Program Architecture

Max Burgert, Tom Cinbis, Paul Höft and Tim Königl

Contents

1 Overview

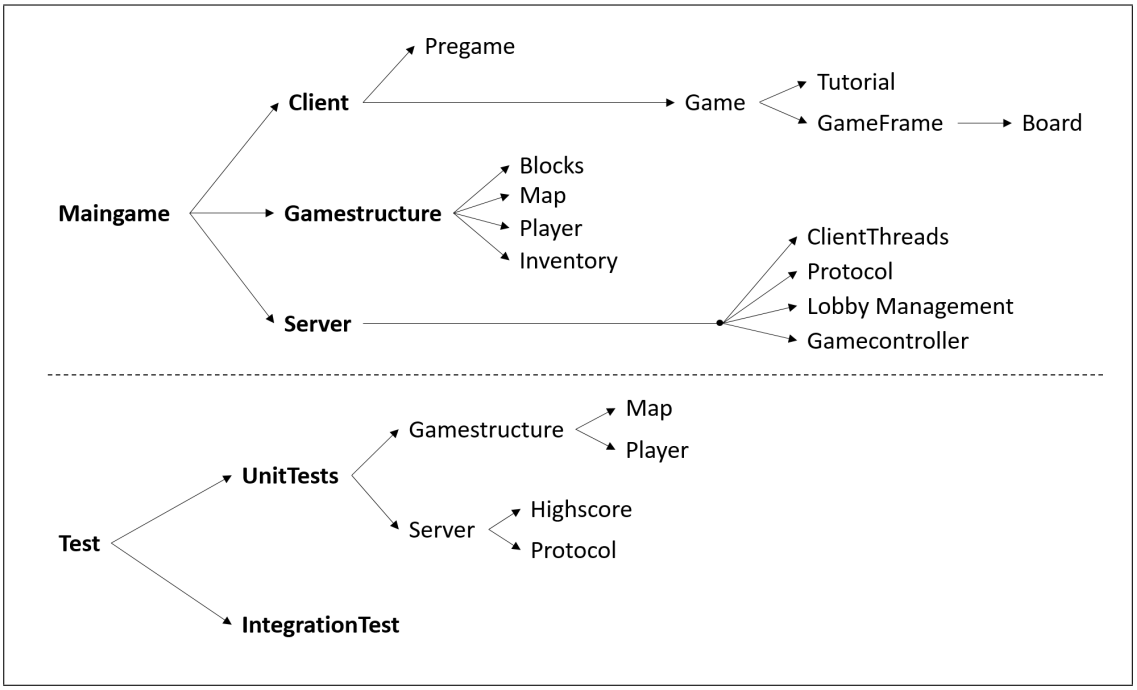

Fig. 1: A brief overview of the general architecture

Our project is divided into two main parts. Part one called 'Maingame' and part two called 'Tests'. Obviously 'Maingame' contains everything important to be able to play a round. We need a server, multiple clients and game objects to interact or play with.
Our tests are responsible for checking the most sensitve parts of our game and assure that after each new implementation or change in general the game still works as expected.

2 Maingame

2.1 Login

Once you started a server and a client we will promt the user to enter a nickname which is used during the whole time the user is playing or chatting. Like Phoenix from the ashes our game starts to establish a connection to the server and will initialize, once we got an answer from the server. To be able to independently send and receive commands and information we start to seperate threads, which process everything. While our client is busy creating his threads and preparing everything, our server creates a thread for each client to receive their messages and send him commands.

2.2 Communication

Because our game is a realtime game, the server-client communication is one of the most important parts of our project, so in this section we want to give more information about our protocol, how we designed it and of course, how it works.
Furthermore there is a figure and an explanation of the general path all informations go through when they are exchanged between server and client.

2.2.1 Protocol

One requirement was that the protocol should be human readable. To achieve this we used different enums for every command a server can send to its clients and the other way around.
First of all, the fact is that all commands are built the same way:

<command>;<nickname>;<arg1>;<arg2>;...;<argn>

That all arguments are split by semicolon could at first lead to some mistakes and loses of smileys, but meanwhile we replace it while it is sent by some rare combinations that nearly nobody uses in the normal chat.
The list of all command can be looked up in our protocol documentation in our git repository if some interest exists.
Another focus of us was to make the protocol modifiable which was another reason we switched from normal strings to enums. By using enums we have the server and the client enums in two classes and at first sight it is possible to see what commands are in use, but also it is clearly better to add new commands.
To do this, only an enum - combined with a additional switch case - has to be implemented in our protocol process method plus the corresponding sending and receiving method on server and client side depending on which way the information goes.

2.2.2 Flow of communication

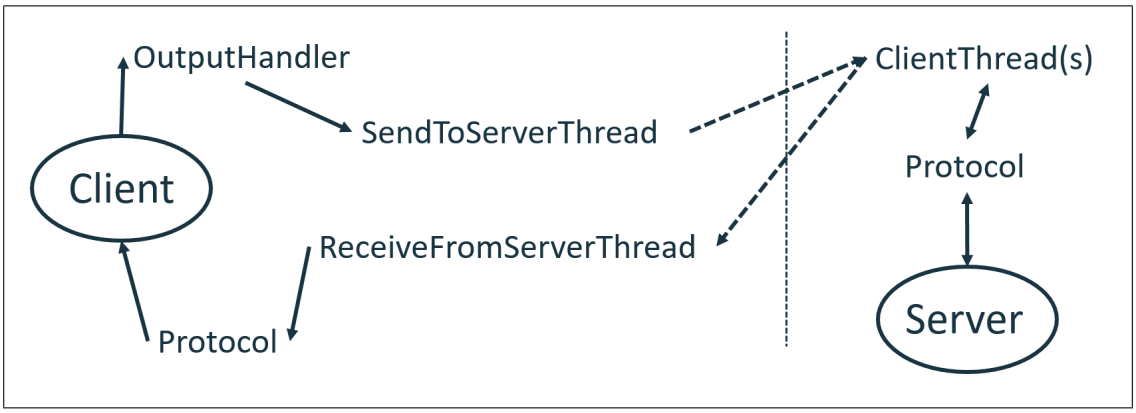

Fig. 2: Server and Client communication

In Fig. 2 a schematic artwork can be seen that shows the ways information can go through in our

project. The core of this communication are the sending and the receiving threads which interact overlapping and the proctol which calls the right functions.
If we start our way on the clientside the Outputhandler gets all the output the client wants to send to the server. This Outputhandler uses the sending thread to deliver the command to the serverside where the ClientThreads - the server has one ClientThread for each client - receive this message.
Interesting to see here is that the ClientThreads have a double function. On the one hand they receive all incoming commands and on the other hand they send off all outgoing commands from the server.
Going back into the direction of the client, the receiving thread from the client forwards all information to the protocol which calls the final methods of our client in the end.

2.3 Gamepreparation

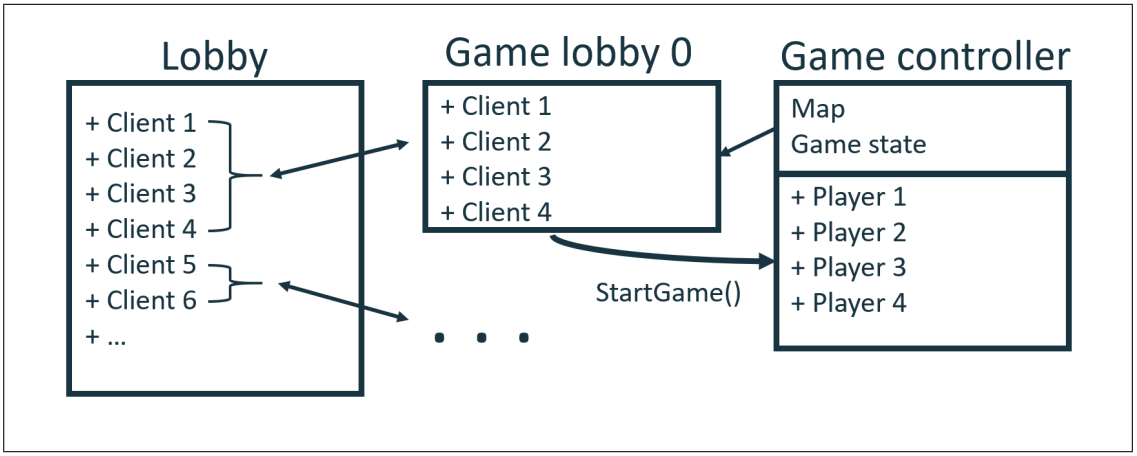

Fig. 3: Lobbymanagement

Fig. 3 shows the organization of our lobbysystem which can be explained with the aid of sets.
After connecting to a server and logging in, every user ends up in the standard lobby the first set. He can now decide wheter he wants to create a new gamelobby or join an existing one.
By entering a gamelobby a big change happens because these two lobbies are not one set. We implemented our lobbysystem in a way that the users can only see all chat messages from other users which are part of the same lobby. So whenever a chat is sent, the server validates each clients location and only sends the messages to the right ones.
These restrictions can be bridged by using two chatfunctions: Whisper & Broadcast. Whispering and broadcasting won't be affected by any gamelobbies, so a client can whisper some private messages to other clients or he can broadcast important stuff to all clients connected to the server.
Back to the gamelobbies which are the essence of starting a game. If a gamelobby is full and all clients pressed the ready button, the server recognizes this and starts a game. What happens is that the gamelobby gets kind of transformed into a GameController which has the right properties

to start a game.
The GameController now has a map (more in section 2.5.1.1) and a gamestate (open, running, finished) but most important there are now not anymore normal clients but playerobjects for each clients (section 2.5.1.3) who in the end make it possible to play a game. Additionally the GameController is the class where the serverside gamecycle takes places, but more later.

2.4 Game

After the introduction of the overall server-client communication and the lobbyhandling the next step is now to explain how our game itself works.

2.4.1 Initializing of a new game

In section 2.3 we read that by starting a game a gamelobby gets converted into a gamecontroller and by looking at Fig. 1 we can see that there are two more clientside instances, the GameFrame and the Board.
The GameFrame itself is the container for our graphical environment and the Board is half of the heart of our game. The Board has three important tasks:

1. Listening to keyinputs by the user
2. Sending all changes of the player and the map to the gamecontroller
3. Updating the graphics depending on the incoming server updates

To listen to all keyinputs the Board implements a keylistener which is directly connected to the clients player and updates its position and actions locally.
Every client Board sends its own changes to the server which handles all clients and in the end the Board - which additionally is a runnable - listens not only to the keyinputs, but also to the server who gives him the information about how he has to update the other players and the map. In the end the Board calls the draw functions and draws all players, the map and a minimap into the GameFrame.

2.4.2 Gamecycle

In the last section we already talked about the Board of the client which is half of our gamecycle and its tasks. But there is another part - the GameController.
The GameController gets initialized serverside after starting the gamelobby. He holds all player information of **all** clients, he holds the permanently updated mapstatus and for each client there is a ArrayList where all client updates are stored. While initializing the GameController has some different exercises to do. He has to tell the players that the game will start soon, he initializes all ArrayLists, changes the gamestate of the lobby to running and starts a GameThread and a GameFrame with a Board. This is where the gamecycle starts.
This gamecycle is now a permanent circular flow between the Board and the GameController. The tasks from now on are in general similar to the ones from the Board, so he has to update the

players and the maps and the has to send back all information to all clients. And this is where it gets tricky.
The GameController does not only get updates from one other part, he gets inputs from all playing clients and has to process them properly.
Additionally to these similar but still more complicated tasks like the ones from the Board he also always has to check if the game is still running, if the gametime is expired or if one team has reached the winning point.
One point where we achieved a big simplification was the change from sending the whole map to sending single blocks. There is not only a ArrayList for the player updates, but also a ArrayList for the blocks. So if a player interacts with a block the server gets a seperate update command and only updates single blocks in its map. The same way it is working on the client side.
In the end if a team wins the game or the time is expired the GameController has to inform all clients that they have to close the game. When every client is back in the lobby the GameController changes its gamestate to finished and displays the highscores and the scores from the finished game.
By opening up the gamelobby a new game can be started and the loop will start again.
- Create a GameController (Server) - Generate a map - auto generation? - Add all players from one lobby - call gamecontroller.start() - init Buffers for sending and receiving - Tell the players, that a game will start soon - change the state of a game lobby to 'running' - start a new gameThread - our gamecycle starts from now on (see 2.4.2)

2.5 Gamestructure

2.5.1 Game objects

2.5.1.1 Map

The whole idea is based on a world, the user can interact with, build things, destroy things and overall do what he wants. So we decided to use a grid based layout which stores, manages and displays different types of blocks (see 2.5.1.3). Our map is responsible to check whether a block can be removed by the player or not.
Additionally metapoints like spawn points and winning points are set and stored in the map class. A big feature is that our map is generated in a random way so it is different each time a game is started.

2.5.1.2 Map Generator

The generator for the map is divided into six parts, as shown in Fig. 4 to Fig.9. Each time a new game lobby is created or the status of the game lobby is changed to 'Open', a new map will be generated and added to the game controller. The game controller gets a map which processes a 2D double array from the map generator into a 2D Block array.
In the end, the map will be surrounded with bedrock blocks, as well as the 'winning platform' as they are not a part of the map generator.

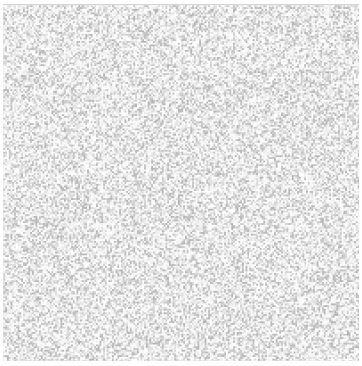

Fig. 4: Random noise with bias to moderate peaks

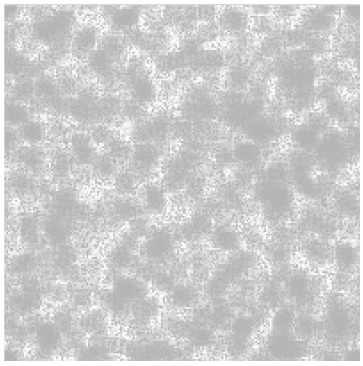

Fig. 5: Find peaks and lift surrounding pixel to it

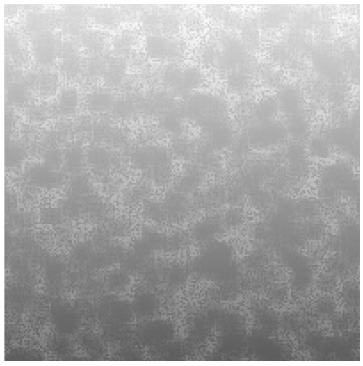

Fig. 6: Apply gradient from top to bottom

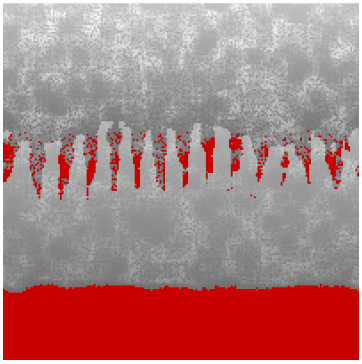

Fig. 7: Define the solid parts of the map

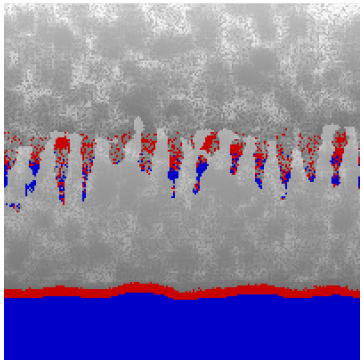

Fig. 8: Apply stone (blue) to solid layer

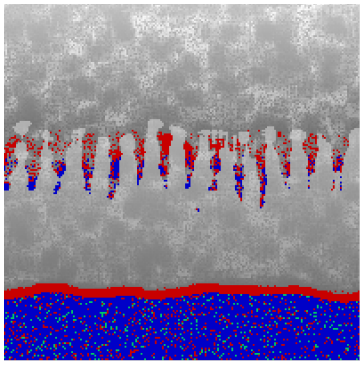

Fig. 9: Add Teamblocks to stone layer

2.5.1.3 Blocks

As already mentioned in previous sections, we currently have five major block types, but before talking about each type of block in detail, lets understand the basics of a block.
Each block has to define **eight** major parameters: Is it **removable**? Is it **visible**? Which **texture** does it use? How long is its **removing time**? Can a player **collide** with it? Which **teams** can collide with it? What is his x and y **position**? Is it a **winning point**?
All these properties define our different block types which are going to be listed in the following:

1. Airblocks
2. Dirtblocks
3. Stoneblocks
4. Teamblocks

In general the first three are pretty much identical and just differ in texture and removing time. For instance an airblock is not removable, because a player can not interact with it, nor does he realy

see it. Just Dirt- and Stoneblocks are blocks every user can remove, place or jump on.
Kind of special are our teamblocks, because only one team can use its teamblocks (colored in their teamcolor). Players of other teams cannot remove it and also won't collide with these blocks. Thats achieved by asking a player for its team id and based on that our collission detection will decide whether a collision will happen or not.

2.5.1.4 Player

The player object is the container for all properties that are needed for our gamelogic (section 2.6).
A player is the object with far the most different properties, he does not only contain obvious ones like its team, name, position, speed and the lookdirection. He has also lots of attributes like a connection to a special clientThread, a personal score, the map, a camera and a inventory to mention only a few important ones.
But the the biggest part of the player is the gamelogic where all the interaction between him and the map will be calculated and validated.

2.5.1.5 Inventory

In the last section we talked about the different attributes of the player. One of them is the inventory.
The inventory consist of different stacks for each block type with constant defined sizes. By using the number keys a user can switch between the different block types and if a player removes a block it automatically adds a block to the corresponding stack and marks it as the active one.
The inventory is validated internal, but it is also displayed in our GameFrame, so a user always sees how many of each block he has for choice.

2.6 Gamelogic

We have talked a lot about players, the map and that they interact but in this section we want do describe how we implemented our gamelogic, mainly our collision detection.
First of all it is important to say that our collisions are detected by intersections of rectangles due to the fact the players and the blocks are built of rectangles.
We use a three-staged reverse linear interpolation to get all collisions. So if the player gets an update that he should switch its position he wont spawn immediately there. Instead at first the diagonal movement x+y will be regarded and if there is a collision at the updated position he will be moved back until there is now more collision. But that is only the first stage. The same process is done next in the x direction and in the last stage in the y direction to guarantee that the player is not displayed in a block.
Depending on the framerate we set in our Board this has to happen about 20 to 30 times per second.

3 Tests

As part of our quality assurance we have made two types of tests. First we used UnitTests to test critical edge cases in important classes of our project and furthermore we implemented a integration test to test the whole project by using a testrun.

3.1 Unittests

We divided our Unittest into gamestructure and server tests to guarantee a working game.

3.1.1 Gamestructure

In our gamestructure the map and the player are tested to be sure that the interaction can between them can be used correctly and our program handles wrong inputs right.

3.1.1.1 Map test

In our map - which is initialized by a string - we mainly focus on a right parsing of the map from a string to map or the other way around. There it was especially necessary to not only test correct strings, but also completely wrong ones in different variations.
After testing the whole map itself, also the single components of the map - the blocks - will be validated. So there are tests of initializations of different blocks and of block requests.

3.1.1.2 Playert test

In our player test we focused on the update and the removing functions which means we first tested if a player gets correctly updated depending on the updatetime . Additionally a test method lets a player remove some blocks which he can remove and ones he cannot remove.
That made sure that our player interacts in the right way with our map.

3.1.2 Server

In our server tests we decided to test two different parts of our project. There was no doubt about the testing of the protocol because this is the part of the project on which everything depends. Additionally we tested the highscore class.

3.1.2.1 Protocol test

The protocol which is possibly the most important test was accomplished with a big variety of inputs. We initialized a protocol by using a mockserver and a mockclientthread and processed first correct inputs and afterwards wrong ones. Especially when testing the wrong ones we tried to think of completely different and wrong inputs the protocol could get to improve our errorhandling.

3.1.2.2 Highscore test

This test will check whether all of our parsers work correctly and if our logic inside the highscore class works correctly. So lets talk about the logic and the doing inside.
First of all we read all data that already exists in our saved 'highscore.txt' file and temporarily store them in ArrayLists. These datasets are already sorted by their scores (descending), because we order all of them before saving and writing them into our file. Now we can find the correct position inside of the ArrayList, get all Teammembers as a String array and their score as an Integer and insert it into the list.
In the end everything gets written into the txt file. The entries are always inserted in a special way:

nickname1,nickname2,...,nicknameN:totalPoints

The seperators are handled in the same way as in the protocol.

3.2 Integration

Here we test our whole ecosystem, by using it all togehter. A mockserver will start, a client wants to connect automatically, writes some messages in the chat, creates a gamelobby, joins it, starts a game and then walks around a little bit. While doing all these things sequentially our test will check some basic points and then decied whether it was a successfull run or not.