Radboud University

Research & Development: Real Life App

Authors:
Thomas Berghuis
Steven Bronsveld
Jelmer Firet
Robert Koprinkov
Thijs van Loenhout
Bram Pulles

Teacher:
P. van Bommel
Student assistant:
Jan van Bommel *group:*
PT-02

Contents

1 Introduction

This is the project report for *Arcemii*. Shortly put: Arcemii is a multiplayer rogue-like app for android.

Arcemii came to be, mainly because of one reason: there wasn't anything like it yet. The idea was planted over a year ago. During the summer break before this year, Steven and Thijs wanted to play something with their friends, something with a lot of replayability. We looked for multiplayer rogue-likes, but could not find any. Summer ended, the university started, but this idea remained. Now with this course, *Research and Development*, it was finally time to realize it!

Above, the term 'Rogue-like' is mentioned often, and it will be mentioned even more frequently later on. But what is a 'Rogue-like'? According to Wikipedia: *Roguelike is a subgenre of role-playing video game characterized by a dungeon crawl through procedurally generated levels, turn-based gameplay, tile-based graphics, and permadeath of the player character, mostly with a fantasy narrative.*

So, Arcemii is a game where you go through randomly generated levels, killing fiends along the way. However, few rogue-likes have a proper multiplayer mode. This serves as the spot Arcemii has to fill. Alongside that, we want the game to be able to be played in short bursts, like in the break of a lecture. We hope to create something people can jump into easily and can bring them quick enjoyment.

The demographic we aim at, therefore, consists of people who would like to play a game on their phone with friends. Maybe they normally game on a dedicated system or their pc, but we want to bring that experience, alongside the chaotic fun of multiplayer games, to their phones.

2 Description

In this section, we will give a description of our overall system as well as a justification for our system and diagrams to support the description of the overall system.

2.1 Focus on properties

Our system contains a lot of properties, to make all of these properties clear we use an enumeration.

1. The user can see the connection status to the server on the top right of the home screen. The user can also create a party from here and go to the settings and join a party screen.
2. If the user's phone is disconnected to the server, then the player cannot use the create and join party buttons.
3. Our application has a few settings: the user can change and save their name. This name will be shown in the lobby. The user can toggle btween offline/online mode using a button and the user can tobble between muted and unmuted music (heard in-game) using a button.
4. The user can join a party by filling in the game pin of that party and clicking on enter.
5. The user can select abilities in the lobby. The user can select three abilities which will be displayed in the game in the order they are chosen. The user can also see the master of the party, all the other party members, the ready status of the other party members and the party's game pin.
6. If the user is the master of their party, they can start the game by pressing the button, provided all other players are ready. If the user is not the master of their party they can get (un)ready by pressing the button.
7. The user can see the health of their character in the top right corner when in-game. This value changes in real-time when the character takes damage or is healed.
8. When playing with multiple players, every player has a differently colored character.
9. While in-game, the user can walk around using the joystick and activate abilities, when they are not on a cooldown (they will appear opaque when on cooldown), using the corresponding buttons on the bottom right.
10. The abilities have various effects, but in our current implementation the user can teleport 4 tiles in the direction the character is facing, the user can damage other entities using arrows and a melee attack and the user can heal itself with ten health.
11. When a player's health is displayed in the upper right corner when in-game. When it drops to zero, the player respawns at the level's spawn-point.

12. The enemies fight back using their well developed A.I.
13. When the boss is defeated, the player(s) return(s) to the lobby.

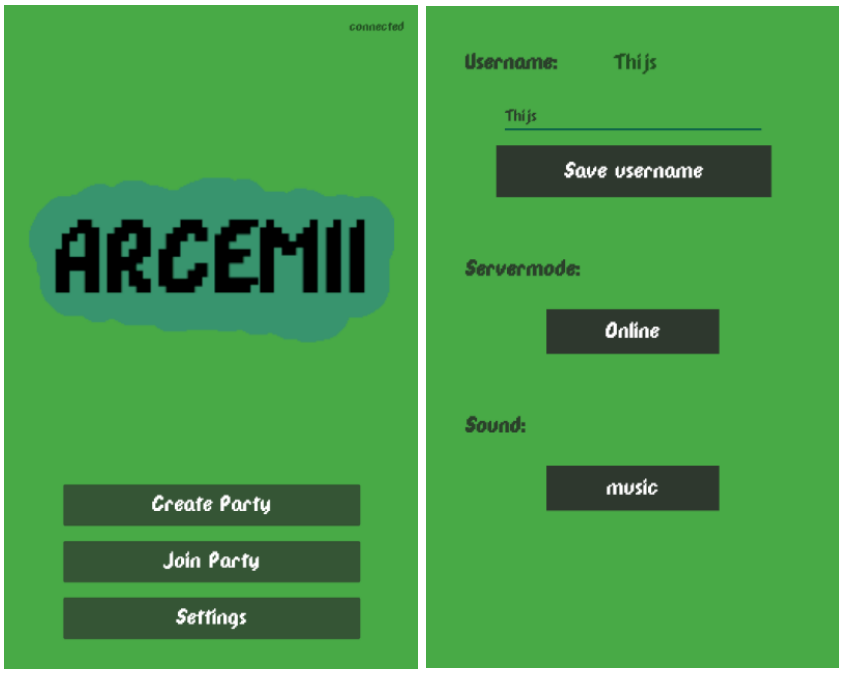

Figure 1: The main-menu and the settings menu of the app

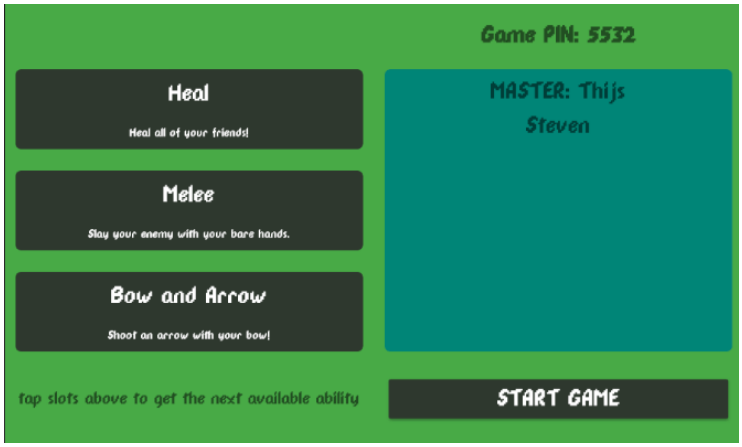

Figure 2: The lobby of the app, on the left you can see the chosen abilities, and on the left you can see the members of the party

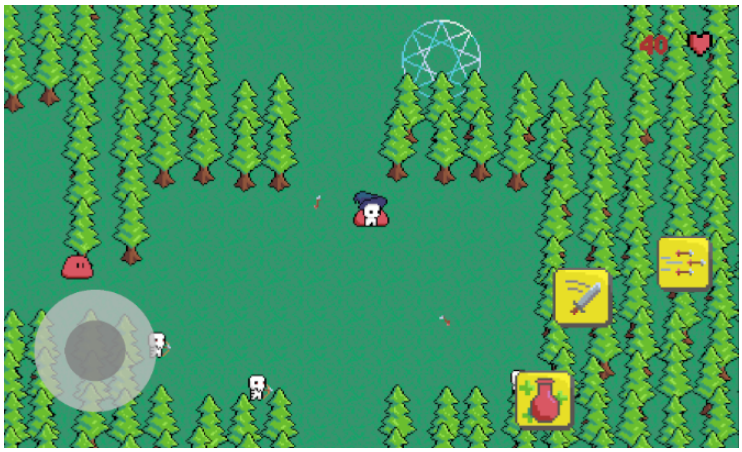

Figure 3: The actual game. You are the player in the center of the screen. On the bottom right you can see two skelletons shooting arrows. On the left you can see a slime.

2.2 Product justification

2.3 Specifications

In section 2.1, we presented a list of properties of our application. As we'll go pretty in-depth in the next section, we decided not do that too here, as we'd only be explaining ourselves twice.

We do want to discuss some of the structures used in our code. For Reference, see sppendix A. We decided not to expand every square present there, showing their methods and such, because the figure would become much too large and not easily readable that way.

3 Design

In this section, we give a global and detailed description of the design of Arcemii. Furthermore, we give a justification for our design choices.

3.1 Global design

3.2 Detailed design

In this section, we give a detailed description of the design in terms of classes, methods, and attributes.

3.2.1 Server-client relation

Arcemii makes use of clients and a server to enable the possibility of playing the game in multiplayer mode. In this section, we will describe the most important details of this server-client relation.

Arcemii can be played in two different modes: offline and online mode (for singleplayer and multiplayer respectively). In both cases, we make use of a server. When playing in offline mode, the server is running in the background of the mobile phone. When playing in online mode the server is running on a dedicated server, which can be connected to through the internet. Apart from this, the only real difference between singleplayer and multiplayer mode is to which IP-address the mobile phone will try to connect. In singleplayer mode, this is the so-called `loop back address`, also known as `localhost` or `127.0.0.1`, whereas in multiplayer mode, it tries to connect to the IP of the dedicated server.

Server Let's start with a detailed description of the server. The server-side has four classes: `ArcemiiServer`, `Server`, `ServerGameHandler` and `Console`.

Since the server needs to be able to run on its own, we have a `main` method in the `ArcemiiServer` class. This method creates a new object of all the other classes to start the server. `ArcemiiServer` also contains a `stop` method which stops the `Server` and `ServerGameHandler` classes from running. When the server is run on a dedicated machine, we can just run the program separately from the rest of the application. When the server is run on the background of the mobile phone we just call the `main` method which simulates the same behavior, but then locally.

The first class we instantiate when we run the server is the `Console` class. This class creates a terminal interface for interaction with the server. This is very useful when the server is run on a dedicated machine to enable some control over the program. The console has three commands: `help`, `stop` and `log`. The `help` command gives a list of all the available commands. The `stop` command terminates the program. And the `log` command toggles the logging on and off. Logging is very useful for debugging the server. All the classes in the server call the method `log` in the console to send debug information.

The second class we instantiate is the `ServerGameHandler` class. This class, as the name already says, handles all the game logic on the server-side. When this class starts, it creates a new thread which sends an update message to every party on the server every tick (which is set at 20 per second, in the version that was handed in). Apart from this, there are two very important methods in this class. Namely: `addPlayer` and `handlePlayerInput`. When the first method is called, a new thread is created which listens for messages coming from that specific client. Every time a message arrives, the second method is called to handle the message from the client. This method calls the appropriate method for all the possible messages. The messages are all a subclass of the abstract class `Message`. Some examples of messages are:

- `CreatePartMessage`, to indicate to the server that the client wants to create a party.
- `JoinPartyMessage`, to indicate to the server that the client wants to join a specific party.
- `ActionMessage`, to indicate which actions the player wants to execute while playing the game.

The last class we instantiate is the `Server` class. This class continuously listens for new clients, creates a connection with these clients and adds the client to the `ServerGameHandler`. To prevent overloading of the server, we also check if there is already a client on the server with the same IP-address. If this is the case the old client will be removed since this one is now replaced by the new client. This reduces the load on the server because the server does not have to listen for messages coming from the old client this way. Note that we have an exception for the `loop back address` to make it possible to connect with multiple emulators to the server for testing purposes.

Client Here we will give a detailed description of the client. The client-side has two classes: `Connection` and `ClientGameHandler`.

The first class, which will be instantiated already from the `MainActivity`, is the singleton `ClientGameHandler` class. The first thing this class does is create a new connection in either offline or online mode (more about that in the next sub-paragraph). When this connection is created, the client starts listening for messages from the server. This works the same as on the server. We listen for messages and handle the messages with the `handleInput` method.

Next up the `ClientGameHandler` starts a listener for a change is server mode (online / offline), since it is possible to change this setting in the `SettingsActivity`. Whenever a change of server mode is detected, the `ClientGameHandler` stops the old connection and starts a new one.

The last thing the `ClientGameHandler` does when instantiated is start the `gameLoop` method. This method draws the game on the screen and gets the actions of the player and sends these to the server every tick.

The second and last class which will be instantiated is the `Connection` class. This class either starts a connection with the online server or starts a server on the background and connects to this server. The IP-address of the server is set according to the server mode. This class contains a very important method `sendMessage` which is called whenever the client wants to send a message to the server. This class also contains a `stop` method which is very important to prevent memory leaks (due to not stopped threads) and to be able to start a new connection with a server.

3.2.2 Parties

The server keeps track of all the clients using parties. The usage of parties enables the clients to play with their friends. Whenever a client is connected to the server the client can `create` or `join` a new party. This party gets assigned a random so-called `party id` which can be used by the players to identify their party and join their friends' party.

As stated in section 3.2.5, every party gets updated per game tick. We also want this update message to be used to determine whether the client is still connected to the server. Upon connection with the server, the client gets assigned a party just for that client. The game update messages from the server, sent to this party every game tick, can then be used to determine the connection status. Note that the connection status does not make use of this feature yet, but this is for future implementations.

A client can only be in one party at the time. When the client joins a new party, it gets removed from the previous party. The party also uses the ready status of all the clients to determine if the master of a party can start a new game. This is only possible when all the clients in the party are ready (except for the master, the master-client is set ready automatically when it sends the `StartGameMessage` to the server).

3.2.3 Level design

Levels are generated each time a new game is started. Levels have to fulfill a number of requirements:

1. The Level should consist of many different rooms, connected with roads.
2. The roads connecting different rooms should not intersect (so the graph should be planar).

To generate a level, we start with a small grid that represents the final level on a more global scale. Each cell on this grid is called a `Block` because it represents a cluster of tiles on the full map, and it takes on of the values from the enum `Block` (`Empty`, `Room`, `Road` or `RoomEdge`). On this globalized map of the grid, we randomly pick some cells and designate them rooms. Then, we increase the size of the rooms by adding cells bordering room cells to the room, so we get bigger rooms. All of these room tiles are their own 'island' still. So, we have a lot of individual rooms that are not connected.

To connect the rooms, we see each room as a node in the graph representing the grid and we add an edge between every pair of rooms with length equal to the Manhattan distance between them. This is a better approximation than the Euclidean distance, considering roads only go in cardinal directions. To get a subset of edges such that none of them overlap, we use the minimum spanning tree, which we find by applying Kruskal's algorithm. The method `kruskals()` returns an ArrayList of `Edges` showing all the edges in the minimum spanning tree. We will use these edges to connect all the rooms. We go through all the edges and do a breadth-first-search from the beginning to the end of every edge and connect the two points with the shortest path. All nodes the shortest path passes through are marked on the global grid as 1×1 tiles (as `Road` from the enum `Block`). However, as rooms contain multiple cells, roads are thinner than rooms. Now we have a level that is connected and is planar. To avoid the begin and the end points of the level spawning too close to each other, we also monitored the two nodes which are the furthest apart and put the start in one room and the finish in the other. So, the player will always have to walk the maximum distance.

Now, we are ready to enlarge the miniature grid with which we have worked until now. For that we create a new grid that contains for every cell `blockheight`

rows and `blockwidth` columns, and we simply enlarge the miniature grid, copying every cell from the miniature grid into the bigger grid but `blockwidth` × `blockheight` times bigger. In this process, everything that is `Empty` (so not a road or a room) is converted to an empty tile. Empty tiles are converted to walls.

Now we have an empty level and have to fill it with monsters. We go through all the cells, randomly assigning turning a room into a room with skeletons, with slimes and with slimes and skeletons. If we find that a cell is the end point, we spawn the boss in that cell.

3.2.4 Game rendering

Arcemii makes use of animations and static sprites in-game. These have to be displayed on the screen properly. In this section, everything concerning this is explained.

GameView The rendering of the level with all its entities is done in the GameView class. GameView extends the standard Android View, therefore it has to be assigned to a layout. This is done within the `GameActivity` class (client/activities/GameActivity.java: onCreate). After it is assigned to a layout, the `init()` method is called to initialize all objects used in the rendering of the GameView, using the dimensions of the layout. Then two methods are used to render levels: `updateLevel` and `onDraw`. `updateLevel(Level)` prepares a level to be drawn by converting it's tiles and entities to `RenderItem`s and figuring out in which order to draw them. `onDraw(Canvas)` then takes these `RenderItem`s and draws them to the canvas in such a way that the player is centered. Locks are used between these functions because `onDraw` is executed on a different thread than `updateLevel` and they both use the `renderItems` list.

RenderItem All aspects of rendering a single texture are handled in the `RenderItem` class. A `RenderItem` contains the following attributes:

- `texture` defines which bitmap to draw.
- `x,y` defines the position where the texture should be drawn.
- `refX,refY` defines the position within the bitmap used for alignment.
- `layer` defines the layer on which this object is drawn.
- `animationOffset` the number of frames this object's animation is ahead of the default animation
- `rotation` defines the number of degrees to turn this image counter-clockwise.
- `flip` defines whether this image should be flipped horizontally.

The function `compareTo(RenderItem)` determines which of two RenderItems should be drawn first. The function `renderTo(Canvas)` renders this RenderItem to the canvas, applying all transformations specified.

Texture This class handles the loading of textures, to prevent a `Bitmap` to be loaded every time it needs to be drawn. It has a `HashMap` that stores all textures using a `String` as a key. The key corresponds to the path of the texture within the assets/sprites folder. `getTexture` is a factory method that loads a `Bitmap` to the `HashMap` if necessary and returns the corresponding `Texture`. `getBitmap` returns the Bitmap associated with a Texture object.

Animation `Animation` is a decoration of the `Texture` class, in order to be able to handle animations. It's `getBitmap` method takes the current time into account to return a certain frame of the animation.

Generation of RenderItems RenderItems are created in the classes of the objects they visualize (`shared/entities/...`; `shared/tiles/...`). This was done to prevent shadow classes for each entity that only generates RenderItems. Unfortunately, the entities and tiles are shared between the server and client, which means that a server can't be run without compiling all the client code with it. We didn't find it worthwhile to adjust this before the deadline, as we would run the server from within Android Studio for testing.

3.2.5 Game logic

Before we can explain how the game logic works, a few definitions have to be given:

Tile `Tile`s are the building blocks of Arcemii's levels. A `Tile` can either be solid or non solid, meaning entities can either move through it or not. The most important `Tile`-types are:

Wall A solid tile.

Empty A non-solid tile.

Void A tile outside of the bounds of the level.

Start A tile on which players are spawned at the start of a game.

Finish A tile denoting the final room of a level.

Entity An `Entity` is a non-tile game-object with a specific position within a level. An `Entity`'s position is not limited to whole numbers (where a `Tile`'s position is). Every `Entity` is identified by an unique UUID. The entities currently implemented are:

Player Controlled by the actual players of the game.

Skeleton Enemy of player, shoot arrows and flees from the player.

Slime Enemy of player, attacks the player by jumping toward them.

Boss Enemy of player, teleports and summons slime entities.

Arrow Flies in a certain direction, can either hit player or non-player entities. Deals damage to entities it hits.

Level A level contains a grid of tiles (instances of the `Tile` class). These tiles represent the world. Furthermore, the level contains a list of entities (instances of the `Entity` class). A more detailed description can be found in 3.2.3.

Ability Arcemii is based on the notion of abilities. Players can choose the abilities they want to use in the game at the beginning of a run. Every entity has abilities. An `Ability` can be executed with `execute`. When a `Ability` is executed, its action is performed on the level or the entity that performed the `Ability`. A `Ability` can be executed directly, or in some cases, it can be invoked first. If an ability is invoked, certain parameters needed to execute the abilities are set. For example: if an entity wants to execute the `Move` ability, it needs to specify in which direction it wants to move, so the entity has to invoke the ability with a `direction` parameter. How invocation and execution work in practice is explained in more detail later. What follows is a list of some of the abilities present in Arcemii:

Bow When `executed`: create a new arrow entity.
To `invoke`: The direction in which to shoot, as well as information about wether the it can hit the player or not.

Heal When `executed`: heal the entity that executed the ability
To `invoke`: The amount to heal is needed

Melee When `executed`: deals damage to entities in range of the entity that executed the ability.
To `invoke`: whether to damage players and the amount of damage is needed.

Teleport When `executed`: teleports the entity that uses it.
To `invoke`: The direction is which to teleport is needed.

Additionally, `Move` and `Spawn`, etc, are also abilities, albeit not explicitly for the user. Everything a player can do, is seen as an ability, and every `Ability` could theoretically be executed by every `Entity`.

Game logic explanation Because Arcemii is a multiplayer game, it is split into two main sections: the server and the client. Concerning the actual game, the client only acts as the view, and a little bit as the controller part. The actual game logic happens on the server side. The current state of the game is saved as a property of `Party` because each party controls its game-state. This game-state is saved in the shared object `Level`. The level class contains the current world-state of the game. The game loop on the server updates all parties every `TICKSPEED` amount of time. When a party is in-game, and `update` is called the following happens:

1. **Invoke** The `invokeAll` method of all entities, except for players, is called. This method invokes all abilities that the entities want to execute (based on their A.I.). All executed abilities are saved in a `actions` list. The actions players want to take are added to their respective `actions` when they submit new actions to the server.

2. **Execute** All entities now have a list of the actions they want to execute. The execute method of these actions (abilities) is called and the changes are recorded.

3. **Send updates** All changed entities are collected and sent to all party members.

3.2.6 Enemy AI

Slime As previously mentioned, the slime moves to a player and then attacks it. For this, it first checks which players it can see. The slime chooses its target player as the visible player where the value of (slime.UUID xor player.UUID) is the smallest. This is to ensure that slimes don't all target the same player. It then moves to its target player and attacks it.

Skeleton The skeleton chooses its target player in the same way as the Slime. It also checks whether there are any players within 3 tiles, and computes the average position of these players. If players are too close it moves in the opposite direction of this average position. Otherwise, it starts shooting at its target player. For this, it sets the `shootingStart` to the current time, and after 1.6-1.8 seconds it invokes the `Shoot` ability; in sync with the animation.

Boss The boss chooses a target player in the same way the Slime and Skeleton did. If it can see it's target player and its last action was 1 second ago it does one of three things: teleport, summon an explosion (no visualization) or spawn a slime. The teleport and spawn of slime occur in a random direction; The explosion is directed at the target player.

3.3 Design justification

During the design process, we had to make a lot of decisions about the structure of the project. Because we had to deal with a server-client relation, our project structure became considerably more complex. In a lot of cases, we chose 'the obvious' solution. Because there have been a lot of decisions, we will focus on the less obvious or more outstanding ones.

First, we will justify why we chose to take abilities as the basis of the game. By making it so that every action an entity takes is part of an ability, we created a scalable and generalized system. All entities are equal because they all use abilities to perform actions. The ability `Teleport` was originally only used by the `Boss` entity. We wanted a fourth ability for the player and only had to add this ability to the possible ability pool of the player. If we had coded the abilities differently, we wouldn't be able to just add and remove abilities so easily.

We made use of the `Message` class, instead of plain strings or the object directly, because we could add the `getType` method to the messages and we had a clear understanding of which classes could be sent between the server and the client. If a class extends the `Message` class, care needs to be taken, for example when adding new properties (by adding the `transient` keyword).

We chose to use one single class for one entity. It would be possible to create three separate classes for the different ways an entity is used: on the server side, during transfer, and on the client side. We made a conscious decision not to do this. First of all, we would have to create a lot of classes (three times as many). Additionally, we would need to create transform functions to turn one of the classes into the other when we need the other kind. (We would, for example, need to transform a server-entity to a transfer-entity and then in the client to a client-entity). This decision had, however, a few side effects, which may seem weird at first. The textttPlayer class that is used in the client still has two null-variables for the in and output streams. They are not needed in the client at all but still exist in the client. Furthermore, the `Tile` and `Entity` classes contain the `getRenderItem` method. This method is only used on the client for drawing these objects. Strictly speaking, this has nothing to do with the server, but the `Tile` classes on the server have these methods too. The advantages of using a single class outweigh the disadvantages by a lot. The gained elegance and the easy serialization have value too. Lastly, it may seem that this method decreases the speed or efficiently of the program, but this is not the case. By using the `transient` keyword, variables that only belong on the server or client are kept there and are not sent through the internet.

4 Project management

Introduction

This section contains a log for the process of building our teams second app for the *Research and Development* Course at Radboud University. Our process will be presented in a weekly description of the active tasks that week, project meetings, assignment of tasks, problems we encountered, etc. Before that though, a list of global tasks, and the division of them between group members.

4.1 Task assignment

Arcemii is a rather large project for the time available to us, so it was very important that people could work concurrently and that they not have to wait for others to finish. This proved rather difficult, as there were a lot of parts dependent on each other. Some examples: how would you implement player movement and ability casting, without it being displayed on your screen? How would you make a lobby activity for the party to choose their abilities if there is no client-server connection?

We realized this early on, and also realized that our idea for multiplayer was rather risky. Also, a lot of the game would depend on the multiplayer aspect, and with that the connection aspect, so this *had* to be done first, albeit maybe not in its final form.

- Server-Client Connection: as stated above, this is very important and a lot of the app depended on it. Steven took it upon himself to make this part's 'skeleton'. Bram and Steven became our server experts. Bram was assigned the subtask of making a singleplayer mode using the structure already established.
- Sprites and animations: not a part that is very important code-wise, but is important for the app's presentation. Making Sokoban taught us that visual feedback is a good motivator: once you have something touchable, more concrete, it becomes more fun to work on it. Also, we decided on using animations for this app, and those take a lot of time to make, so it was a good idea to start early with making them. This task was assigned to Thijs.
- Rendering: The sprites had to be displayed on the screen. This was Jelmer's task. Good communication between him and Thijs was needed, as Thijs must deliver Sprites Jelmer was ready to implement or vice versa.
- Game logic: we had talked this through with the whole group a lot, we felt like everyone should be part of the way the game would play out. Steven provided the structure for the game logic. Later on, everyone could add to it (this was doable, because of the way we structured abilities, entities, etc.) Jelmer was assigned the subtask of making *A.I.* (patterns in which they behave) for the enemies.
- Level generation: this was a task separate from the rest, but crucial to making the app a game. This task was assigned to Robert.

- UI and app look and feel. This task could only really be done later on, as it required some structure and game to already be there. Thomas was assigned the subtask of making in-game UI. Thijs was responsible for making the app as a whole look decent. He was also assigned the subtask of adding music to the game.

Overall, it was doable to split the tasks above into smaller ones, especially bug-fixes. As the project became larger, however, it was increasingly harder to be an 'expert' in every part of the app, so that something in, say, the server side could only be added or changed by someone already familiar with it (without too much risk)

4.2 Log by week

Week 19

After the spring break, in which we did not yet start our new project as most would've been unable to work on it, we decided that a meeting on Monday morning would be wise. This meeting took place in Mercator I, which lasted from 10:30 till 12:15, and was attended by Jelmer, Robert, Steven, Thijs and halfway through also by Thomas. The main topic of this meeting was choosing which app idea we would run with. We had three prominent ones:

- A multiplayer dungeon crawler. This idea had us very excited at first, but we realized that this would mean a very similar project structure to the Sokoban app of the previous assignment, so we dismissed the idea.
- An medical application in cooperation with medicine students, as suggested by Patrick van Bommel. This idea seemed cool but did not motivate us as much, as we would have to deal with outside requirements instead of our ideas about what would make a good addition to the app.
- A Mario-party-like game with multiplayer minigames. By now we had settled on the idea of making a game, as we had a lot of fun doing the previous assignment. By making a game centered around minigames, we think we'll be able to make this process fun for ourselves. We'll have to put some effort into making a connection between two phones.

The third is the idea we had settled on about halfway through the meeting. We made some sketched of the project layout and brainstormed on some minigames (like spyfall and charades)

We planned our next meeting for Thursday, the third block. By then, Steven will have set-up a new Github repository.

Thursday we had our next meeting, which was attended by everyone. In the previous days, we had all thought about the idea of something using a server and had become skeptical about how good of an idea it was. We discussed the following risks:

- Our app will probably be tested by one person, but they will have to be able to use the app's full functionality to properly grade it. Robert has sent an e-mail to our TA with questions about this, so we'll await their response and then look at this risk again.
- It is risky that our whole app relies on one connection. If something goes wrong in the connection part, the whole app will suffer from it.
- Four weeks for an app is already a short timeframe. Will we be able to afford the time to spend on multiplayer functionality? This is something we'll have to decide upon once we have more concrete ideas about games we'd like to make.

Because of the risks as mentioned earlier, we further discussed the possibilities of single-player games, as the idea of the app as a game remained unchanged. Thomas and Thijs were charged with the task to think about games in the rogue-like genre, to see if such a game would be a valuable app idea. Steven

will make the structure of a client-server program. This way, we can have some extra days to decide on our idea, as we'd like the whole group to be fully behind it, hard as it is, while also getting started on some code.

Week 20

We had our first meeting of the week on Monday morning, again. Over the weekend, more doubts about the party-game idea had arisen. Everyone was present, though Robert was there only the second half. First order of business was making the final decision for our idea. We all felt this was long due, and we're all a bit frustrated because of this. So: today we were to make a decision and we would stick with it.

Our main two problems with the party game were design-problems: we would have to make some smaller, individually not very impressive things, that would have a very simple structure. This did not feel like enough of a challenge, even the multiplayer aspect is taken into account. The second problem was that it would be hard to make it one well-rounded app. Additionally, because it had pieces that were so disjunct from others, we would evade the whole project idea of this course. To conclude: we switched back to the idea of one game. till multiplayer. The remaining of the meeting, we discussed possibilities. Firstly, we looked at possibilities to expand on one of the discussed minigames:

- Curve Fever
- Bomberman
- Spyfall

However, as we didn't find this complex enough, we looked further. A game like hill-climb racing was discussed, but we did not have concrete enough ideas for implementing multiplayer into that game. In the end, we landed on the idea of a multiplayer rogue-like yet again. Here is a small run-down of how we envision it:

You create a party with your friends. Every player is presented a random selection of 'abilities', of which they can choose some. They enter a level of some kind and fight enemies. At the end of a level, there is a 'boss fight', after which the players are rewarded and can go down to the next level. Some opportunities for player-versus-player were mentioned, like the strongest player being a final boss after all levels had been completed, but we decided to postpone these ideas.

When Robert joined us for the meeting, he and Steven discussed the way we would tackle the multiplayer/server aspect code-wise. They proposed different approaches:

- Steven suggested starting with a full-scale client-server connection. This would prevent us from having to spend much time later on in the project to change existing code to make it suitable for multiplayer and such.
- Robert suggested to first make the game in single-player form and add multiplayer later, as this could prove tricky and time-consuming.

It was agreed that Steven would work on his vision for the next meeting, where we would decide if we would stick with it or not.

Thursday was our next meeting. Everyone but Steven was present because he 'is too cool to use the bus'. We started a Discord-call with him, and he showed us his work from the past days. In the end, he used Robert's approach. Thijs made a little character animation to look at possible styles and the possibility for animations. We'll use pixel-art, as it is less time-consuming while still being charming.

The main item on the agenda was dividing task:

- Jelmer: Graphics rendering
- Bram: Being able to create and join parties
- Robert: Dungeon generation
- Thijs: Sprites
- Thomas: Looking at abilities

Week 21

As usual, we had our first meeting on Monday morning. Everyone was present, though Robert only attended the second half. We started by discussing what everyone had achieved over the weekend. The divided tasks were pretty big, so everyone was still busy with them. Because of Jelmer's rendering, we could see the first animations on screen.

We then discussed the game envisioned in more detail:

- We want players to be able to play the game in short bursts (like in the breaks in the middle of lectures), so we're aiming at 15-20 minutes per total game. That means that we'll probably have 3 levels per game. In the future, we can expand this with an option to choose between a short, medium and long game.
- We'll start with making just 3 abilities, of which the player can choose 2 or 3 at the beginning of the game. The idea is that this will be easy to expand on later.
- For enemies, we'll start with 3 basic ones:
  - An enemy that does damage on touch: a slime. Slime will be able to make a jump attack.
  - An enemy that attacks with range: a skeleton that will shoot at players.
  - A boss that is bigger and does more damage.
- We discussed abilities being on cooldown versus using mana to use them. The majority is pro cooldown, but we'll come back to this later when the implementation becomes relevant.

We set our next meeting for Thursday and gave everyone new tasks (well, most of them are continuations of the last ones). New ones are:

- Steven: make it able to test multiplayer.
- Thomas: work on ability selection
- Bram: finalize the lobby activity.

Thursday was our second meeting. Everyone but Thomas was present. This is very inconvenient, as he did not finish his task.

Jelmer had continued with his rendering and the only thing left is rendering outside of a level.

Thijs was assigned the task to work on some sprites, and he made these animations:

- Idle and walking animations for 4 player characters
- Idle and jumping animations for a slime
- Idle, walking and shooting animations for a skeleton
- A floating animations for a boss
- A tree

Robert is almost done with the dungeon generator, and he'll finish this for next Monday. We made some more dungeon specifications for him to generate: enemy locations and start en goal positions.

Thijs and Bram discussed the layout and style for different activities. Mainly, the lobby activity, where the party waits for the game to start and players choose their abilities, has to change so that abilities will be able to be chosen. This is a task given to Bram for next Monday.

Steven aims to make all game-logic that is not directly dependent on tasks that are given to others.

Thomas is given the task to start on the in-game UI.

Saturday, Steven implemented a simple skeleton AI, but this will need later polish, as the skeletons just run away to the upper-right corner (even walking through trees that are supposed to be solid)

Week 22

At Steven's request, because he had an appointment, the meeting started later than usual. This meant a shorter meeting, as Thomas had to leave us in the break. Sadly, not everyone was on time, so we effectively had only 45 minutes for our meeting. This is very sub-optimal as we start to realize that there is still a *lot* of work to be done before we have a presentable application.

Robert has completed the dungeon generator but will make some final adjustments before he will merge it with the master branch on GitHub.

Steven did some work on passing objects between the server and the client. Still, a lot has to be done in this part before we can properly test everything. A big part of our struggles is that most tasks seem intertwined with others: Thomas would like to work on in-game abilities, but he can't until a proper connection is made. Bram would like to work on the lobby activity, but this has little meaning when there are no abilities.

An important breakthrough this weekend was that Bram was able to join Jelmer via the app! This means the connection that *is* being made, will probably work correctly later on. However, we realize that working with multiplayer was probably a very ambitious idea and the risks we were afraid of in earlier weeks are staring us in the face.

Bram spent most of his weekend working on a bug that causes the player not to enter a good multiplayer mode after they have first switched to singleplayer mode – as a side note: there is a singleplayer toggle added, which just 'tricks' the app into thinking it's local device is the server, for testing and later on for true offline game possibilities.

Thomas has not been busy yet with the UI and controls. We urge him to look at this quickly, we want (theoretical) player movement by Thursday!

Thursday was Ascension Day, and the university was closed. We did not arrange a meeting, as most had family-obligations to attend, but we discussed some important things via Whatsapp so that everyone would still be able to work this weekend.

There were some problems with the dungeon generation (namely that rooms are not connected) that Robert fixed immediately on Wednesday.

Thursday, Thomas added a joystick for movement, but this is not yet fully integrated into the app.

Week 23

This is the final week of this project and because we still have a lot to do, we'll discuss our activities on a day-by-day basis for this week. But first, these are the major things we'll have to work on to make our app ready for submission:

- Most importantly: the game is not yet playable. To change this, we'll have to make sure player movement works, abilities can be cast and entities have health and can be killed. As we had these ideas already when we started coding, the structure for these properties is already present. We'll just have to implement them. Steven will work on the first two points and Jelmer on the last. Problematic is, that Thomas needs to clean up his code behind the in-game UI first.

- Multiplayer will have to work on mobile phones. In theory, this has to work, but the previous assignment has taught us that a theoretically working program isn't everything, especially not on the subject of reliability. To properly test this, the points mentioned above should be implemented first, so that we can be sure the server-client connection handles these requests properly.

- There needs to be a lobby activity where players can choose their abilities.

Additionally, there are a lot of things that need polish, or would be very nice to be implemented as well: the overall UI is pretty terrible and ugly, music would be nice, there are a lot of animations not yet used, while the situations where they should be used are present, you can still walk through walls even though stat should not be possible.

Monday 10:30 was our standard meeting and it was... stressful. Everyone was present. The deadline is creeping closer and closer. We discussed the main tasks mentioned above. Steven will work on movement and ability casting. Bram will make multiplayer ready for mobile phones and look into some nasty bugs. Jelmer will take on enemy UI and in-game logic like not being able to walk through trees. Thijs will add app-icons and a melee attack animation. Thomas will have to make his ability cards in the in-game UI change dynamically

and Robert will have to make the lobby activity, including choosing abilities. Everyone should finish their tasks before an additional meeting planned on Wednesday, 15:15.

Today, we received our Sokoban grade, and this served as a major boost to our morale: we got a 9.5!

Tuesday, an unscheduled meeting happened organically. Thomas was not present, Robert was later. Thijs will make the music player this afternoon. Bram will make is so that the hosting player can only start the game when all other players are 'ready', a state they can toggle to using a button. Players will also be able to press 'ready' when they have chosen three abilities. All these tasks were completed this afternoon. Bram also made the first multiplayer contact where both players can walk (still using emulators though) Jelmer made health changeable and improved the AI.

Wednesday, Thomas was not at the meeting. Steven added most of his documentation (others added theirs directly after writing the code). He also fixed the UI buttons, so the app won't crash anymore when pressing them and also made the player able to shoot arrows. Thijs improved the lobby UI and added general styles in the XML. Bram fixed more of the bugs occurring when trying multiplayer and made singleplayer mode not crash the app again.

At Thursday's meeting, everyone was present. Thomas made these buttons change dynamically (later than the deadline). Today, we had the first mobile multiplayer test, admittedly a bit late, but earlier was not viable due to reasons explained above. And the result was... the app crashed almost immediately. We suspected that Jelmer's server, which we used, was not powerful enough. Steven's priority will now be the optimization of this process. Jelmer made a lot of fixes with the arrows. Everyone but Thomas had a call on Discord for the entire evening making small fixes, testing, polishing. Thijs made a gif for the main activity. Multiplayer runs a lot smoother now. Not very smartly, everyone was quickly pushing to GitHub, causing a lot of merging problems. Bram added the boss' teleport ability to the player's ability options, but this is still buggy.

Today is the presentation and hand-in day! Although it was not planned, everyone was making a lot of small changes this morning: teleport is less buggy, server testing, etc. Testing is a lot of fun now.

Bram and Jelmer gave our presentation to Jan van Bommel, our TA. This went very well. Everything ran smooth and there were no server problems. His joking reaction was 'that we are crazy to have done this', hinting at the amount of work it must have been. He said everything looked good and he did not note any points of improvement. This is good news and made us relax! The afternoon was spent by making more small changes, like scaling the ability cards in-game nearest-neighbor instead of bilinear, before Robert handed in the app.

5 Evaluation

In this section, we look back at our project and discuss parts of the process we'd like to handle differenlty in the future, possible issues or opportunities for our app, etcetera.

Overall, we are satisfied with our product. We think we made something pretty impressive, given we had only four weeks for the project. Multiplayer sounded like an obstacle, too big to overcome. But we did. It feels like we made a game a definitive step up from Sokoban, code-wise at least.

Although we achieved our techinal goals, Arcemii does not (yet) achieve *all* of our goals. Namely: we wanted the game to be something we'd enjoy playing in breaks, for example. This is possible now, but honestly, we haven't been playing the game as much afterwards as we did with Sokoban. Why is this? It would be easy to say that the game just isn't much fun, but we don't think that's fair. We spend a good amount of time brainstorming about the gameplay, so that can't be it. However, we had a short amount of time to make Arcemii, so definetely not all our envisioned features are already implemented. This creates opportunities for the future though! We structured our code in a way, that it is easily scalable. Now that we have the basis, we can add anything we want. This is our main look on Arcemii's future. Some additions we'd like to make, include:

- More enemies and bosses.
- More abilities to choose from.
- 'Meta progress', where you can permanently unlock new abilities.
- Sound effects for getting hit, hitting enemies, etc.

There may be some issues that need fixing before these additions should be made. For example: the ip-address to connect to in multiplayer mode is hardcoded in the code. This means that you cannot connect when the server is not active. For the app to become succesful, this should be different (by using a server that's always on for example, one with a bigger capacity, etc).

Things we learned and our satisfaction with the development process go hand in hand. Surely, all of us learned something about programming, XML, using Android Studio, making animations... The list goes on. But these things were never our main focuspoints when we were in a meeting, for example. The real hurdle was the teamwork. Everyone was confident in eachothers skills, but with Sokoban we learned that not everyone wants to do their tasks at the same time (some have a 'if I work on it now, I won't have to later' approach, while others more of a 'why do it today when it can be done tomorrow'). Because of this, setting deadlines and such admittedly did not go as smoothly as we'd hoped for. While everything came together in the end, it would have been less stressful for everyone if we could've been clearer about when to do things. This is a big learning point.

A UML Diagram