

There are numerous applications for such construct, but in this work it is used to optimize algorithms based on sequence manipulation. Consider this: many steps in the previously described algorithms can be considered either acting on a sequence of values. To find what units have been captured after a move we need to consider points in all new captured regions. To get all new captured regions, we first iterate over all cycles created by the newly placed point. To get all cycles we recursively iterate over neighbours of the starting point. Those seem like logical boundaries by which the algorithm can be decomposed into different functions. The problem here is that many of those functions return sequences or sets of values: getting neighbours of a given point returns a set of points. Enumerating cycles formed around a given point returns a set of cycles. Applying a flood-fill algorithm returns a set of points inside a given cycle. Using dynamically allocated containers such is array or Set here would have a drastic impact on performance, as there would be a big overhead in memory allocation. Also the entire resulting set would need to be pre-computed and stored in memory before returning from the function. This is often not needed, as the resulting elements are consumed one-by-one and access to the whole set at a time is not required.

Graphical user interface, text, application, email

Description automatically generated

Figure 8. Example of an iterator method.

This is where iterator methods are useful. They allow us to use the same principles for code decomposition, without sacrificing on performance. Figure 9 shows an example. This method enumerates all points inside a boundary of a given cycle using a flood-fill algorithm. “GetDirectNeighbours” is also an iterator method. Here it is consumed by a “foreach” loop.

## Game rendering

### Board rendering

### Captured area rendering

With sprite-shape controller

## Client-server interaction

### Overview

Networking model is built around client-server architecture with JSON RPC protocol over a WebSocket transport. This allows for persistent sessions to be established with two-way asynchronous communication between game-client and server.

Instances communicate establishing a connection with an associated session and issuing Remote Procedure Call (RPC) [add reference] requests. Each request has an associated method and payload, which are forwarded to the handler which processes the request. Both client and server expose a set of supported RPC methods with respective request handlers. Handler can respond with an optional response to the request.

It is important to note that while there are distinct server and client roles, they are only used in the context of establishing the connection: client dials the server and originates a new session. After the session is established, both client and server can issue RPC requests to the other party.

During the lifecycle of the connection the client goes through a number of states as depicted by the diagram on the Figure 9.

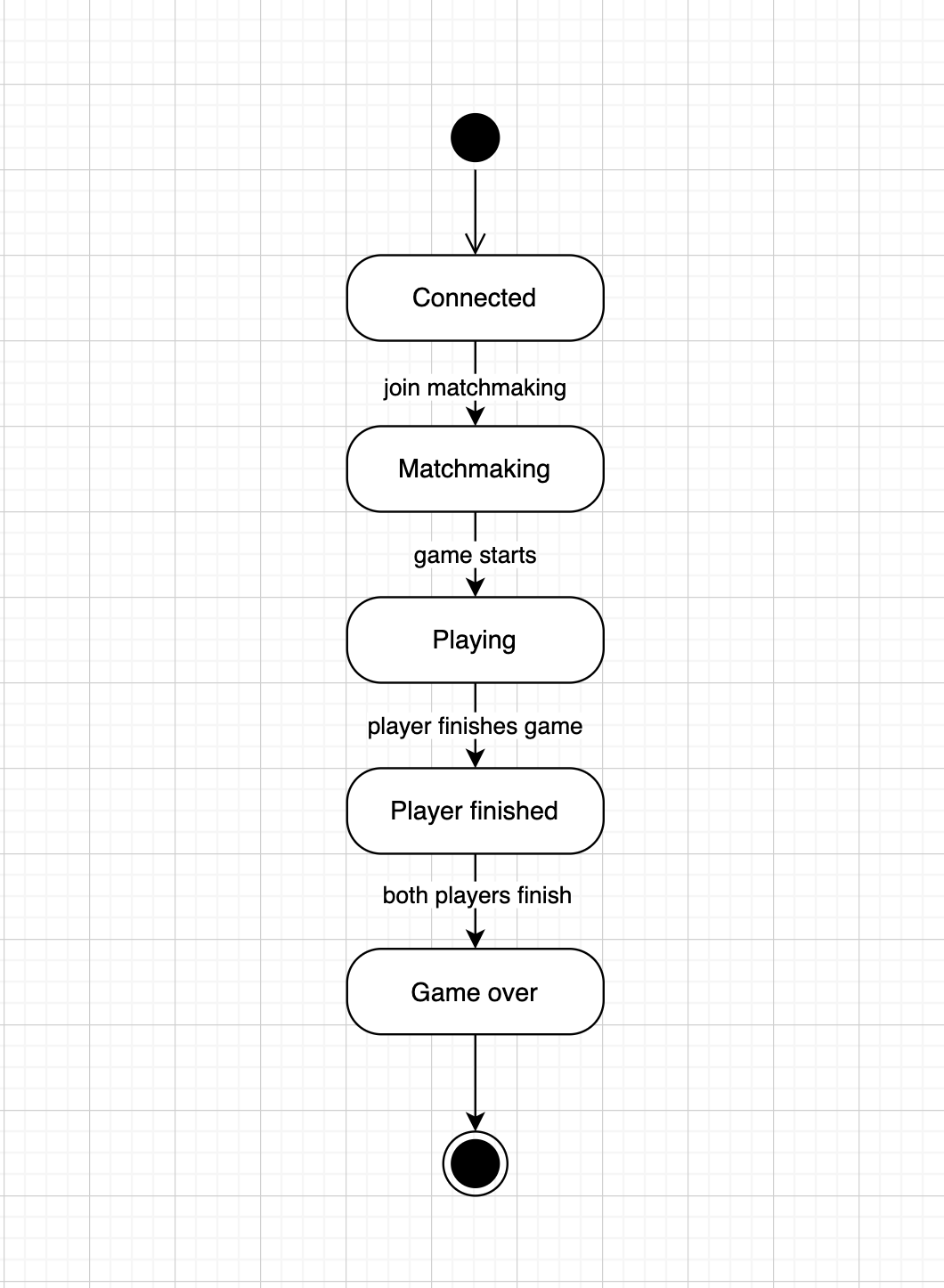


Figure 9. Client state diagram.

### Web-sockets

Rationale for using web sockets.

API reference: JSON RPC

### JSON-RPC

JSON-RPC is a popular, lightweight protocol for implementing remote procedure calls. JSON-RPC spec is available at <https://www.jsonrpc.org/specification>. Version 2.0 is used here.

Each message in JSON-RPC is a JSON [https://www.json.org/json-en.html] encoded object. Request and response format is defined by the protocol. Each request must have the following fields:

* “jsonrpc” – specifies the protocol version. Must always be “2.0”
* “id” – string or numeric request id. Must be unique among other requests. Responses to this request will have the same id, so they can be routed properly.
* “method” – string containing the method name to be called. Method names are application specific.
* “params” – optional field containing parameters for the method invocation encoded in JSON.

The server may respond with either a success response or an error. In any case, the response must contain the same “jsonrpc” and “id” fields. Additionally, for successful responses a “result” field will be present containing the result of the message invocation encoded in JSON. For error responses there will be an “error” field containing the error message. Here’s an example of a method invocation in JSON-RPC:

--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}  
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

The protocol is implemented in „JsonRpc” class. Since both client and server can originate and handle requests the implementation is the same for them. Each one has an instance of the class. Upon creation, a callback is passed to the constructor which allows the class to send a string via the transport socket. Likewise, the incoming messages must be forwarded to the “JsonRpc” class using “HandleMessageFromTransport” method. The „JsonRpc” class itself is transport-agnostic, requiring only that the underlying transport be reliable and delimit messages somehow. Meaning that messages sent using the callback must arrive at the other side without any partitioning or changes.

Since the “JsonRpc” class doesn’t define any specific RPC methods that are available and only takes care of routing and serialization, the code making use of this class must define it’s own handlers for RPC methods. “Handle<TArg, TRes>(string method, Func<TArg, TRes> handler)” is the method used to assign a handler for a certain RPC method. When a request with the matching method identifier comes in, the provided handler will be used to process the request. The request parameters would be deserialized. The return value of that method will be serialized to JSON and sent as a response. Parameter and response types are specified using C#’s generics feature. The response message would be sent with the same ID as the request. “JsonRpc” also doesn’t check request IDs for collisions, the client is responsible for that.

The code that originates requests is a bit more complex. The algorithm is as follows:

1. Generate a unique id for the request. Each RPC client has an internal counter that starts from 0 and generates sequential ids.
2. Serialize call arguments to JSON. Newtonsoft.Json library is used for that.
3. Construct the request message. This includes request id, method name, and serialized arguments.
4. Create and register the TaskCompletionSource. It will allow us to wait for the response to arrive. This will be described in more detail in the next paragraph.
5. Send the payload constructed in step 3.
6. Wait for the task created in step 4 to be finished, it will wait for the server’s response and resolve with the received message.
7. Remove the TaskCompletionSource registered in step 4 to prevent memory leak.
8. If the response was an error, construct the exception object and throw it.
9. If response is success, decode the result using the same Newtownsof.Json library and return it.

Text

Description automatically generated

Figure 10. Method to originate an RPC call.

The method will block while waiting for the server to respond. To prevent blocking an OS thread which might stop game logic from executing, we leverage C#’s Task asynchronous programming logic (TAP) [https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model] alongside with async-await syntax. This allows us to make use of so-called “green threads” which logically look like separate execution threads, but don’t result in separate OS threads being created. They are also a lot more lightweight then OS threads which allows us to use many of them without performance concerns. They implement cooperative multitasking model rather then preemtive one. This means that the current green thread will yield the control back to the scheduler to execute other tasks. In this case it happens when “await” keyword is evaluated, as show on Figure 10. Scheduler will then wake up the task to continue its execution when the RPC response arrives.

To integrate request origination and handling functionality a message router must be present. It determines whether a message is an incoming request or a response to a one that was sent. If message has a “method” field, it is an incoming request. In this case a handler for that method is retrieved from the set of the ones that were registered. The handler is executed. And a response message is sent with the same ID.

If the incoming message doesn’t have a “method” field it is considered a response for a request. In this case a TaskCompletionSource for the specified request ID is retrieved, and the green thread waiting for the method response is woken up and forwarded the response message. Figure 11 contains the code for the router.

Text

Description automatically generated with medium confidence

Figure 11. Router for incoming messages.

There are number of improvements that could be made to the current implementation. First and foremost the client does not handle calls that don’t return values. This would correspond to “void” return type in C#. Same goes for methods without parameters. There’s a workaround for this problem: “object” type can be specified and “null” value can be returned or passed as an argument in such cases. Adding support for this would require a special case in the code. But since the workaround is pretty straightforward the issue is not high priority.

Second improvement can be made to a way how APIs are defined. Currently the server calls “rpc.Handle” with a string identifying the method name and passes the handler function. It is then the client’s responsibility to correctly call that method by name as well as match the parameter and return types. Incorrectly specified types would not be checked during compilation and would result in a runtime error. This means that there’s no way to formally specify the API in code that would be statically checked by the compiler.

The problem can be solved by users having to declare C# interface that describes the available API methods. Then, using reflection, the server could enumerate the interface’s methods, and extract their parameter and return types. Likewise, on a client the RPC implementation could dynamically create an interface instance using the code generation instruments provided by the .NET standard library. The aforementioned interface could be shared between the client and the server and act as a contract between them to keep the API consistent.

### Backend server

Backend server is written in C# using the .Net Core runtime. The choice of C# as backend language offers a significant advantage by enabling code-sharing with the game code. All of the data that is shared over the network is defined in a single module which helps to keep the client-server API consistent. Using .Net core simplifies hosting as it can run natively on Linux-based systems, which are by-far the most common in hosting platforms.

Upon start the backend server starts creates a game-manager instance which tracks all current games as well as the matchmaking queue. The server also starts listening for incoming WebSocket connections on a given port. The port can be configured through an environment variable.

The WebSocket [add reference] protocol implementation is provided by the “NetCoreServer” library [add reference]. Mainly, it exposes two classes: WsServer and WsSession. The former is used to spawn a WebSocket server that listens from incoming connections on a given port and then creates a session object. The WsSession is used to represent that session. The connection state as well as packet decoding is handled by the library. The two classes can be inherited from to add domain logic that handles incoming messages and can send outgoing ones.

The following methods are available on the WsSession class that are important for the implementation:

* OnWsReceived(byte[] buffer, long offset, long size) – to be overridden by the subclass. Allows to capture incoming data packages.
* OnWsConnected(HttpRequest request) – called when the connection is esablished.
* OnWsDisconnected() – called when the connection is dropped.
* OnError(SocketError error) – called when there’s an error in the connection.
* SendText(stiring msg) – Can be called to send a data packet.

The core of the networking implementation is located in the SocketServer and SocketSession classes. The SocketServer extends the WsServer class which listens on a given port and then spawns a GameSession instance. GameSession instances represent a single incoming connection and therefore a single game client that is connected. They are identified by a GUID [add reference] stored in the Id prop.

GameSession instances also set up a JSON-RPC peer that can handle incoming requests from the client and also originate server-to-client calls. A pair of handlers is set-up to decode and forward binary messages coming from the web-socket to the JSON-RPC peer, and another one to write the response messages into the web-socket channel. A series of handlers is set-up to handle the calls coming from the client. They are described more in detail in section 6.5.

The state of the games is handled in ServerState and GameState classes. The latter represents a state of a single ongoing game as well as the information about players involved. The former hosts the collection of games as well as the waiting queue for players. The separation of those classes from SocketServer and SocketSession was deliberate to separate core domain logic from networking aspects. This makes it so that the domain logic can be tested separately without potentially complex integration with web-socket library in tests which is undesirable. It also makes so that SocketServer and SocketSession are mainly responsible for integration domain classes, web-socket networking and JSON-RPC together, which limits their complexity.

### Client-Server API

Both server and client register a number of JSON-RPC methods with corresponding handlers. On the server side they are registered in the “PlayerSession” class. The following methods are provided by the server for the client to call:

* “JoinMatchmaking” – Request by the client to put in the matchmaking queue. Has a single parameter of a type string – that player’s display name. It will be displayed to the opponent when the game starts. Example call is: “JoinMatchmaking(“Bob”)”.
* “MakeMove” – Sent by the client to make a move during a live game. Client issues this command in response to the player clicking on a board. Has a single parameter of type “Move”. This and other data transfer objects (DTOs) will be described lower in detail.
* “FinishGame” – Sent by the client when the player clicks the “Finish game” button. This means that the player can no longer issue moves. This method has no parameters.

Additionally, the client provides a number of methods callable from the server. They are defined in “ServerConnection” class. The following methods are provided:

* “UpdateClientState” – Issued to notify the client that it’s state has changed. Has a single parameter of type “ClientState” that will be described lower.
* “UpdateBoardState” – Called when there’s an active game the client is participating in to notify that the state of the board has changed. Has a single parameter of type “BoardState” that was described in section 4.1.

The “Move” class defines a data structure to represent a single move. It has the following fields:

* “Player” – of type Player (enum). Either Red or Blue player that made the move.
* “Row” – the row index of the placed unit.
* “Col” – the column index of the placed unit.

The “ClientState” class defines a data structure that represents a state of a particular client as seen by the server. It has the following fields:

* “State” – enumeration of type “StateEnum”. Represents the current state of the client as well as defines what fields will be present in “ClientState” object.
* “Player1Name” – string, name of the first player. Only present when State is “Playing”.
* “Player2Name” – string, name of the second player. Only present when State is “Playing”.
* “Winner” – type “Player” (enum). Describes the winner of the finished game. Only present when State is “GameOver”.

The “StateEnum” can have following values:

* “None” – initial state when client has just connected to the server.
* “Matchmaking” – set when the client enters the mathmaking queue.
* “Playing” – set when there’s an active game.
* “GameOver” – set after the game has finished.

### Connection flow

The message flow from connection to the game start is shown on the Figure 12. In the scenario below the red player connects first, joins matchmaking and then waits for the blue player. When blue player joins, the game commences.

Upon connecting to the websocket endpoint, the red player receives “ClientStateUpdate” message with the state “None”, which is the initial state for the newly connected as shown on the Figure 9. After connection, the client requests to join the matchmaking queue and also provides the display name of the player, which was previously entered by the user. The server updates the state and marks the player as waiting for the match. Corresponding “UpdateClientState” message is also sent with “Matchmaking” state. At this point a message is shown to the player that a server is looking for an opponent.

Once the blue player goes through the same flow and joins matchmaking, the server can pair them together. A game entity is created on the server, linking both of the players. Players are also taken out of the matchmaking queue. Both players will receive “UpdateClientState” message with “Playing” state. This signals the game client to display the game board.

Server will also bootstrap an empty board state with initial configuration. Following that, “UpdateBoardState” message will be sent containing the full board state serialized in the JSON format. The clients will deserialize the message and update their local state correspondingly. Because the state itself is not particularly large, we can serialize it and send it over in a single message. This simplifies the code as well as there no need to deal with partial state updates: client just receives a new state from the server and replaces the previous one with it.

At this point both of the players are ready to play the game, the process of which is described, in-detail in the next section.

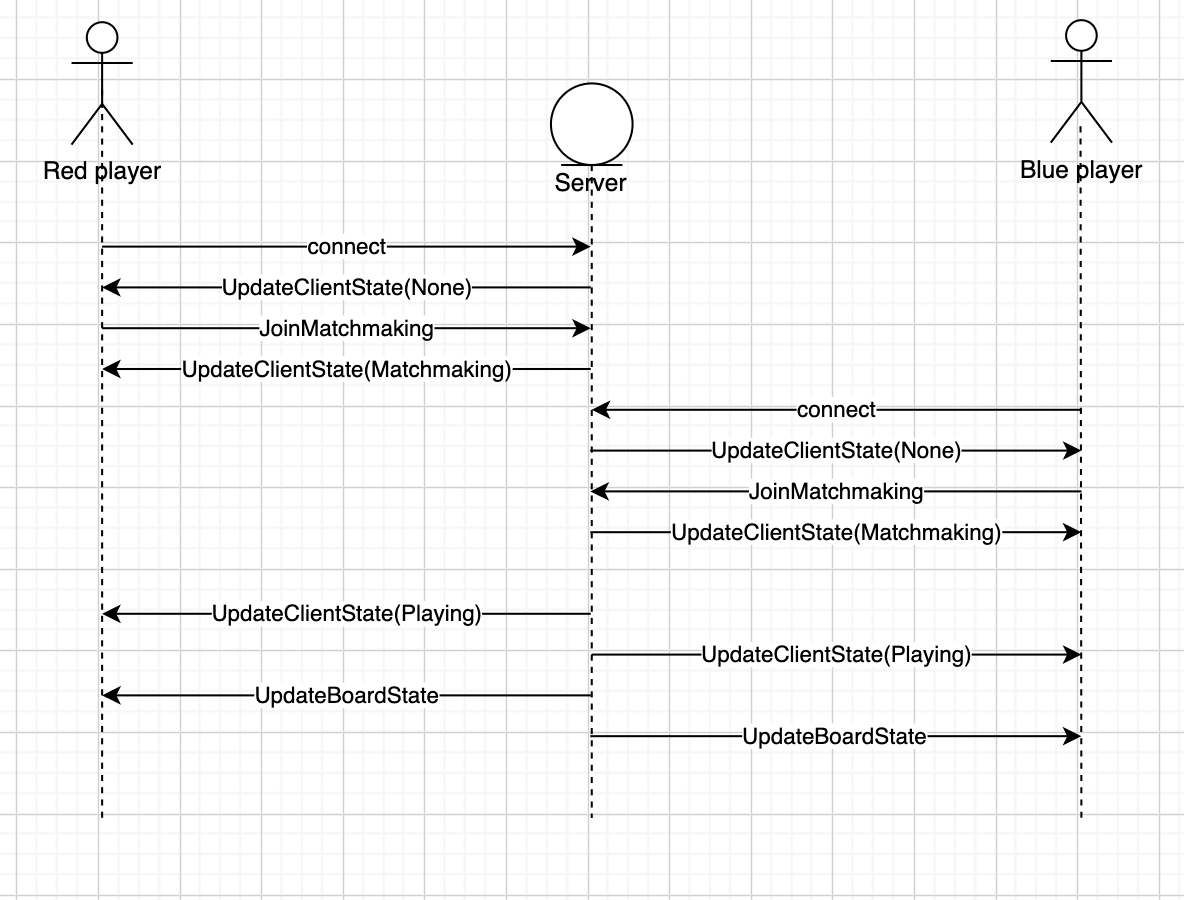


Figure 12. Connection and matchmaking sequence diagram.

### Game flow

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*.