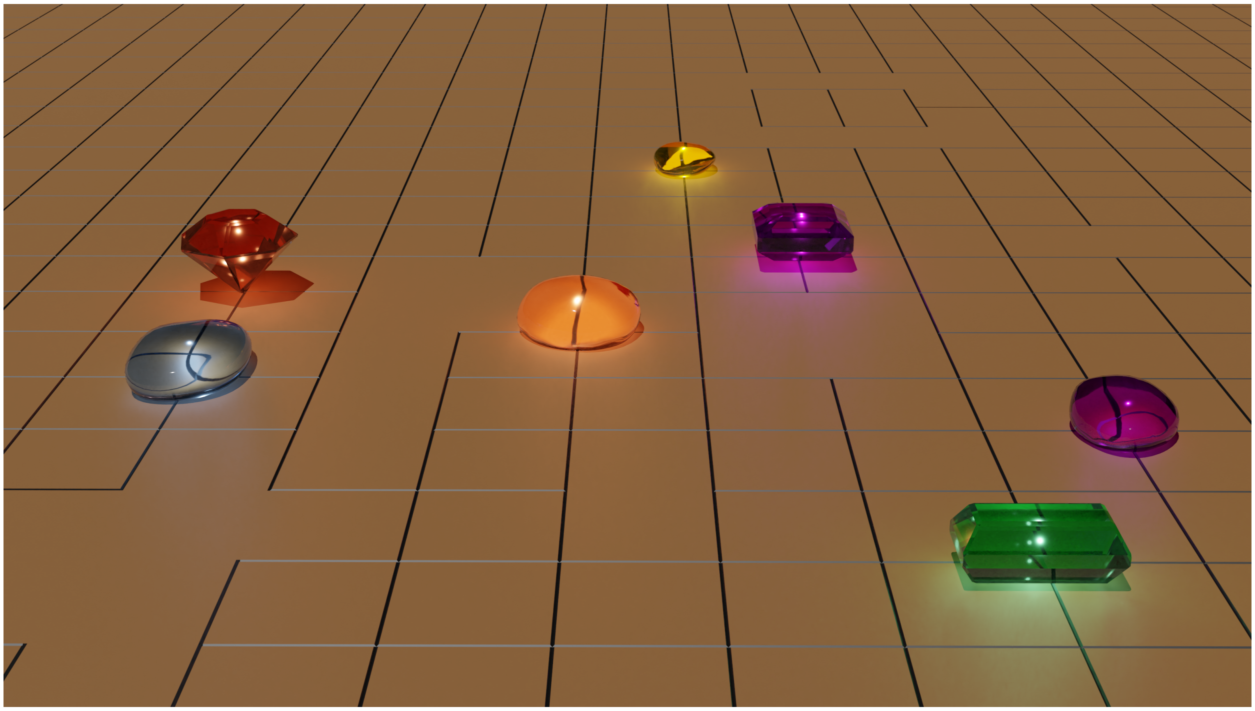
Full Report: *Gemji*

Team *DreiKopf*:

Felix Brendel
Jonas Helms
Van Minh Pham

July 2021



Contents

1 Game Idea

1.1 Game Description

Gemji is a single player puzzle game. The goal of each level is to maneuver a set of Gems across a two dimensional board to specific finish tiles. The level is considered complete when all finish tiles are occupied by the correct Gems at the same time.

1.1.1 Game Design

The game features different types of Gems with different effects that activate after a Gem is moved. These effects can for example cause specific movements of adjacent Gems. The strength of these effects varies, e.g the number of tiles a Gem is moved. As effects trigger on movement, a chain reaction of different effects can be triggered. Depending on the level we also intend to assign effects to specific tiles on the board in a similar fashion.

We created a quick example animation of the gameplay we have in mind, which you can watch on the following page: `https://wiki.tum.de/display/gameslab2021summer/Team+DreiKopf`

1.1.2 Example effects for the Gems

- Black: Purely an obstacle. Cannot be moved by the player or other Gems.
- White: Cannot be moved by the player, only due to the effects of other Gems.
- Red: Pushes away other Gems that are next to it after being activated.
- Blue: Will always move back to its original position after being moved by another Gem effect. When it is moved by the player it will stay in the position.
- Green: Pulls Gems that are positioned next to it towards itself after being activated.
- Orange: Pushes itself from other Gems or the level boundary if it was activated.

1.1.3 Visual Clarity

As players should be able to think ahead more easily, we want to ensure that effects attached to the Gems can be identified instantly. For that reason we plan to add specific visual effects that represent a certain class of Gem ability. Furthermore we are thinking about adding tooltips to the Gems, that show up when moving the mouse over them, which are not invasive to the gameplay but still help the player to learn the Gem mechanics.

1.1.4 Campaign & Level Design

The levels themselves will be designed by hand to ensure a suitable difficulty curve for the players as they learn the game concepts. The levels will be structured in a campaign that will introduce new Gem types and the dynamic map effects in a step by step fashion. We further plan as one of our high-goals to have levels procedurally generated thus creating an endless play mode that can be played after finishing the campaign.

1.1.5 Order & Chaos

The way we want to incorporate the theme of *Order & Chaos* is in relation to the gameplay mechanics. One movement of a Gem can easily cause a chain reaction that may seem chaotic initially but the effects are deterministic and the Gems clearly indicate which exact effect is assigned to them. A light-up indicator also highlights the order of execution of the effects. Therefore *Gemji* is a game which has the goal to find the correct order, so occupying the finish tiles with the correct Gems but may be perceived as chaotic in the execution of the

1.2 Technical Achievement

1.2.1 Introduction

The central secondary big bullseye idea for our project is to develop our game idea in our own engine. Our group always wanted to build their own game engine from scratch and we thought that this practical provided the perfect opportunity to put this into reality. We already started working on this engine in the winter semester practical and developed a puzzle game from scratch. We want to use this practical course to further expand on the engine and see how well we can readapt it to our new game.

1.2.2 Motivation

The main motivation to build our own engine stems from the fact that we believe that we can reduce the overhead and therefore provide better optimization for our games on all levels of the engine, from the graphics pipeline to resource allocation and automatic memory management. Furthermore we believe that building a game engine from the ground up presents a perfect learning opportunity, especially when trying to find suitable optimizations that fit our design philosophy.

1.2.3 Game Engine

In the following sections we will provide a small overview of the components of the game engine that we want to develop for this semesters project and how we try to optimize these. Furthermore we will go over the features of the game engine that we will most likely tackle in the follow-up project and how we solve the interim solutions for this semesters game.

1. Graphics pipeline

   The game engine will use the Vulkan Graphics API to implement a rendering pipeline. Vulkan is a relatively new API developed by the Khronos Group (maintainer of OpenGL) with a focus on overhead reduction and was released in 2016. Vulkan provides a low-level control over the rendering process when compared to other Graphics APIs and has several advantages that also align with our overall philosophy in the design of the engine:

   - The ability to run on all operating systems and devices
   - Explicit control over memory management
   - Decreased CPU workload due to reduced driver overhead and batching
   - Making use of the driver independent Vulkan Loader to access Vulkan API entry points

   The Vulkan Loader is responsible for transmitting Vulkan API calls to the appropriate graphics driver. This means that we just have to connect to the Vulkan loader in our engine and do not have to worry about drivers. Furthermore we can pre-compile our shaders into the SPIR-V binary format instead of compiling the shaders at runtime. This allows the use of a larger number of different shaders per scene and reduces application load times.

2. Overhead reduction in the engine

   The game engine is developed in the C++ language that all of our team members are familiar with due to our TUM Bachelor courses such as Game Engine Design. We have also taken further steps into the direction of our core concept of overhead reduction by omitting parts of the C++ standard library, that perform memory allocations.

3. Resource Loading & Automatic Memory Management

   To increase the performance of the engine we want to make sure that the loading of resources such as a texture map or a mesh is never done redundantly, which is likely the case in a puzzle game as key components are similar between different scenes. In order to implement this we allocate buffers upfront to store all our resources and a hashmap that maps the file paths of the loaded resources to their pointers in memory. If a resource becomes necessary in a scene, we can cross check whether the file path has already been loaded and then reuse the already loaded file instead of reloading it. This means that we will only load the difference between two levels which will reduce load times and create a smoother gameplay experience for the player. The hashmaps also provide further advantage for the memory management as we can free the memory and GPU memory for the texture resources by iterating over the hashmap and can incorporate this in the scene load/unloading process.

4. Sound System

   Sound is very important to our design goal of creating a puzzle game as we believe that it has a relaxing or even focusing effect on the player. We will use the already existing sound system from last semester that is developed with the help of the IrrKlang library. We want to further expand on our already implemented functions to play sound effects and looping background music to also enable smooth fading between tracks and triggering a natural change for the music in response to specific game events.

5. Physics System

   The current point of view in our team is that we will not implement a physics engine as part of this semesters project as it would exceed the scope of the engine building aspect. We will instead use keyframe animations and bake the limited number of physics interactions directly into the animations or generate them procedurally. This also comes with the advantage of having a tighter control over the Gems. The interactions of the Gems should not be simulated in a physics based fashion but rather deterministically executed.

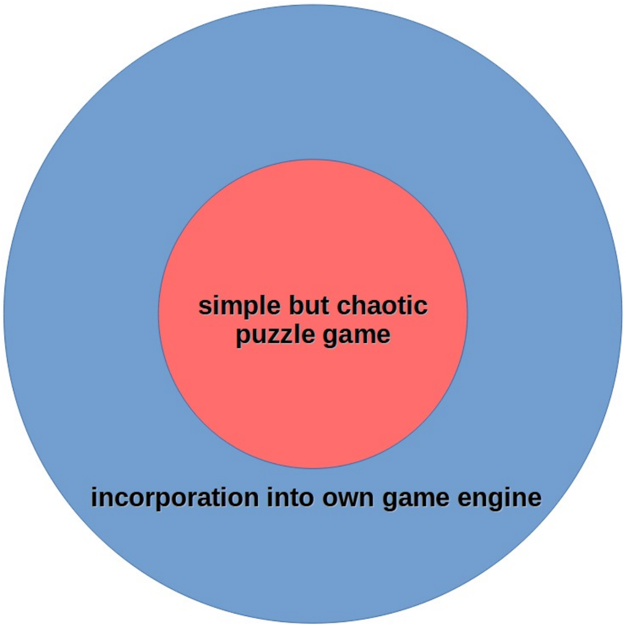
6. Animation system

   The animation system will be a very important part of the engine as it will substitute our physics interactions and help to increase the graphical fidelity of the game. Implementation of the animation system will start very early on and the core functionality of keyframe animation will be finished for the interim demo. We aim to implement a system where actions and animations can be scheduled in advance to be able to deterministically describe the Gems movements.

7. Particle System

   We feel like a particle system and thus interesting effects can add to the correct vibe of the game, especially in puzzle games in which you can easily get lost in the mathematics and logic of the problems instead of the game world. At the same time we have to be careful that these effects do not feel overbearing and contribute to the overall feel of the game. Sadly we were not able to implement a particle system for last semester's project which is the reason why we want to increase our focus on this aspect for this practical course and have already alotted time for it in the schedule.

1.3 Big Idea Bullseye


simple but chaotic puzzle game
incorporation into own game engine

1.4 Development Schedule

1.4.1 Layers of Development

1. Functional Minimum:
   - One simple level
   - Basic selection and movement of a Gem
   - Finish condition for a level
   - One sample Gem effect
2. Low Target:
   - Effect(s) for Gem types
   - One level designed for each type
   - Sound effects & Music
   - One sample map effect
3. Desirable Target:
   - Visual clarity for effects & tooltips
   - Visually appealing particles

- Dynamic board effects
- Full campaign progressively introducing game concepts

4. High Target:

    - Procedurally generated levels

5. Extras:

    - Mobile platform

1.4.2 Tasks

1. Functional Minimum: a. First simple level

    - No "obstacles"
    - Easy finish tiles/conditions
    - Only a few Gem types

    b. Basic selection and movement of a Gem

    - Multiple Gems instead of one (qubi)
    - Targeting positions on the board
    - Targeting other Gems
    - Movement as a do_{slide} for 1 field?

    c. Finish condition for a level

    - Structure of game state considering
        - Multiple Gems
        - Dynamic Gem & map effects

    d. One sample Gem & map effect

    - Set up game logic for effects
    - Already implement the logic considering that effects may change during a level

2. Low Target:

    - Effects for different Gem types
        - Design and test effects with physical prototype
        - Implement the effects
    - One level designed for each type
        - Design levels
    - Sound effects & Music
        - Write first track for the game
        - Implement seamless change of tracks with IrrKlang
        - Look for fitting SFX or create own

3. Desirable Target: a. Visual clarity for effects & tooltips
    - Specific shaders to highlight turn order of Gems
    - ImGUI pop-up tooltips at Gem locations

    b. Visually appealing particles
    - Particle spawner

    c. Dynamic board effects
    - Design board effects that would enhance the current set of effects

    d. Full campaign progressively introducing game concepts
4. High Target:
    - Procedurally generated levels
5. Extras:
    - Mobile platform

1.5 Assessment

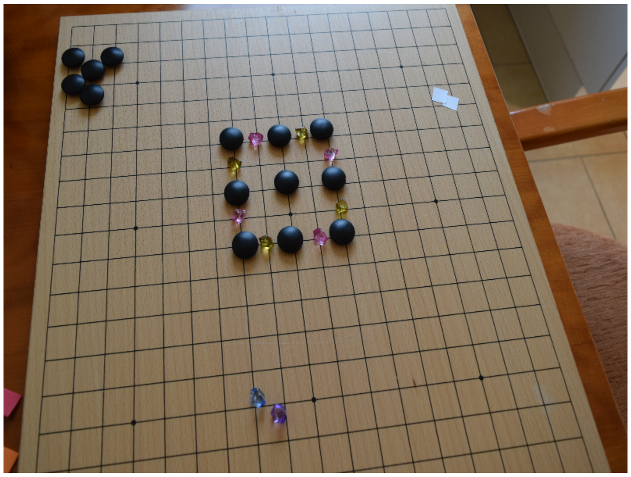
The goal we set for ourselves with this project is creating a game that is easy to get started with, which can be played whenever the players want to relax in a casual setting. A game that is intriguing by its simplicity but at the same time complex on a level that requires tactical thinking and decision making to succeed. This is our interpretation of "order and chaos" – restoring order in a system that is defined my simple rules that spiral out into chaos. It's a game that gives the players the space and time they need, it is not about solving a level as quickly as possible, rather it is about letting the players feel accomplished when they solve a level. The focus is more on creating an environment in which the players feel relaxed and maybe even cozy when they play our game.

With these goals we are confronted with a few design issues that we need to solve. With a puzzle game that is intended to be learned by the players without invasive tutorials, players can easily feel lost, or get the feeling, that they do not yet have all the necessary knowledge to solve a puzzle and get frustrated. We will have to find a way to keep any frustration to a minimum. With the game design we intend, it would be possible for a level to become unsolvable after a wrong move. We will therefore introduce a undo and restart feature, that lets the players restart the level or undo the last move they made. Making mistakes is part of solving a puzzle and we do not want to punish the players; they should be able to try out their ideas and if they want to go back, they can.

2 Prototype

2.1 Prototype

With *Gemji* being a tile-based puzzle game we decided to build a real life approximation of what we had in mind for the game. The physical prototype stage was also something we were looking forward to, in order to refine as well as test mechanics and rules that we had in mind. The gameplay concept we envision for our game is highly fitting for a phyiscal prototype which gave us ample opportunity to follow-through with this approach. For simple levels one player was usually enough to enforce the rules and play the level at the same time. This encompasses the player move, activating the effects and considering the correct order of the chain reactions. Later on when we were testing more complex levels, one member was playing the level while the other team member was double-checking that the rules were implemented in the correct way. This allowed us to test several configurations for Gem effects and rule sets. In the following section we present the tools and materials we used as well as the levels we experimented with.

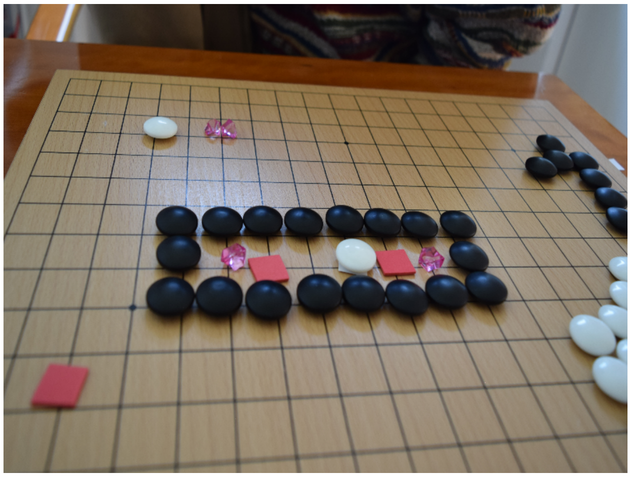


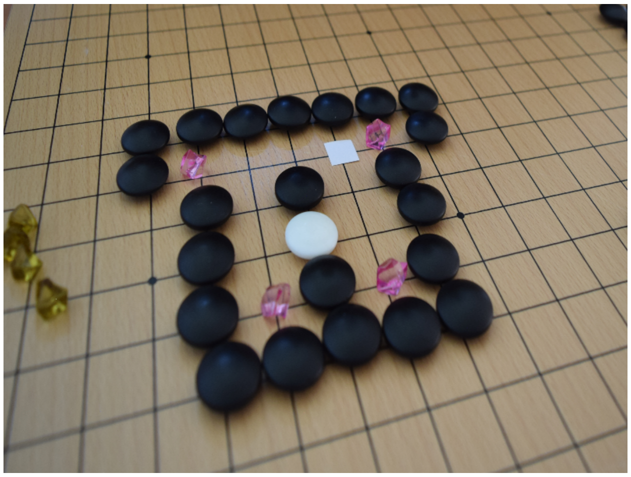
2.1.1 Prototype Setup

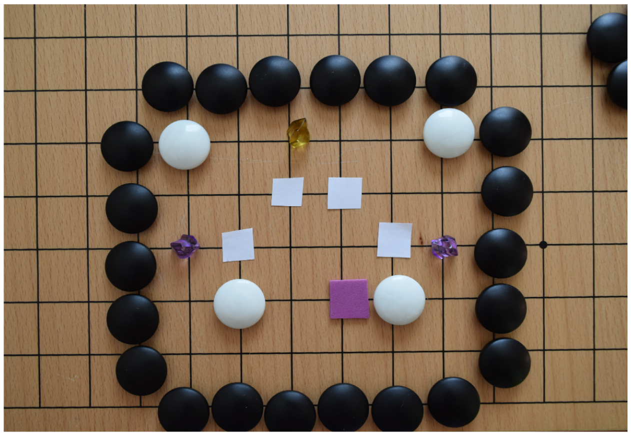
For the prototypes we used a Go board as basis. Additionally we had a variety of game tokens from other board games at our disposal which we used as Gems in our levels. To represent the finish tiles we cut out some paper and foam markers. Using the colors of the gems/stones we experimented with the following types:

- Red gems: push back adjacent gems by 1 tile
- Yellow gems: pull in gems that are at most 2 tiles away
- Blue gems: teleport back to the original position if the position remains free
- Purple gems:
  - if moved by an effect: swap position with the gem whose effect moved it
  - if moved by the player: swap position with the closest gem in clockwise order
- White stones:
  - cannot be moved by the player but by other effects
  - do not trigger other effects
- Grey gems: same as white gems but trigger other effects
- Black stones: serve as obstacles

2.1.2 Example Levels







2.2 Rules & Turn structure

To create consistent gameplay we had to agree on a deterministic set of rules that fit the gameplay we envisioned for *Gemji*. One of our goals for *Gemji* was to create unexpected gameplay realised by the the complexity of the chain reactions. At the same time we identified early on that we have to limit the length of the chain reactions as infinite loops of reactions would otherwise occur. To limit the number of chain reactions we first had to define what a player's turn is in *Gemji*.

2.2.1 Player Turn

1. The player's turn starts when the chain reactions & effects of the previous turn are done resolving
2. The player then selects a Gem that can be moved
3. After the selection the player moves the Gem to one adjacent field on the board along the paths on the board
4. The first effect to activate is the effect of the Gem that the player moved after it has reached the new position
5. By resolving the effect of this first Gem the positions of the other Gems on the map (usually the neighbors) are influenced
6. All the influenced Gems are then primed to activate their own effects and are put into the resolve queue in a breadth-first order
7. This continues until the queue is empty and all effects in the queue have finished resolving

2.2.2 Rule set

When testing the prototype we realized that the specification for the order of the player's turn is not enough and we need further rules to create consistent gameplay. The main issues that had to be solved are explained in the following section.

1. Order of activation

   When a Gem is resolving it's effect it can usually influence several other neighbouring Gems at the same time. Due to the underlying deterministic nature that we want to achieve for our game we needed to agree on a rule that determines in which order new effects are added to the resolve queue, which were triggered by the same effect. After our playtesting session we decided that a clockwise order would fit the game the best and also felt the most natural. Changing this rule could have consquences down to road in the way levels have to be by the players.

2. Infinite chain reactions

   The problem of infinite chain reactions was an aspect that we were already aware of when we first thought up the base concept of the game. The two main ways we discussed to tackle this problem were either limiting the total amount of effects that could be activated per turn or limiting the effect activation for each Gem in each turn. After playtesting the physical prototype we decided on the latter option for now, by letting each Gem have it's effects only activated once per player turn. By limting the amount of effects in this way we realized that the game felt much more like a puzzle game. Additionally the order you chose to move the Gem was now much more significant and that the players would try to avoid the *chaos* of chain reactions by activating Gems in an *order* that minimizes interference.

3. Pattern of influence & neighbours

   One important question we had to decide on, was the degree of freedom on which the Gems interact with each other. The physical prototype was played on a Go-board which has a regular pattern of

nodes which each connect to four neighbouring nodes in horizontal and vertical directions. This was helpful when trying out the physical prototype as it clearly showed which Gems are neighbouring each other. After testing some levels of the prototype we decided that we want to stick to the four directions of interaction that the Go-board provided for both the movement and the consideration of what a neighbour constitutes and not allow for effects influencing Gems in diagonal directions.. This limitation again had implications for the gameplay that we enjoyed during the testing session. One example of this were emerging gameplay patterns that allow the player to aptly move Gems around a corner and thereby not influencing other Gems.
Changing the layout of the map to allow an increased number of directions is something that could provide extra depth for the game in the 'Extra' Layer of development.

4. Move & Activate effect

   During the testing of the game we realized that there are differences for an effect activated by a player move and a chain reaction Gem. This is mainly due to the fact that we want to incorporate Gems that affect another specific Gem, mainly the one that it was triggered from. This also means that we have to differentiate the effect for the player move and the chain reaction activation, when only the target that is designated. At the same time this could lead to interesting gameplay decisions for further Gem designs that we want to explore in the later parts of the project.

2.3 Observations & Revisions

2.3.1 Game Mechanics

When playing the game on a physical board we noticed that the game encourages the players to think about their moves rather than trying random things. This is due to the chaotic behavior of the Gems. Trying out random moves without thinking through the chains of actions that will happen, will result in an unexpected result most of the time.

In the levels we created until now, we further noticed, that the solutions avoid the chaotic effects by trying to move the Gems apart from each other so no unwanted chain reactions are set off. While this is already a challenging game mechanic, in the future we also want to create levels, that use chain reactions to solve the levels, instead of just trying to avoid them.

2.3.2 Emergent Effects

A valuable outcome of this physical prototype was the knowledge that we gained about some emergent effects of the Gems. These are effects, that a constellation of specific Gems has that we initially did not think of. For example a chain of yellow Gems in a line can form a train, when one end of it is moved along the line that the Gems form. We also realized, we can build simple logic gates using the Gems such as and-gates and or-gates, that transfer signals in the form of Gem activation impulses through a network of Gems. This can then be used as the key idea in a level. The player would have to notice the effect or meaning of the given constellation to solve the level.

2.3.3 Tool-assisted level generation

The levels we created for this prototype are quite simple. For most of them, there is a simple mechanic underlying the level design. It turned out to be very hard to create levels that are possible but at the same time challenging to solve. While this might be due to our own lack of knowledge of emergent effects between Gems, we decided to try an tool-assisted approach at level design.

We want to be able to quickly generate a large number of solvable levels, which we as a team can study together with the computer-suggested solutions, to get a better feeling about how Gems can interact with each other to solve levels.

Maybe some of these generated levels will find a place in the campaign of our game, but even if they do not, we still hope we will get valuable insight about level design in *Gemji*.

3 Interim Demo

3.1 Tool assisted level generation

After the prototyping process we decided to create a level generation tool to assist us in finding interesting Gem configurations. In the following we present the components of our level generation tool that creates random levels with specific restrictions that are guaranteed to be solvable in a certain amount of moves.

3.1.1 Random Generation

As a basis we want to randomize specific aspects of a level. These aspects include the board size, the number, type and position of Gems and the number and types of finishes. To prevent an unsolvable level we ensured that the number of finishes do not exceed the number of Gems. The generator further does not place new Gems and finishes at tiles that are already occupied. For the randomization we made use of the C++ standard libraries. The generator then passes the generated playing field to the solver to ensure that the level is indeed solvable and to assure that the count of the necessary turns are in the desired bounds.

3.1.2 Level solver

Since it is important to us to know the minimum number of turns to solve a level, the solver should always find a solution, to which no shorter solution exist. Still, there could be multiple solutions with the same move count; in that case it should find one of them. For this we use iterative deepening (depth first) search. We gather all possible moves in a certain board position and try them out recursively until we reach the maximum search depth. Since this algorithm follows a brute force attempt, the search domain grows exponentially. Still there are some easy ways to speed up the solver a lot.

1. The naive way of implementing this kind of search would copy the board to each recursive function call for it to try out a certain move. Because if you do not do that, you would modify the old board and if the move did not work out at the end you would have to have a way to "undo" a move with all the chain reactions that it caused. The easiest way to archive that is by making a copy of the board and passing it to the recursive call. We noticed, that doing these kind of memory allocations on the heap via `new` or `malloc` slows down the solve. Even with a moderate level `malloc` would be called 20.000 times, which is unacceptable (even though it solved the level in less than a second). Instead we opted for stack allocatations, which proved to be much faster. The general problem that we encountered and the reason why we did not just use stack variables (with implicit stack allocations by the compiler) is that boards can be arbitrarily large. Instead we used `alloca` to allocate memory for the function on the stack.

2. Much more impactful than the memory optimizations was the simple observation that we do not have to continue down a search path, if the board configuration in question has already been processed. This helps in two ways:

a. **Avoiding loops**; Avoiding game positions that were processed in this current path from the start position. Reasoning: Going to the exact same game configuration as on a previous search depth will never find the shortest solution. Two example configurations where circles can be avoided are shown in figure 1. b. **Avoiding dead ends**; Avoiding game positions that were processed before,

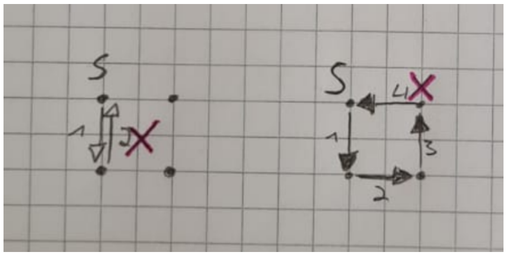


Figure 1: Board positions that were encountered during a search path, will not be considered for the search as they cannot possibly lead to the shortest solution

but in **another path** (and maybe even with a different maximum number of moves) from the start position, from which we know that a solution could not be found from them. An example can be seen in figure 2.

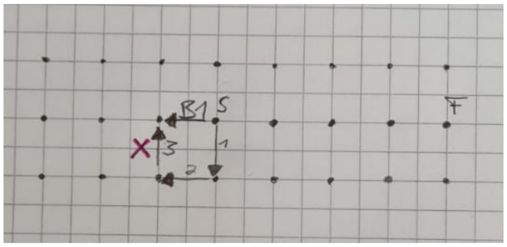


Figure 2: After all possible solutions (max moves = 4) starting with B1 have been checked, later solutions do not need to consider moving to B1 because it is a dead end.

Just because a board position from a previous search path did not lead to a solution however, does not necessary mean, that there is no path that finds the target position that visits the board position in question. This is illustrated in figure 3.

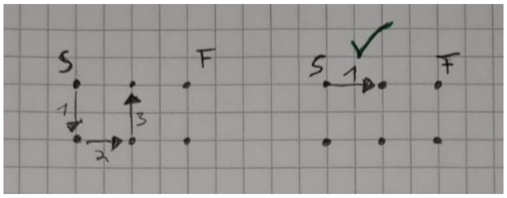


Figure 3: Even though the board position (max moves = 3) after [3] on the left has seen before, it does not necessarily mean that it is a dead end, it could be on the path to the finish, just not on the shortest one

Solution: if the solver with n moves left, visits a board position with was found before with m moves left and $m > n$ then the solver can be absolutely sure that it will not find the solution from trying the move in question and can therefore avoid dead ends.

With these two mechanisms many possible search paths are pruned without changing the solution, only avoiding redundant computations. However we are then faced with another problem. How can you efficiently determine if a board position was seen before and also as (b) requires, at what search depth it was seen? The natural way to solve this problem is by using a hash table that maps the board positions to the minimum depth of the search tree where it was encountered. Still, a hash table would do a quality comparison if the hash value matches, but we really wanted to avoid full board comparisons.

Internally the 2D board is represented by a flat array of integers, that store the index into another array that stores the Gems itself. Internally it does not suffice to only store the Gem type at each position, as we need additional information per Gem, for example, if the effect was executed this turn, or by which other gem it was activated. Further two board positions A and B between moves (meaning no pending effects) are considered equal, if (and only if) they have the same gem types at the same positions.

So to check if a board position was already encountered using a hash table as described above, we should find a hash function for a board configuration. Additionally we want to avoid full board comparisons. To solve both problems at the same time we use integer factorization, which proves that the product of primes to integer powers produce unique integer results. We then assigned each grid cell of the board a prime number and use the enumerated gem types as exponent. An illustration is shown in figure 4.

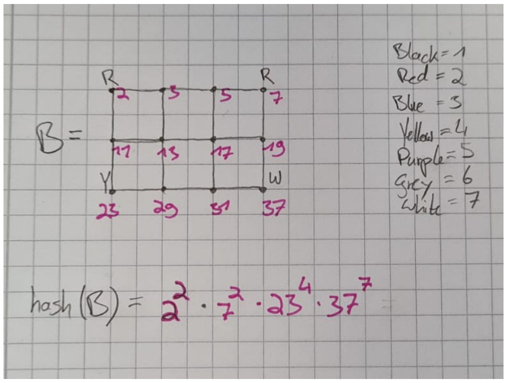


Figure 4: The construction scheme to generate an integer from a board configuration using integer factorization; each position on the board is assigned a prime number while the gem types are the enumerated exponents.

With this we found a method to uniquely represent board positions as integers. By prime decomposition it would also be possible to reconstruct the board from the resulting integer, although we currently do not make use of that. We use unsigned 128-bit integers to represent the boards in that fashion. However we have to note that with huge boards, integer overflow could happen. In that case a reconstruction seems impossible. More importantly, in an unlucky case two board configurations could produce the same hash value. This would have the effect that board positions are considered dead-ends even though they were never encountered before which in the worst case could lead the solver to not be able to find a solution even though it exists. To solve this problem a big number representation would be necessary, which would address both problems. But since this only affects much larger boards, we ignore this effect and assume the integer representation is unique. As a result we were able to efficiently implement the hash table mapping from unsigned 128-bit integers –representing the board– to integers –representing the minimum tree depth they were found at.

Another concern that should be addressed, is using a search heuristic to find solutions quicker. At each board position when we gather all possible moves, we could order them in such a way that the most promising ones are processed first. A simple heuristic would compute the sum of all the manhattandistances from each gem to its finish position. This would not change the result, as still all possible solutions would be checked, just in a different order. Another consideration would be to move away from iterative deepening and use another heuristic search, like A*. We are however unsure if our heuristic would satisfy the conditions for A* to find the shortest solution.

3.1.3 Heuristic for generated Levels

The aforementioned aspects for the random generation are not all elements that we use to control the generated boards with. Deciding to use randomly generated levels from the get go is something that introduces its own type of issues, mainly that we cannot directly control and determine how fun or interesting one of the generated levels will be. To combat this problem we thought about creating a heuristic that would somewhat incorporate aspects that we think will make a level fun to play. First of all we divided the aspects that influence this heuristic into elements that are used directly in the generation of the board and solution and secondly elements that are determined afterwards depending on the solution.

1. Pre-conditions (the inputs)

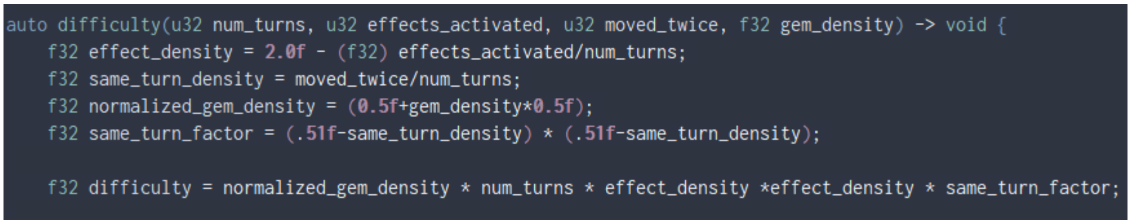
   - Board size
   - Gem count/density
   - Min/Max move count

The pre-conditions are used as input for the level generator but also influence the heuristic.

1. Post-conditions (check the interestingness of a level)

   - Was every gem moved atleast once?
   - Are you mostly moving a single gem?
   - How many effects were activated?

The post-conditions are determined once a solution for a newly generated board, depending on the pre-conditions, was found. In order to do this the board is cloned and the solution played through the board while these metrics are recorded. For now the heuristic is mostly considering the number of turns, how often the same gem is moved, how many effects are activated for the solution and how many gems are present on the board compared to board size. This heuristic is still not final as we are still learning new aspects of our game throughout the process of creating randomly generated levels and will improve if further. For now it determines a difficulty score ranging from 0 (very easy) to technically infinite but in practice 5 (very hard) as we limit the maximal board size. We want to further observe the random generation and try to find aspects that determine a some sort of fun factor for the generated boards.



```
auto difficulty(u32 num_turns, u32 effects_activated, u32 moved_twice, f32 gem_density) -> void {
    f32 effect_density = 2.0f - (f32) effects_activated/num_turns;
    f32 same_turn_density = moved_twice/num_turns;
    f32 normalized_gem_density = (0.5f+gem_density*0.5f);
    f32 same_turn_factor = (.51f-same_turn_density) * (.51f-same_turn_density);

    f32 difficulty = normalized_gem_density * num_turns * effect_density *effect_density * same_turn_factor;
```

Figure 5: Difficulty equation

3.1.4 Campaign & Level Design

The major aspect we want to use the random level generator for is to allow us to implement an endless mode for later on but just seeing the possible solvable board states helped us a ton in the start of our campaign design the we also want to include. The campaign should teach the game mechanics from very simple moves to more complex emergent play patterns. The first levels that we have generated have already helped us to find cool concepts that we definitely want to introduce in the campagin as well but have also shown us that, especially for the early levels, handcrafted design will be the way to go. One main issue is that we want to introduce the gem effects one by one so the player will more easily be able to remember them and not get frustrated. To generated levels that only use one or two gem types we will have to further restrict the pre-conditions for the generation by not only restricting the number of gems but also the number of different types present. One fear is that the thus resulting level will be too one-dimensional, but this is also an aspect that we might include for our heuristic.

3.2 Game Progress

In terms of game development we have been working on two tasks concurrently. On the one hand we implemented a full game loop that enables users to move Gems with console inputs. This was based on the randomized level generation program so we would be able to test the generated levels right away. It encapsulates the complete game logic that we have planned for this phase of the development and will most likely serve as the basis for the game logic of the true game. The other aspect we have been working on is the reusage our game engine from last term's game "qubi".

3.2.1 Console Game Loop

The game loop of *Gemji* is rather straight forward. The player gets the option to select a gem and then move the gem to an adjacent tile. Depending on the color of the Gem a certain effect is triggered which will influence surrounding Gems which in turn will have their effects triggered as well. As of now we have implemented the following Gem types:

Red These will push surrounding Gems away by one tile.

Yellow They pull Gems in that are two tiles away.

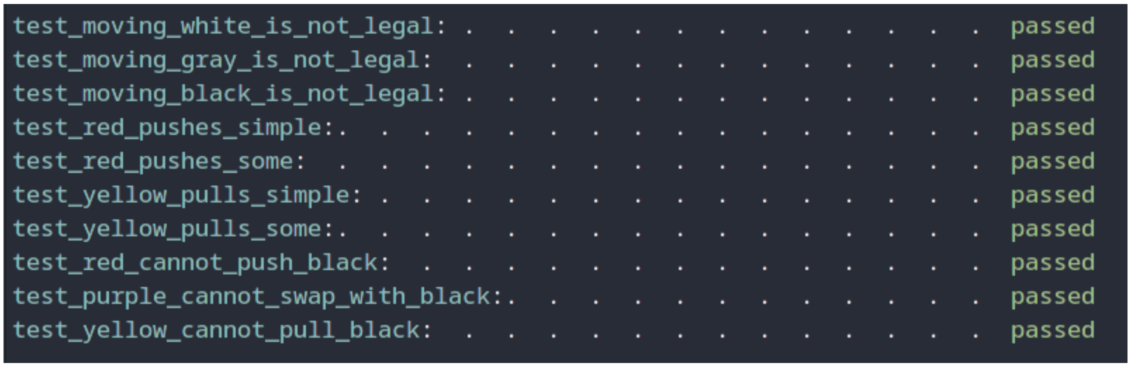
Blue They teleport to their original position back after being moved.

Purple These swap positions with the gem they were triggered by. If these Gems were moved by the player they swap positions with the nearest Gem in clockwise order.

Additionally *Gemji* features white and grey Gems that cannot be moved by the player and only move from other Gem effects. Furthermore grey Gems are the only Gems that do not trigger other effects while white Gems do. Finally we included black Gems that serve as immovable obstacles.
Along with the effects we have successfully implemented the win conditions for the levels.

3.2.2 Test Cases

Since it is really easy to accidentally introduce errors in the code, we were looking for means to ensure the effects work as expected and all possible moves are determined correctly. To address this, we wrote a small test suite that can be extended over time to check specific test cases. With this in place we can confidently refactor the code and run the test suite to make sure no errors were introduced.



```
test_moving_white_is_not_legal: . . . . . . . . . . . . .  passed
test_moving_gray_is_not_legal:  . . . . . . . . . . . . .  passed
test_moving_black_is_not_legal: . . . . . . . . . . . . .  passed
test_red_pushes_simple:.  . . . . . . . . . . . . . . . .  passed
test_red_pushes_some:  .  . . . . . . . . . . . . . . . .  passed
test_yellow_pulls_simple: . . . . . . . . . . . . . . . .  passed
test_yellow_pulls_some:.  . . . . . . . . . . . . . . . .  passed
test_red_cannot_push_black:  . . . . . . . . . . . . . .  passed
test_purple_cannot_swap_with_black:. . . . . . . . . . . .  passed
test_yellow_cannot_pull_black:  . . . . . . . . . . . . .  passed
```

Figure 6: A successful run of the current test suite

3.2.3 Repurposing the old Game Engine

One goal for our group was to develop *Gemji* in our own game engine. As we did not want to start from scratch we wanted to use the game engine from last semesters project as a base and reuse as many parts as possible. For now wanted to focus on keeping the rendering pipeline and the resource allocation.

1. Vulkan Graphics API

   For our game engine we used the Vulkan Graphics API which is very time-intensive to set up due to the amount of control it provides to the developers which is why we wanted to reuse as many parts as possible. For now we were able to keep all parts of the complete rendering pipeline from the Vulkan initialization, buffer structure and more.

2. Resource Allocation & Management

   For the resource allocation & management we were also able to keep all the of the code as was developed in a general purpose way. It is set up in such a way that we do not have to load the same assets twice to conserve memory and time.

3. Scheduler

   As our last game 'qubi' was also turn based game it was natural to use the scheduler that ran our events and animations in the background. The scheduler from our old engine is set up to work well for turn based games and was easily implemented for our new game.

4. Game Logic

   The last step was to refactor the logic we developed for the console game into the new *Gemji* core with the parts from our old engine. We wanted to do this as as soon as possible so we would not get conflicting implementations. Right now the console game as well as level generator and the main game are built automatically from the same source code in the repository thus ensuring their equivalence. First the console game is built with the aforementioned tests that check whether any new implementations broke the game logic that we set out to design.

3.2.4 Hit Scanning

The way we intend the player to select and move Gems is by using mouse inputs. To achieve that we implemented a basic raycaster into the repurposed engine since it was an aspect that did not use for our old game. The purpose of this raycaster is to converts the screen space coordinates of the mouse position to a ray in world space and thus a point on our game board or token. We implemented this with a simple backprojection using the inverse of the view and projections matrix. Since our game is played on a flat board, we knew the height (z-value) and can solve for the board position without working with any g-buffers or check the geometry of the game objects.

4 Alpha Release

4.1 Game Logic Implementation

We previously had the game logic and the rendered scene implemented separately. The game logic at the time was running as a console application. While being 100% functional, this is far from the game we had envisioned. The scene in our engine that was rendered showed the board along with the gems but player interaction with the scene was missing entirely. Our main focus therefore became to combine both elements. The center aspect that is used for interaction is our raycaster that was implemented for the Interim demo. When the player clicks on a spot on the screen with the left mouse key, the raycaster determines the intersection of the resulting ray with the board (as described in the Interim demo). The 3D float coordinates of this intersection are then rounded to obtain the according 2D integer coordinates on the playing field. Since we know where our gems are and how big they are, we can determine whether or not the player clicked on a gem. To then move this gem the player has to hold the left mouse key and drag the gem to another location. On the key release the target coordinate on the playing field is determined with the raycaster as well. Should the target position not be viable- because it is more than one tile away from the original position, already has a gem or is out of bounds- the gem will instantly snap back to the original position. Having completely incorporated the game logic into our engine the game now performs the turn including the chain reaction after the player has moved a gem while also checking that his move is legal. Since the game logic was finished beforehand, this boiled down to a single function call.

4.2 Transition to 3D

As well as the game logic works in the engine, providing visuals to the player is vital to ensure a satisfying and enjoyable playing experience. At any time the player has to see what the current state of the board is, how moves turn out visually and where they are moving gems. In the following we elaborate on how we incorporated these visuals for *Gemji* into our own engine.

4.2.1 Gem Movement Along The Cursor

Gems move by holding down the mouse key and dragging the mouse across the screen which should be shown visually. To achieve this we have the mesh of the selected gem be the intersection of the mouse ray and the board while the left mouse key is being held down. It is worth noting that the origins of the gem meshes are at the very bottom center, so matching the mesh position with the mouse intersection caused no problems with height. When the player releases the mouse key, the gem mesh snaps to the selected coordinate on the board. A similar snap motion also occurs when the player moves a gem after clicking on it. Since the gem will have the origin of its mesh match the projected mouse position on the board, not clicking on the center of a gem will result in this snapping motion.

This would make sense if the player were to pick up a gem to then move it. But what we had in mind was for players to drag gems across the board. So in order to make this clear we added a 2D offset between the center of a gem mesh and the exact position the player clicked on given that this position belonged to a tile with this gem. With this addition gem meshes did not snap to the cursor anymore.

However we noticed that the meshes were way off the cursor position at far edges of the screen. So specifically for the meshes we defined approximating spheres around them. The 2D offset thus became a 3D offset that was calculated from the mesh center and the intersection of the incoming mouse ray and the approximating sphere. This new offset has gem meshes stick closer to the mouse position even at far edges of the screen.

4.2.2 Animations

Animations are especially important to us, because our game is about understanding the interactions of the Gems between each other and their temporal order. Each gem effect has a different animation:

Blue gems Flicker before they teleport back to their original position

Red gems Explosively push other gems away. Movement starting fast, getting slower

Yellow gems Pull other gems towards them. Movement starting slow, getting faster

Purple gems Switches position with another gem, by letting both gems rotate around their average position by 180° around the up-axis.

We also use a simple parenting system where each object can have a parent object. This is used to animate multiple objects together, for example for the purple gem animation. Both gems are first parented to an empty object that lies in the middle between to-be-switched objects. Then the orientation of the empty is animated to give the effect of both objects rotating around each other.

Additionally we use animations for simple camera movements at the start and end of levels as well as for fading tooltips in and out.

4.2.3 Scheduler

We chose not to apply animations in a simple frame-to-frame fashion. But rather give us higher level control over them. Therefore we put a lot of effort into the animation system which is based on our work in last semesters practical. We wrote an animation system which we call the **scheduler**, which can animate any variable of any size in main memory. We use this to animate simple floating point values but also orientations, and positions. To schedule an animation, you provide the scheduler with some information about it e.g when it should start, when it should be finished, what should be animated, what kind of interpolation should be used (linear, ease-in, ease-out, etc). Like this it is really easy to schedule animations to run at any given time in the future. When the player moves a Gem the whole turn outcome is computed in the same frame, and alongside of computing the chain reactions, their Gem movement animations are scheduled.

Additionally to executing animations, the scheduler can also be used to schedule **actions**. An action consists of a function pointer and optionally a set of parameters that the function pointer will be called with – essentially emulating closures[1]. We use actions when we know that after some time, *something* should happen. For example if the player performs the last move, that will set the board into a winning position, we first want the whole animation to play out and only then switch to the next level. So as the player performs the winning move and the board is simulated, and all the chain reaction animations are scheduled, we do a check if the board is in a winning position, and if it is, we schedule the action that

[1] C++ lambdas were not used, to be able to store actions packed next to each other in a flat array. This is not possible with C++ lambdas, which have different sizes, depending on their parameters. A higher level abstraction such as `std::function` would have to be used. We opted to not do this, as `std::function` can perform heap allocations on its own.

loads the next level. The playing of sound effects is also powered by the scheduler to be able to play the correct sound effect at the correct time.

4.2.4 Grid Creation

Originally grids were separate meshes that were built in Blender. As grid sizes vary from level to level, having to create a separate mesh for different dimensions is way too tedious. Therefore we have shifted grid creation from blender to our engine. The necessary vertex and index buffers are filled using a function that takes the grid dimensions as input. So when we create a new level, all we have to do for the grid mesh is to call the function with the dimensions of the playing field as inputs.

4.2.5 Tooltips & Textures

One aspect that we deem very important for puzzle games like *Gemji* is the explaining of the game mechanics without resorting purely to text which is boring and tedious to read. This is even more important as the next milestone is the playtesting stage which will also be the first time that people not on the development team will be playing our game. For this reason one of the goals of the alpha stage was to design and implement tooltips that explain the behavior of the different gems.

1. Design of the Tooltips

   For now We implemented one tooltip for each of the existing gems in the game. For the design of the tooltips we decided to show a before and after screenshot of the effects of the respective gem in an example game state. Additionally we added a title to each of the tooltips that is also color coded. Although the title does not deliver crucial information to the players it does help in terms of making it clear which gem is the focus of each tooltip and will help us for the questionnaire and talking to our testers in the playtesting stage.

2. Implementation of tooltips

   For the implementation of the tooltips we used the ImGUI library which we have integrated into our rendering pipeline. The ImGUI provides easy window creation for all UI elements that we want to use and will be also be helpful when we want to create an Options menu to control elements like the sound volume of the music/effects and rendering options. The tooltips are shown in the bottom right corner of the screen when the player hovers over a gem using the glfw mouse position callback. When the player stops hovering over any gem the tooltip will fade out after 3 seconds. Additionally the tooltip will stay active when the player holds the gem activly with the mouse before making a new move. The animations for the tooltip fading are implemented by our scheduler animation system.

3. Finish tiles

   The goal for a round of *Gemji* is to move gems to their respective colored finish tile. Up until this milestone we did not have a representation of the finish tiles and even the command-line version did not mark the finish on the ASCII boards. This crucial aspect was very important to finish otherwise it would be very difficult to explain to the play testers what the actual goal of *Gemji* is. Additionally the design of the finish tile textures influences the perceived game setting for the players. For now the finish textures were designed as magical circles but could be subject to change in the future. Furthermore the game objects will get automatically created and place to the correct locations depending on the informtion in the "$\text{playing}_{\text{field}}$" struct when a level is loaded.

4.3 Sound Effects & Music

Puzzle Games usually do not have a good way to immerse the player in a narrative. This is especially true for *Gemji* as it is a very abstract puzzle game. Immersing the player in the game world has to be achieved by other elements such as SFX, music and particle effects.

4.3.1 Sound effects

Finding fitting sound effects for *Gemji* was definitly a challenge not only because of the search process but also due to the relation to the "game world". Just as with the finish tiles we decided to go into the direction of mystical/magical SFX that relate to the effect of the respective gem. In the following section we describe the SFX as they relate to the gem effects:

Red gems Pushing - Magical shockwave

Yellow gems Pulling - Slow starting but abrupt ending wooshing

Blue gems Back teleportation - Channelled teleport sound

Purple gems Swapping gem - Magical rotation

4.3.2 Music

Just as the SFX the music has a major role in the players immersion of the world. For now we wrote a simple asian-style mandoline melody for *Gemji* as we feel like we could go into asian-influenced theme. This inspiration also came from the game board of the physical prototype where we used a Go-board which we now also use as representation of our board in game.

4.4 First Campaign Levels

From last semesters project we gathered additional experience how we want to setup the level structure. The level structure is the biggest influence on the learnability of our game. The levels have to be structured in such a way that they not only introduce the basic game mechancis in a sensible matter that allow for good undestandability and a smooth learning curve but also slowly teach what we call emergent effects. Emergent effects are the resulting game patterns that depend on the interaction of the different game mechanics working in combination. This mainly includes the different gem effects interacting with each other but for example also in which order the effects are triggered. The level structure and the learning curve is also something that we want to closely analyze for the playtest.

5 Playtesting

5.1 Playtesting Sessions

As the core of this milestone there are the playtests. After having found participants we sent them a .zip file containing the game. In the following sections we present the contents of the demo, the procedure of the playtesting sessions as well as the questions in our questionnaire.

5.1.1 Playtesting version of *Gemji*

The demo version of *Gemji* that we used for the playtest included 19 levels with increasing difficulty. This number of levels might seem too high but in reality most levels were quite short due to the introductionary aspect they had to fulfill. The demo version includes all 7 gem types and effects which we first had to teach the players one after the other. Each of the gem types has its own set of levels for this purpose, starting with a very simple intro level that displays the mechanic and then additional levels that try to show the player what they have to watch out for when using that particular gem. The latter levels in the playtest were a little bit more difficult as they included several different gem types and were more focused on the emergent effects of gem combinations. Most of the levels were handmade but two were automatically generated using the bruce-force algorithm.

5.1.2 Procedure

After welcoming our testers we had them casually play the game. We left it up to them whether they to wanted to finish all levels or opt to drop out in the middle of the session. Since we included a few levels and most were rather short, all participants were able to successfully finish the demo. We further did not give them hints how the mechanics work and instead had them figure out the mechanics by playing the game and the in-game tooltips.

5.1.3 Questions

After finishing the demo the testers were asked to fill out a Google Forms questionnaire to judge the general sentiment of the gamne among other aspects. Compared to last semesters project we decided to use a fixed questionnaire instead of a free form interview which made the data collection process much easier. We added a free comment and suggestion box at the of the questionnaire to keep similar opportunities that an interview offers. Some testers also offered to play the demo on stream which allowed us to gain additional insight on how they approached the levels.

1. Demographic questions

   Starting of the questionnaire are two demographic questions regarding the age of the participant and their average weekly time spent playing video games.

2. Gem effect questions

   Going into the playtest we knew that we not only wanted capture a general sentiment about our game but also test whether the level structure and design combined with the tooltips would be enough to convey the game mechancis in a reasonable manner. For this reason the first part of the questionnaire was only focused on the participants understanding of the gem effects. For each of the gems we had a mutliple choice question on the gem behavior and how confident the tester was in his judgement of that particular gem.

3. Impressions questions

   For the first impression of our game we mainly asked about the Sound effects, graphics and animations. Additionally we asked about the difficulty and whether our game is frustrating to play. At the end we also asked broader questions about the game itself. If it was fun, or fulfilling or maybe frustrating.

5.2 Results

After playing through the demo the testers got filled out the questionnaire. In the following we present the results.

5.2.1 Demographic questions

While with few outliers the age range of our testers was rather compact, the differences in how many hours the testers play video games vary considerably.

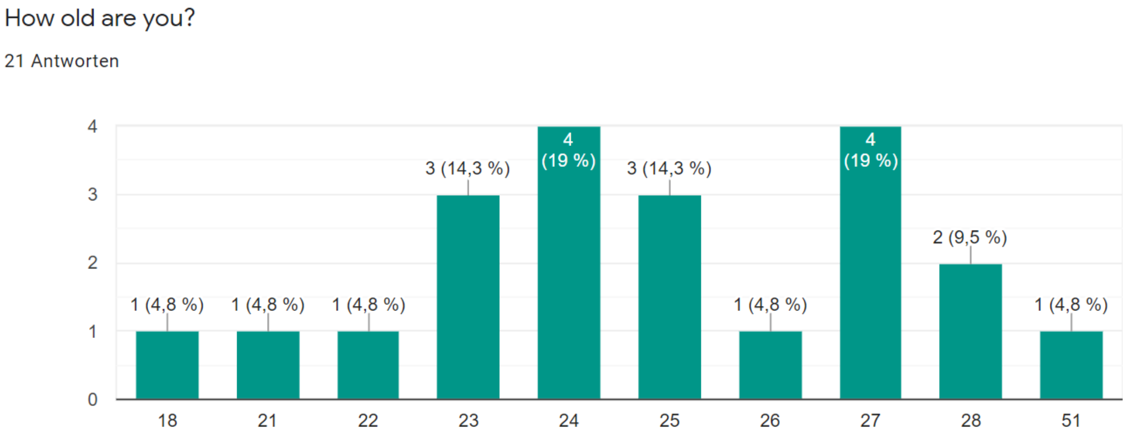


Figure 7: Most testers were in the 20 to 30 age range.

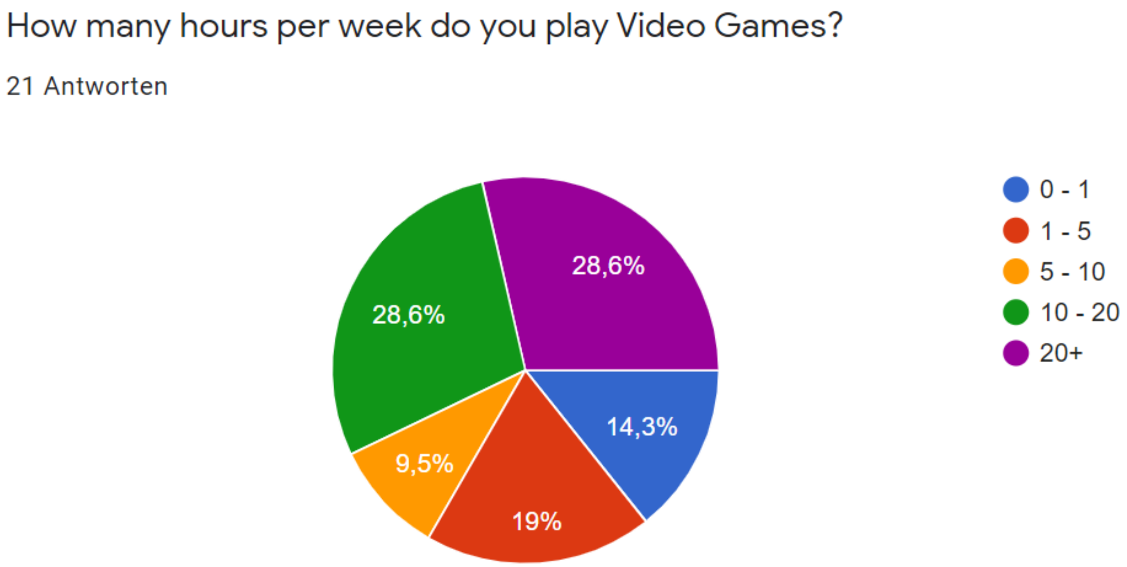


Figure 8: The majority of testers is very familiar with video games.

5.2.2 Gem Properties

The questionaire results for the gems are really diverse. Some gems were understood really well. Testers could choose the correct effects belonging to a certain gem and additionally reported that they are confident in their answer. On the other hand some gem effects seem to be harder to grasp. The majority of the testers could always identify the correct effects of a given gem, however for some gems, the number of wrongly picked effects was higher. The property of a gem to be able to be moved by the player could also be identified by the testers most of the time. However about half of the testers had trouble identifying most gems' ability to activate the effect of other gems. The only exceptions are the grey and black gems that do not activate other gems. Notable exceptions to these trends are also mentionned in the following sections.

1. Red gems

   The pushing effect of the red gems was recognized well by the testers. We attribute this to a few cicumstances. Mainly the red gems were introduced first to the player. The first few levels only contained the red gems, so the player had a lot of time to familiarize themselves with their effect. Furthermore the pushing effect is quite apparent and easy to remember.

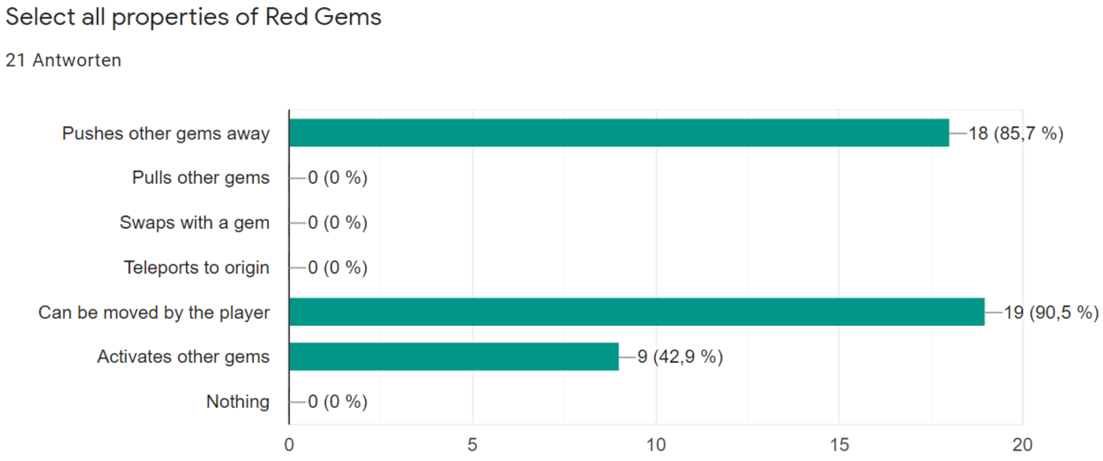


Figure 9: The effect of the red gems could be identified by almost all testers.

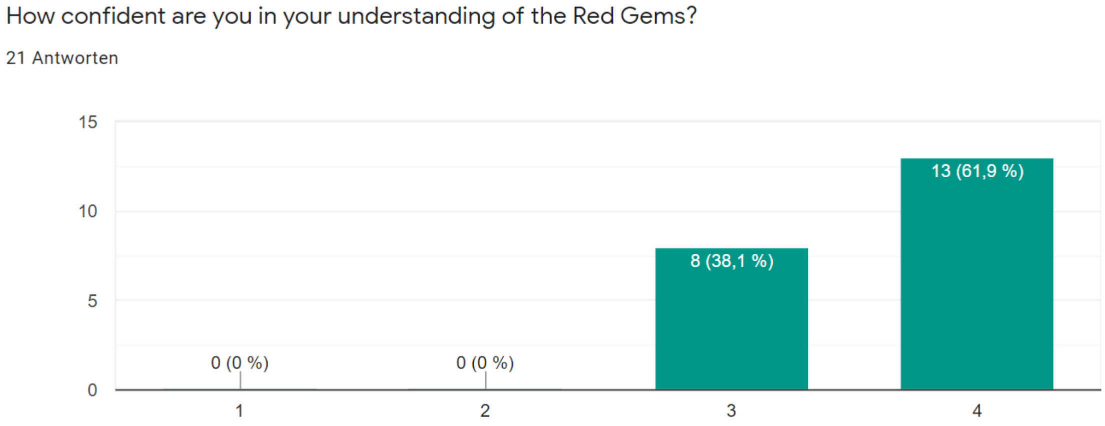


Figure 10: Most testers felt confident in their understanding of the red gems.

2. Yellow gems

Yellow gems were introduced after the red gems. Their pulling effect could be identified by most of the testers, however some confused it with the pushing effect. The testers had an even higher confidence in their pick than for the red gems. Overall the confidence of the understanding of the yellow gem was the highest considering the results of the questions on the gems.

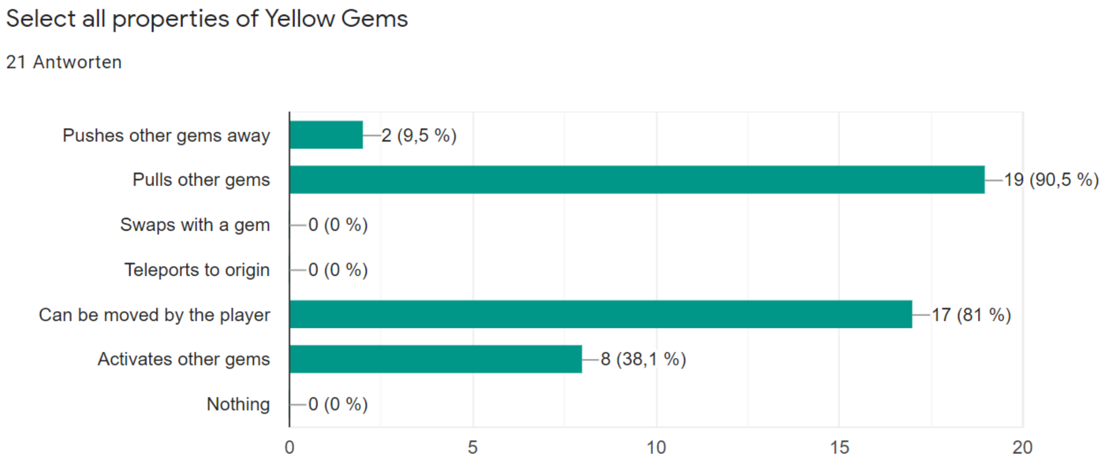


Figure 11: Some people wrongfully attributed the pushing effect to the yellow gems but the majority could identify the pulling correctly.

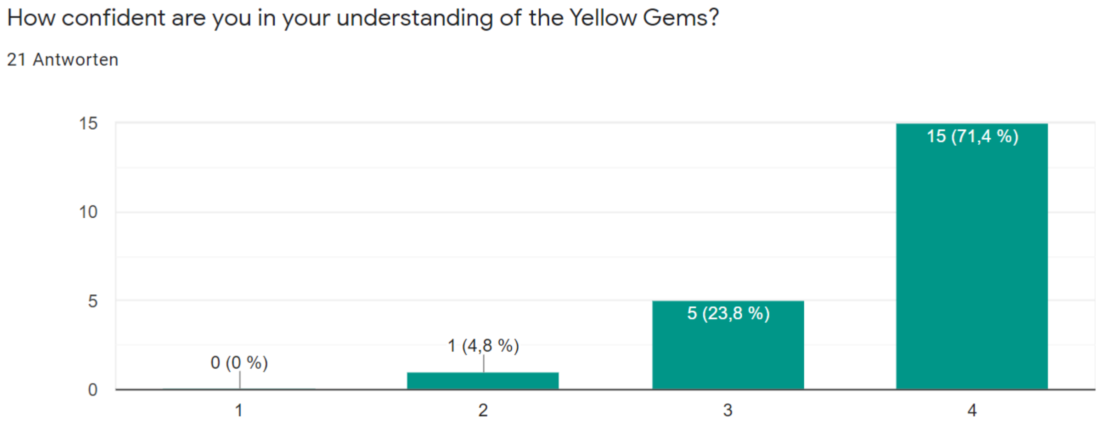


Figure 12: The confidence over the understanding of the yellow gem was higher than for all other gems.

3. Blue gems

   Blue gems teleport back to their starting position of the turn, as long as it is still free at the time of the action, which adds a bit of complexity to the effect. The players could identify the teleportation. However the effects of other gems were also wrongfully identified to the blue gems. Interestingly for blue gems most testers identified their ability to activate other gems. This is probably because its effect was used many times in the testing levels to activate other effects.

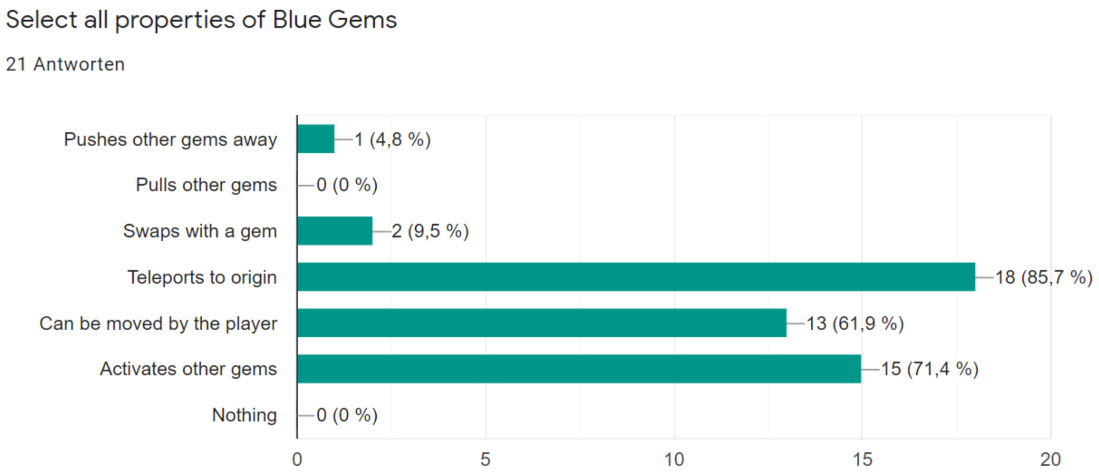


Figure 13: While the teleportatiopn could be identified, other effects were also attributed to the blue gems.

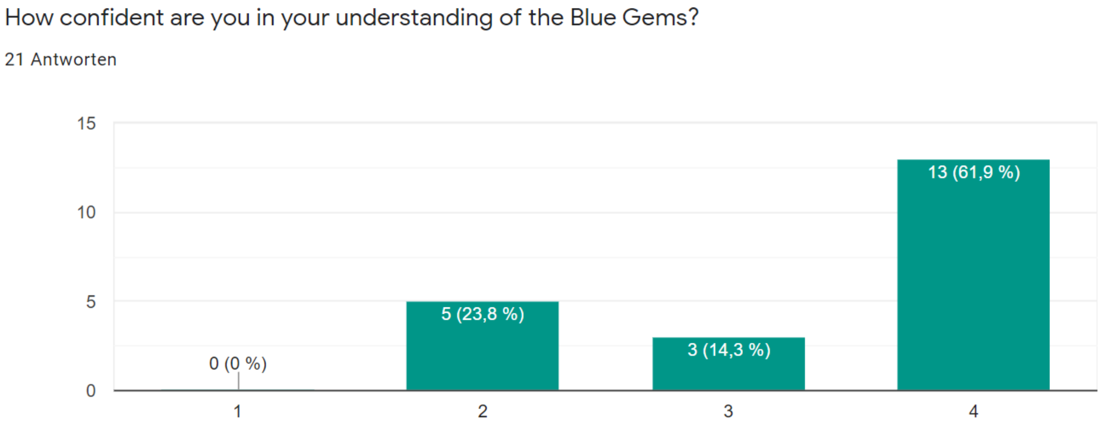


Figure 14: People felt more unsure about the blue effects compared to the previously introduced ones.

4. Purple gems

   Purple gems swap positions with the gem that activated it or – if it was moved by the player – try to find a neighboring gem to swap positions with them. The search happens in clockwise order starting from the neighboring field just north of the purple gem. This makes the purple effect harder to understand fully but there are also only few occurences of this rule in the testing levels. Overall the effect could be correctly identified. Similar with the results of the blue gem effect, effects purple gems were similarly confused with blue gems. We attribute the confusion between the blue and purple gem effects to the fact that they together have an emergent effect that is used throughout some levels.

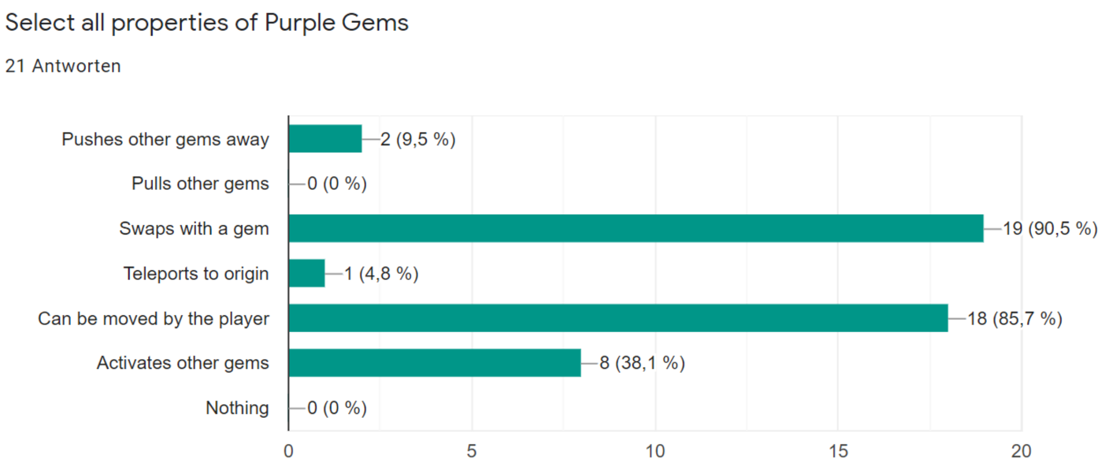


Figure 15: The swapping effect of the purple gems could mostly be identified correctly.

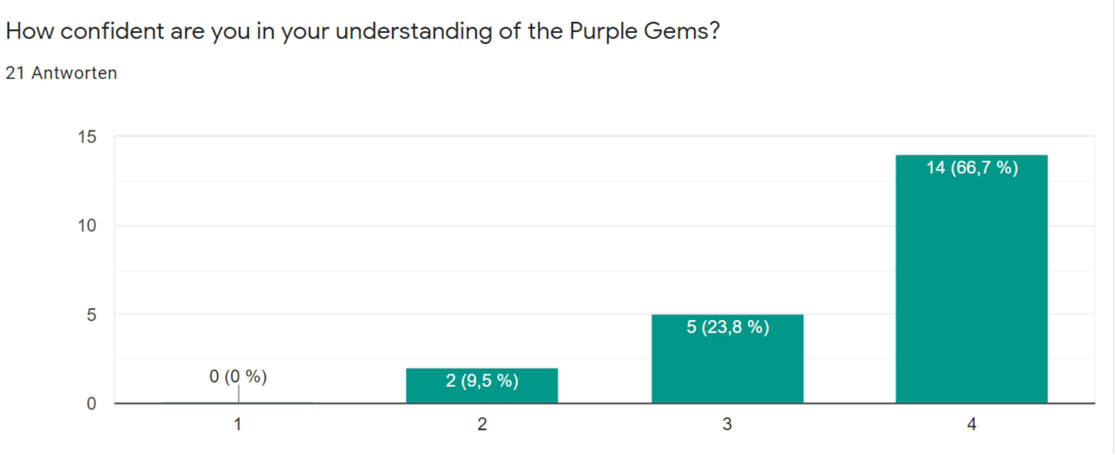


Figure 16: Most testers also feel confident in their understanding of the purple gems.

5. Grey gems

   Grey gems do not have any effect and cannot be moved by the player. They can only be moved by effects of other gems. The vast majority understood this correctly however they did not feel confident to have really understood the properties correctly.

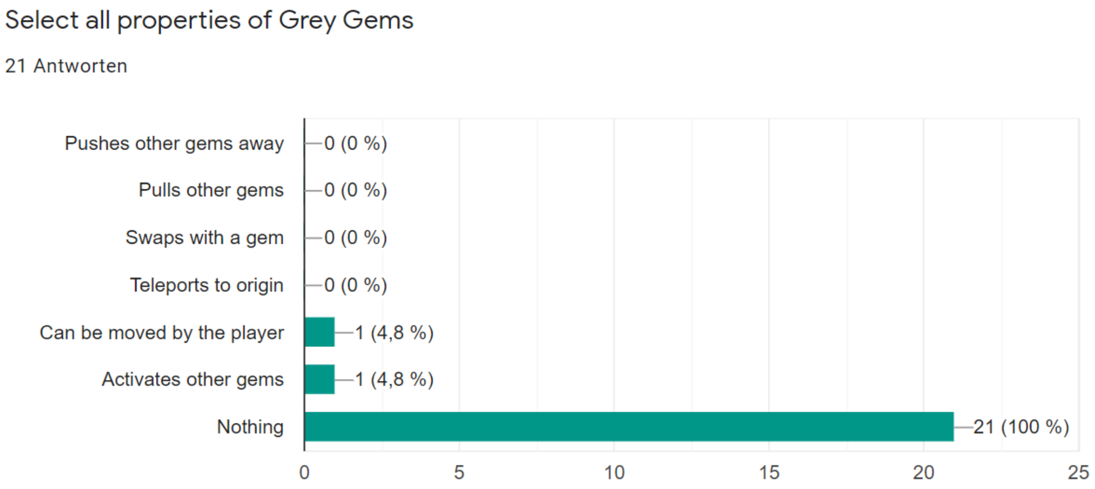


Figure 17: The testers did not have any difficulty to understand the properties of the grey gems.

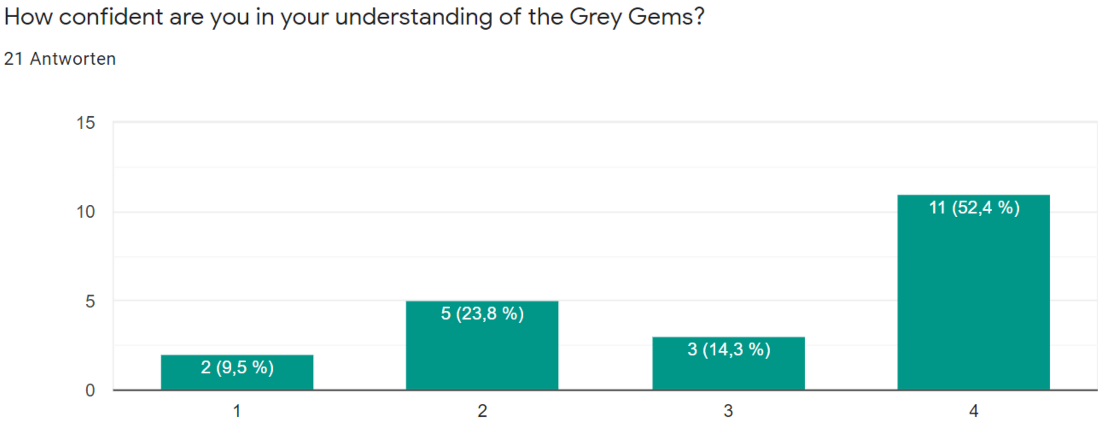


Figure 18: Even though the grey gems do not have any effect, the testers felt unsure about them.

6. White gems

   White gems behave identically to grey gems with the only exception, that when activated it will also activate the effect of all its neighbors. Interestingly the effects of other gems were also identified with this gem, maybe because the white gem activates the other gems. Regardless, the majority correctly identified the absence of any effect but only half realized that the white gem also activates other gems. Notably the majority of testers was unsure about the properties of the white gems.

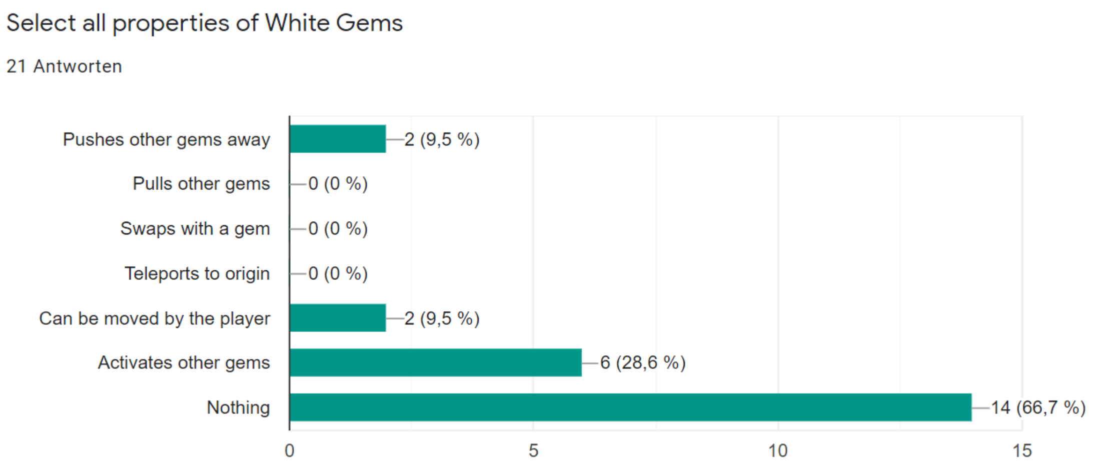


Figure 19: Only around 29% of the testers realized that while gems can activate other gems.

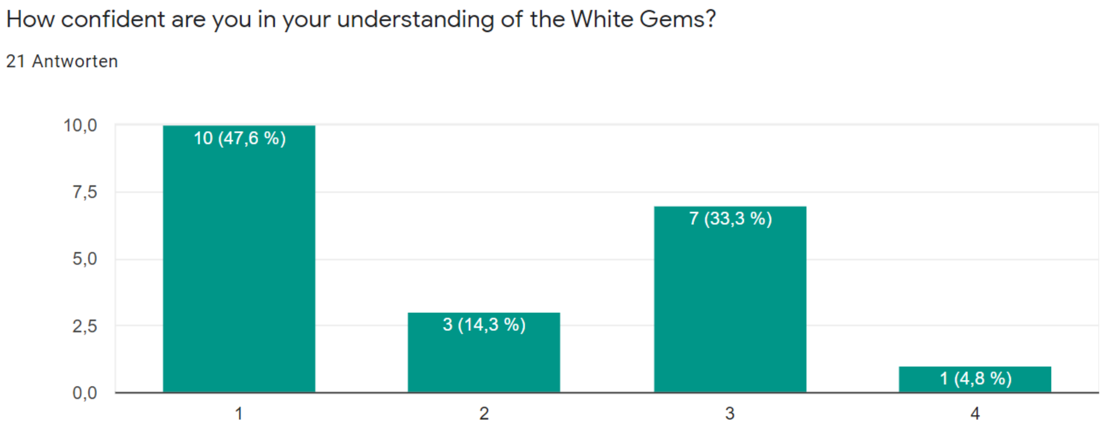


Figure 20: Most people felt really inconfident about their knowledge of the white gems.

7. Black gems

   Black gems do not have any effect and can neither be moved by the player nor by other gem effects. Effectively they are obstacles on the playing field. We attribute the simple properties to the clear result.

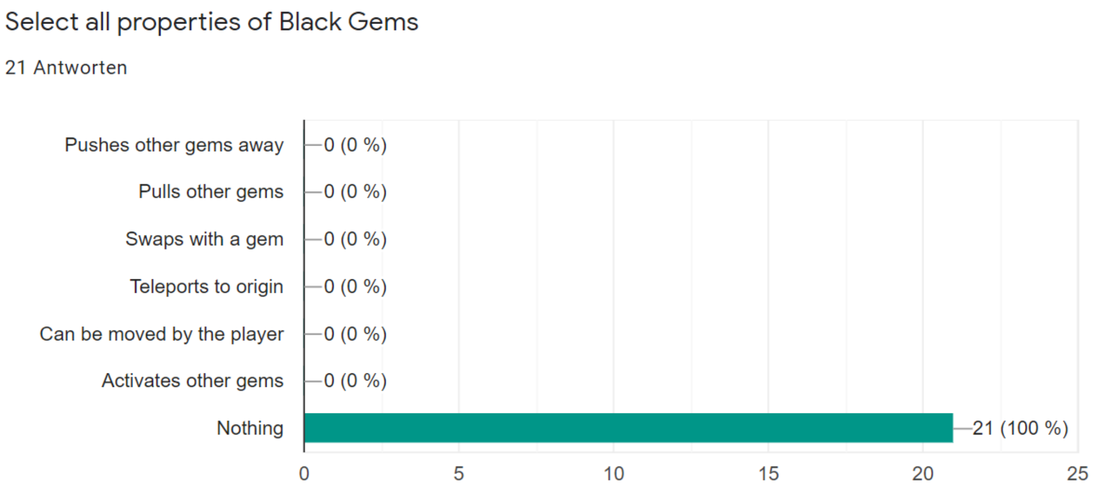


Figure 21: All testers could correctly identify the properties of black gems.

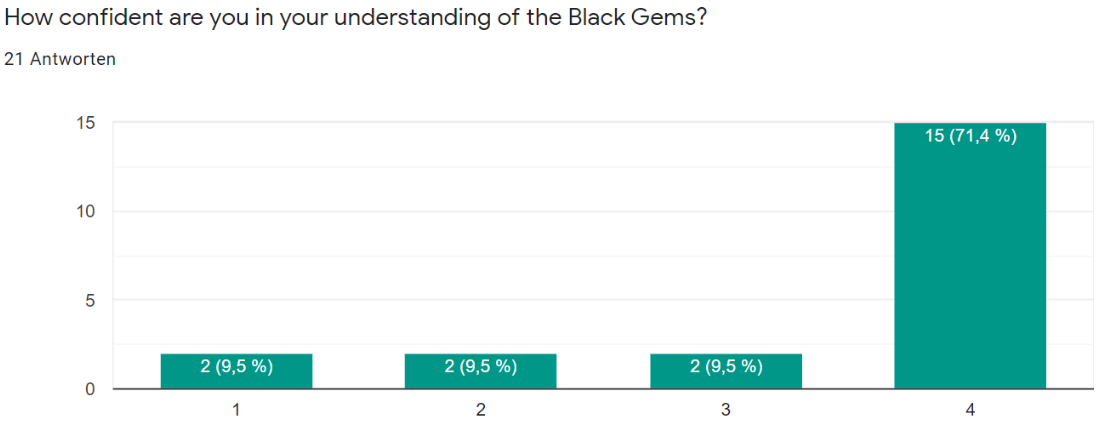
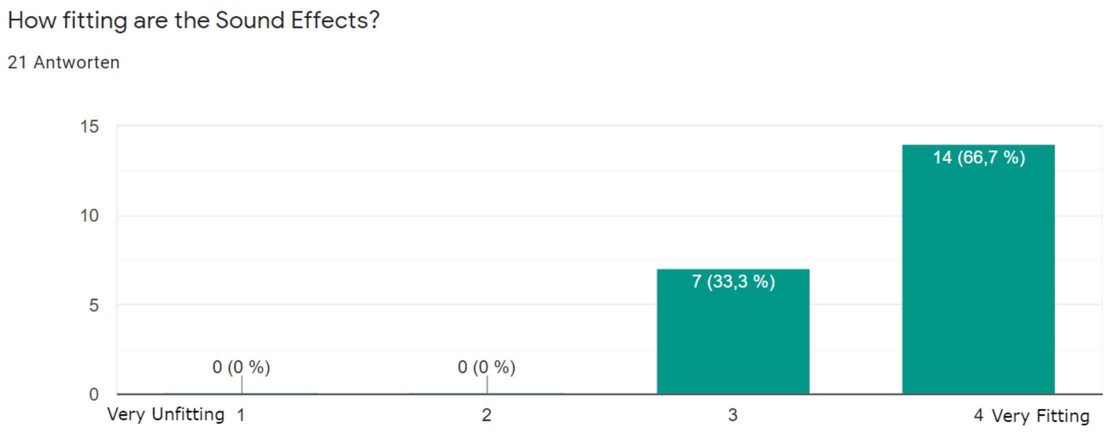


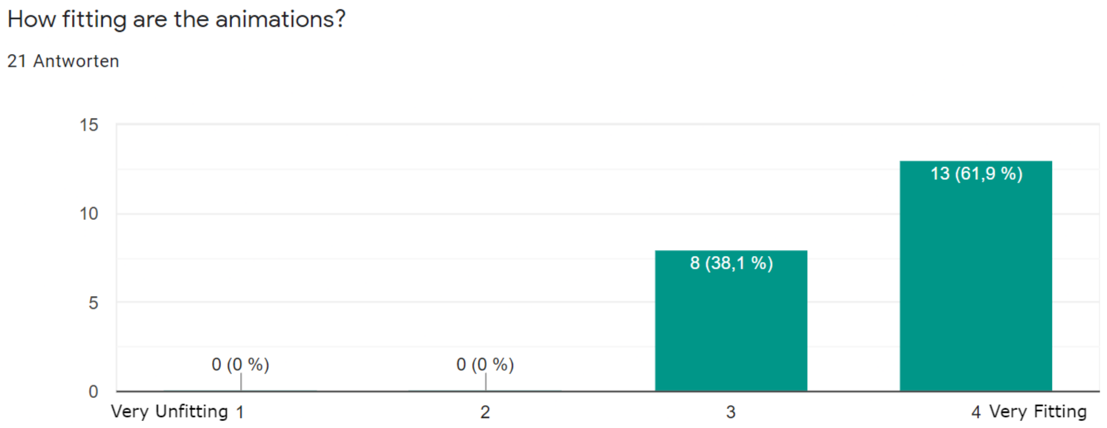
Figure 22: Most people felt really confident in their understanding while a small percentage was not too sure.

5.2.3 Impressions

The animations and sound effects were received rather well. As 'small' as they are, they did not get in the way of the testers' thinking process while playing.

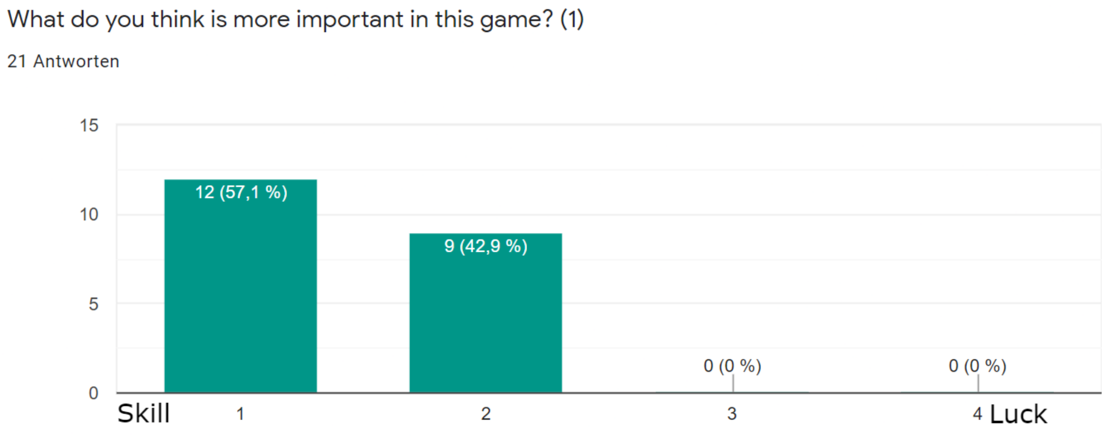

How fitting are the Sound Effects?

21 Antworten

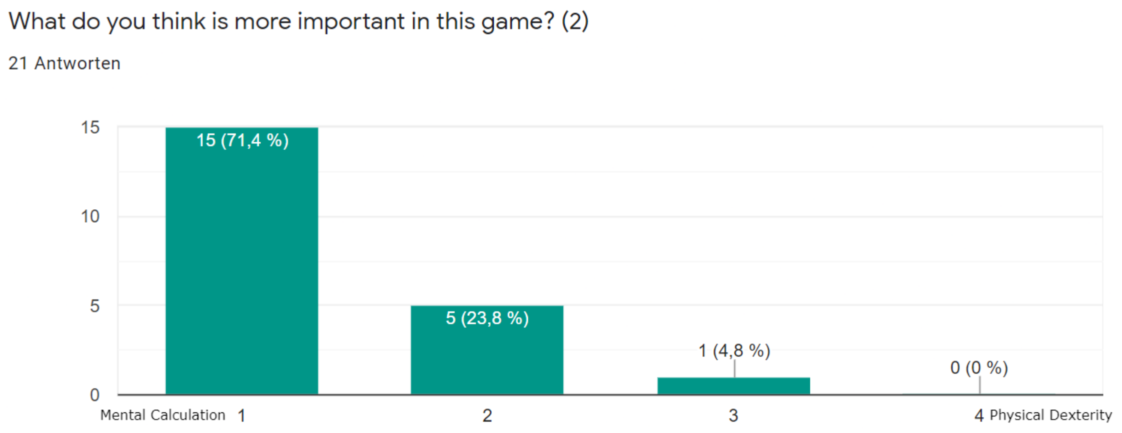

How fitting are the animations?

21 Antworten

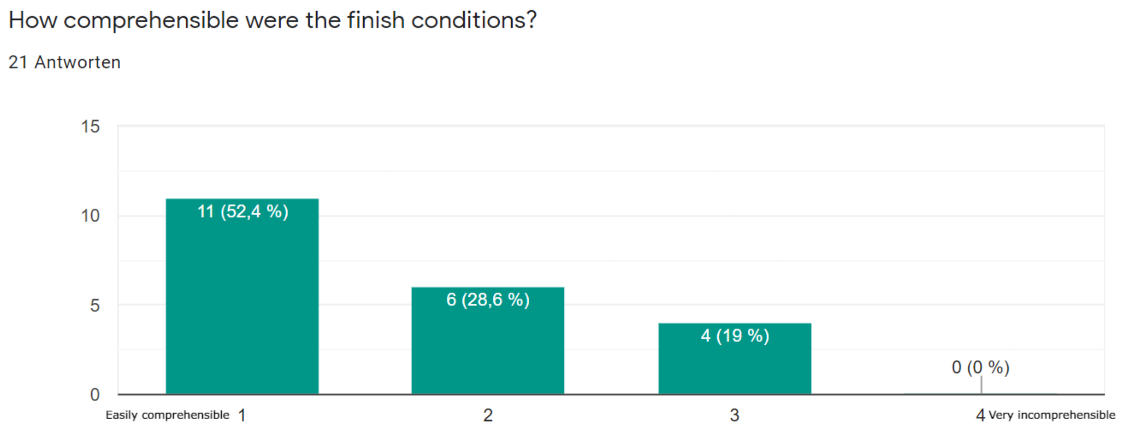
In regards to the play matrix the results reflected how we envisioned *Gemji* to be: A skill-based puzzle game that requires mental energy to beat.

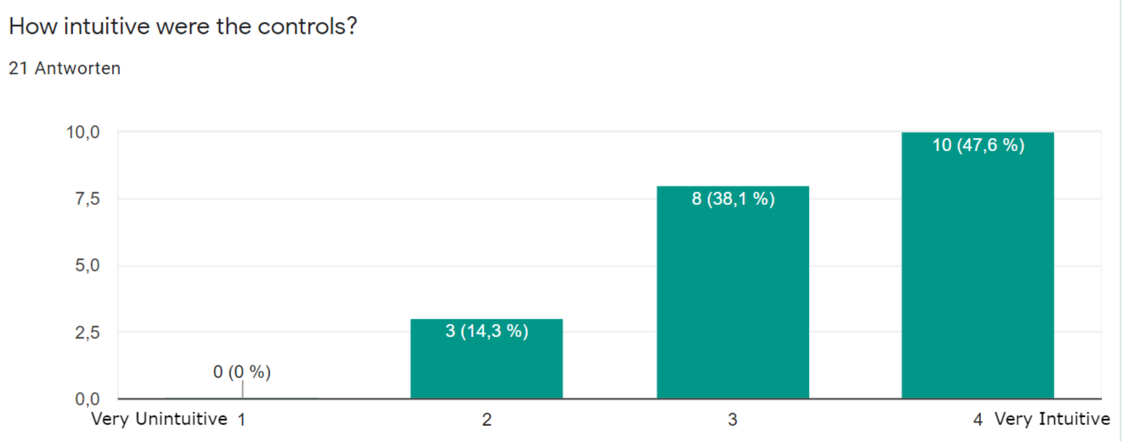

What do you think is more important in this game? (1)

21 Antworten

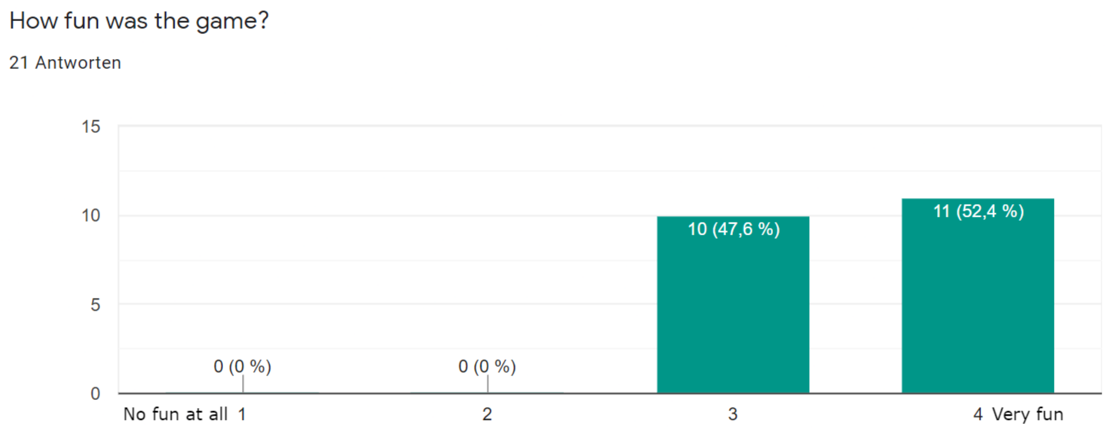


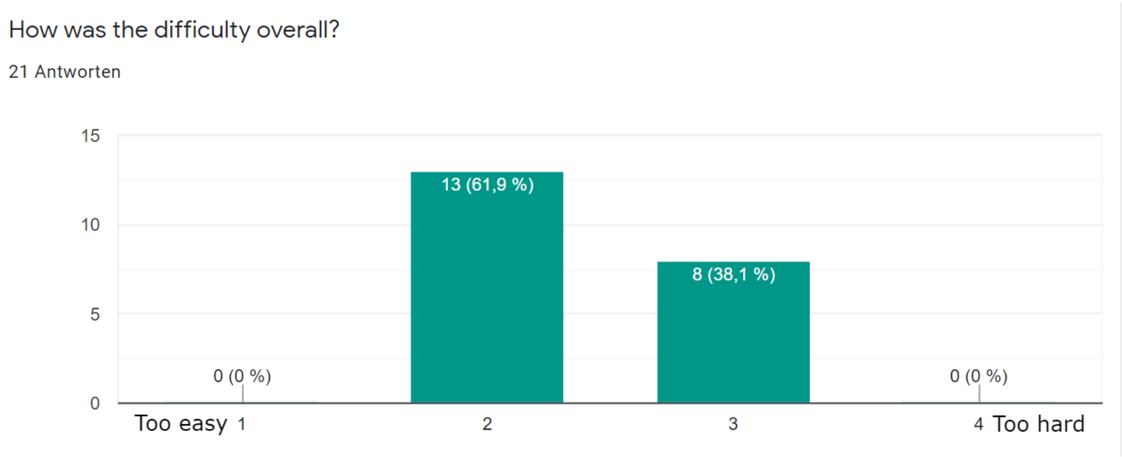
In terms of accessibility the game seemed to do well as both finish conditions and controls were received as fairly comprehensible and intuitive.



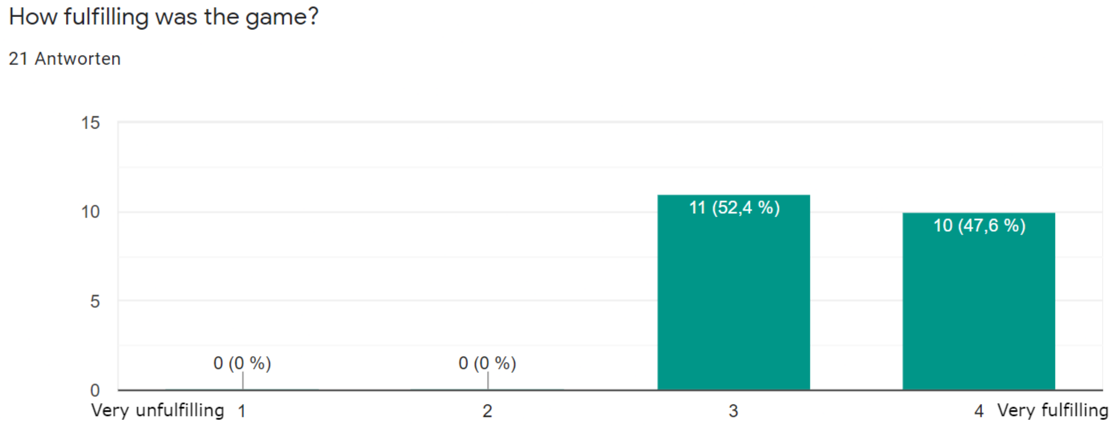


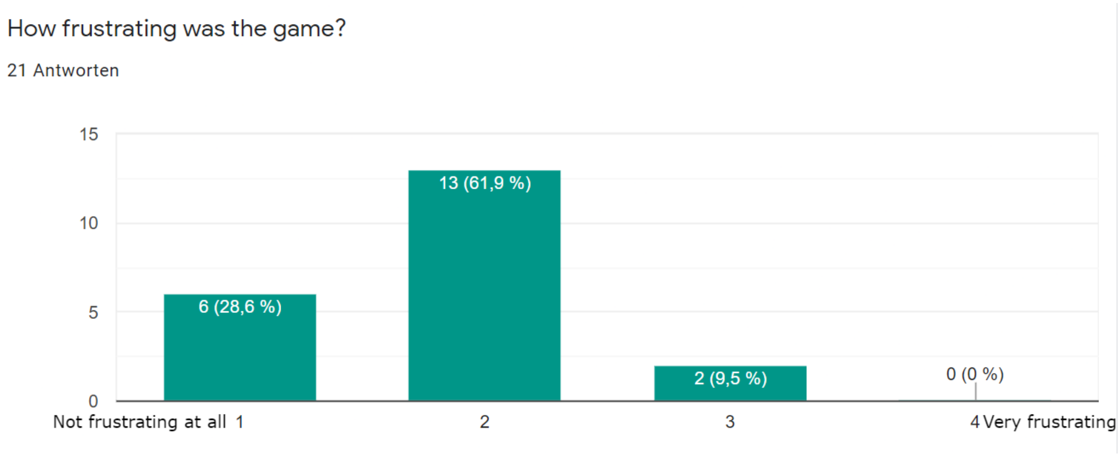
Gemji further came across as fairly fun by everyone. A reason for that may be the difficulty that was regarded as not too easy or too hard. This indicates that we are on the right path in terms of level design.



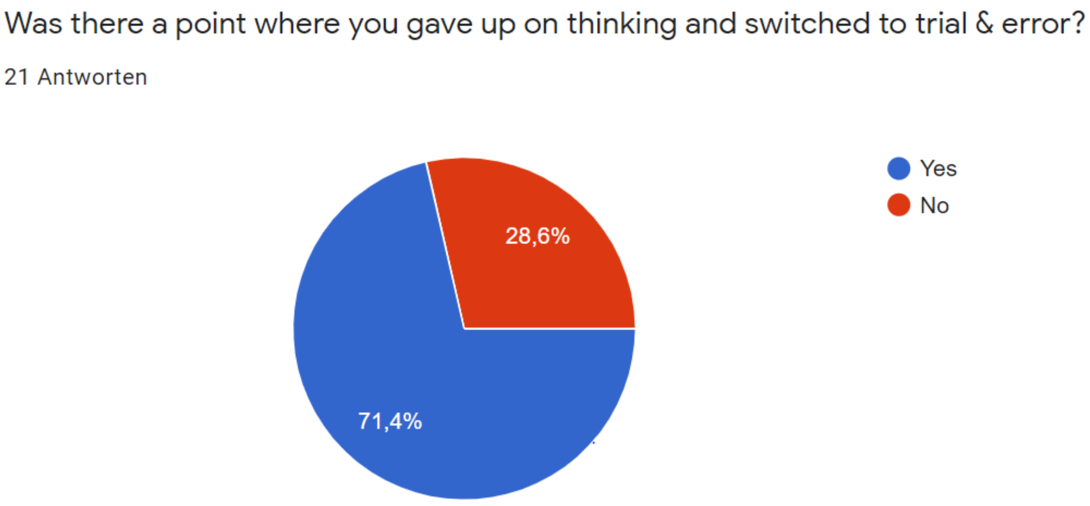


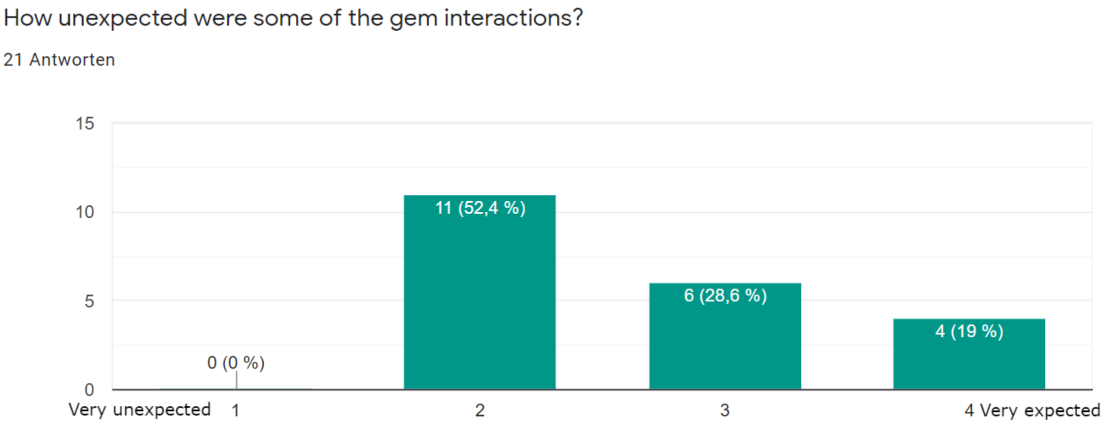
Fulfillment was something that was present across all testers. Especially in more convoluted levels this fulfillment after finishing the level seems to go hand in hand with a certain degree of frustration, albeit to a lesser extend.



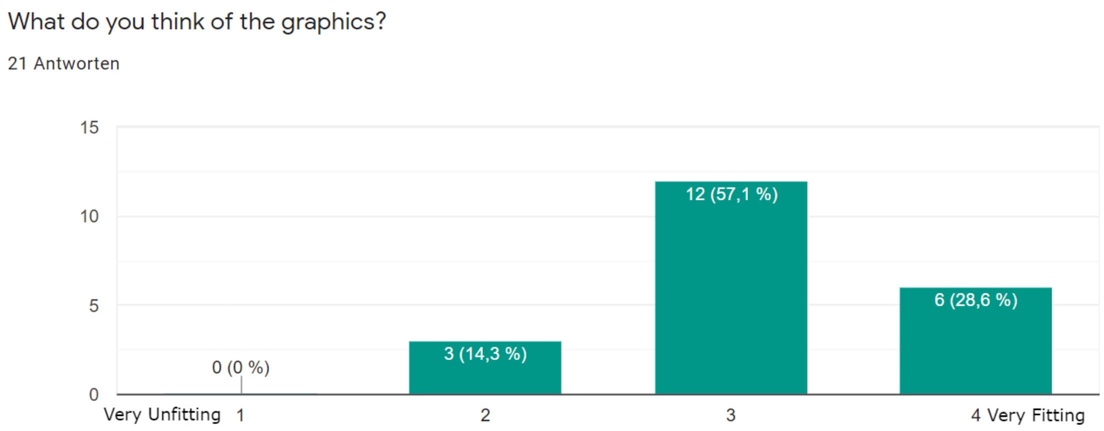


The majority of participants also resorted to trial and error at some point. Since a considerable amount of participants reported unexpected gem interactions -although they are completely deterministic-, our integration of the theme "chaos and order" seems to be a success.





Graphics were regarded as fitting. This is a field we definitely need to work on. We therefore did not expect the graphics to be a highlight of our game.



Observing testers a trend that further became apparent was that there was nothing conveying to players that gems can only be moved one tile at a time.
Finally addressing comments and suggestions the ones that stood out referred to the instructions of the mechanics that we provide in form of images. Testers suggested to be more specific with explanation especially in regards to gem interactions. Since we want want to keep instructions to an absolute minimum, we will have to strike a balance.

5.3 Conclusion

In this section we will discuss the main conclusions we have taken away from both the questionnaire and watching the testers play the demo. Furthermore we will suggest solutions to problems we encountered during the playtest.

5.3.1 Intro to basic game mechanics

One thing that was very apparent from the start was that we have to do a better job in explaining the base mechanics of the game, namely: How gems move along the grid and that they can only move one space per turn. For this reason we want to implement additional one line tooltips in the first few levels that explain how the gems can be moved along the grid. Additionally when watching some of our testers play we realized that they often tried to move gems over other gems or into an already occupied space. To prevent this we want to use point lights that shimmer in the possible spaces a picked up gem can be moved to. We hope that this and the textures make the movement clearer for the players, especially in the first few levels.

5.3.2 Clearer Tooltips

Some playtesters critiqued that a few of the tooltips explaining the gem effects were not clearly formulated and could even be confusing. This was especially true when the tooltips included gems that were not already introduced. The original plan was to implement a Picture-in-picture clip that displays the gem effect on an example board. Although this is important we want to develop other aspects of our engine first which means that we might not have enough time to implement the pip-tooltips. Depending on the remaining time we might update the existing tooltips or implement the picture-in-picture functionality.

5.3.3 Better level structure

We were overall content with the level structure of the demo and it was also not negatively mentioned in the quesionnaire but when watching the testers play through the demo we realized that the level structure could use some small improvements and that some levels should probably be placed a little bit further back. We came to this conclusion as some levels were clearly more difficult than we first anticipated. Included in this small restructure will be additional intro levels for all gem types and emergent effects.

5.3.4 Gem activation and chain reactions

The aspect that was critiqued the most in our playtest was the clarity of the effect outcomes. This problem was mentioned both in free the comment section and deduced from the score judging the unexpectedness of the gem interactions. We assume that the main reason for this is unclarity is that the player does not know the order the gems activate in and thus can not calculate the effect combinations that occur of several effects chaining together. We want to improve on this problem by implementing point lights that are created in gem once it is triggered and the rest of the animations are playing out. We hope that this will be enough of an indication to the players that the order of effects matters and that they are working according to a set of rules. Realizing that there are underlying rules that govern all of the gems behavior is something we assume to be a fun part of puzzles games but can only really be observed over a longer playing session.

5.3.5 Improved VFX

The most lacking aspect of our game right now are the graphics and the overall setting. This was not only mentioned by some testers in the questionnaire but is also something that we were aware of before going into the playtest. One comment mentioned that they were expecting much more exciting effects of the gems moving which we did not implement yet because we are still working on the particle system. We also think that we can improve the current "world setting" of our game as it is very dry for now with only the grid showing. We want to add atleast one fully fleshed out level with details around the game board that give it the necessary flair it deserves. Point lights and the particle system will be a great starting point to achieve this goal.

6 Final Release

6.1 Final Version

Since the alpha release of *Gemji* several additions have been made. Starting with processing feedback from the playtests, we further added features we felt would enhance the experience of the game. In the following we present these features.

6.1.1 Movement

Previously the player would have to click on a gem, then drag it to the new position. A considerable amount of playtesters found this input method cumbersome. Sometimes the gem would not move to the desired spot because the mouse position was slightly outside of the boundaries of the spot. So we made the following change: When holding a gem now, the game calculates and shows the nearest viable position the selected gem could go to (including the original position) in form of a ghost gem. Should the player now release the mouse key, the gem will snap to that shown location. We believe this input method is an upgrade to the old one as players do not have to drag the gem from spot to spot completely accurately anymore. The exact mouse position on release does not matter as much anymore, the right direction suffices now. This should help players concentrate more on the puzzles without the controls getting in the way anymore.

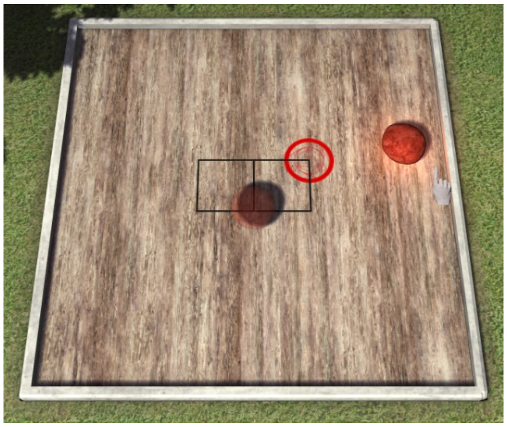


Figure 23: Here a gem is being held here. Should the player now release the mouse button, the gem will snap to the location shown by the ghost gem

6.1.2 Improved Visuals

The most critized aspect from our playtesters as well as from ourselves after the alpha stage were the visuals of the game. This is why we heavily focused on improving the visual fidelity of our game for the final release in several different ways.

1. Particle Effects

   One of the first things that we wanted to implement for improved visuals were particle effects. Most of the gems in our game have magical effects that interact with each other but the way they interact was not automatically clear just from their animations. For this reason all gem effects also trigger a type dependent particle effect when they trigger their effect. We think that the implemented particle effects are a great way to visualize the effects of the gems and will help our players to understand the game more easily. The particles additionally add more flavor to our world and make the game more fun to play in general. We also added a visual effect when the player is clicking the cursor similar to effects in games like Hearthstone. We think that these kind of effects add a feedback to the player that triggers the same sensory areas in the brain as haptic feedback does and are a great addition to turn-based games like board, card and puzzle games. The particles in our game are implemented as game objects with textures that are rendered on a CPU side quad mesh (just like all other game objects). A wrapper function fills a particle info struct which is then used as the argument to a spawner function which creates the particle game object, animates it and frees said particle from the information stored within the info. The VFX that are used in our game right now are hand crafted and animated but we also implemented some functions for general VFX.

2. Point Lights

   To add more atmosphere to the game we implemented point lights and put a colored point light in every colored gem so that it would shine in its own color. While the gems are animated, the lights will follow the gems' position. Point lights consist of a position, color, radius and intensity where

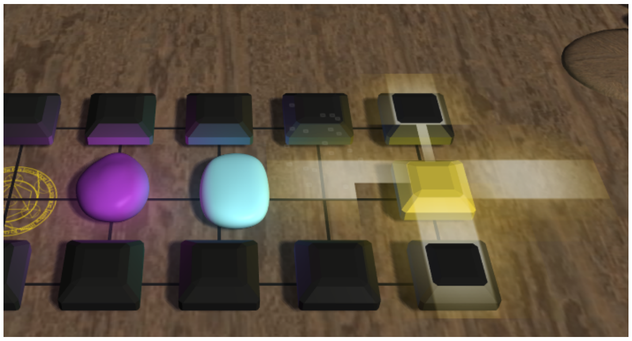


Figure 24: Yellow particle effect

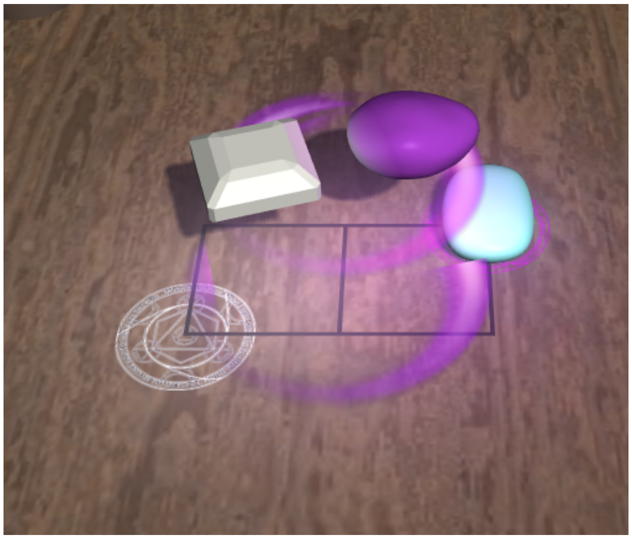


Figure 25: Purple particle effect

the light falls of non-linearly. We experimented with point lights that "emit" negative light, so basically steal light around them for an effect for the black gems, but ultimately decided against it because we wanted to keep the graphics style bright and happy. You can see the effect of negative lighting in Figure 27. Additionally we also animated the light intensity depending on if a gem is triggered or not. However we also noticed that this would make the game harder to understand for beginners and also decided against using it in the final game.

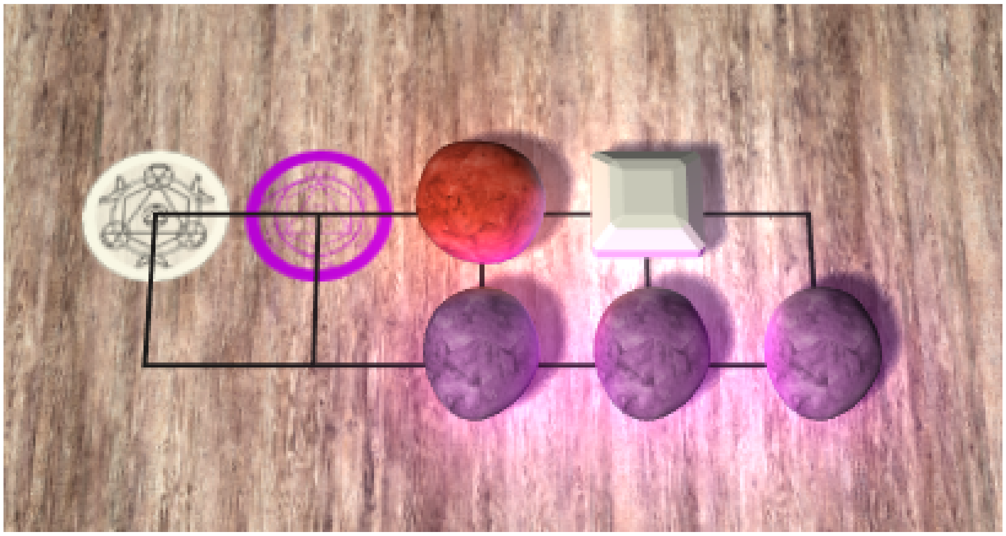


Figure 26: Every gem emits light in its own color

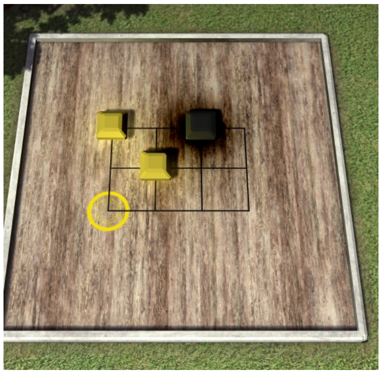


Figure 27: An experimental effect of black gems that did not make it into the final game

3. Animated cursors & new textures

   As we were looking for a general game setting for our game world and scenery, we also decided to upgrade the cursors to better reflect that setting. The new cursors are in form of a marble hand and an hourglass. The new cursors also change depending on the object they are hovering over. There are six different cursors that indicate the following states:

   - Normal: The default state
   - Can Grab: Additional green ring around the cursor when the hovered gem can be grabbed
   - Grabbing: A cursor showing that the player is holding a gem

- Cannot Grab: The normal cursor wih a red cross-out showing a non-movable gem
- Placement Forbidden: The same cross-out without the hand when a placement of a gem would be forbidden
- Hourglass: A hourglass showing during loading and when effect animations are playing

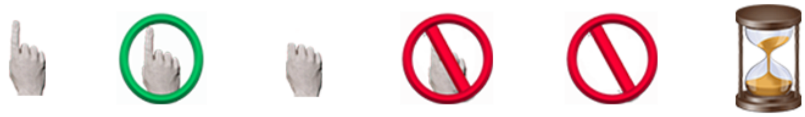
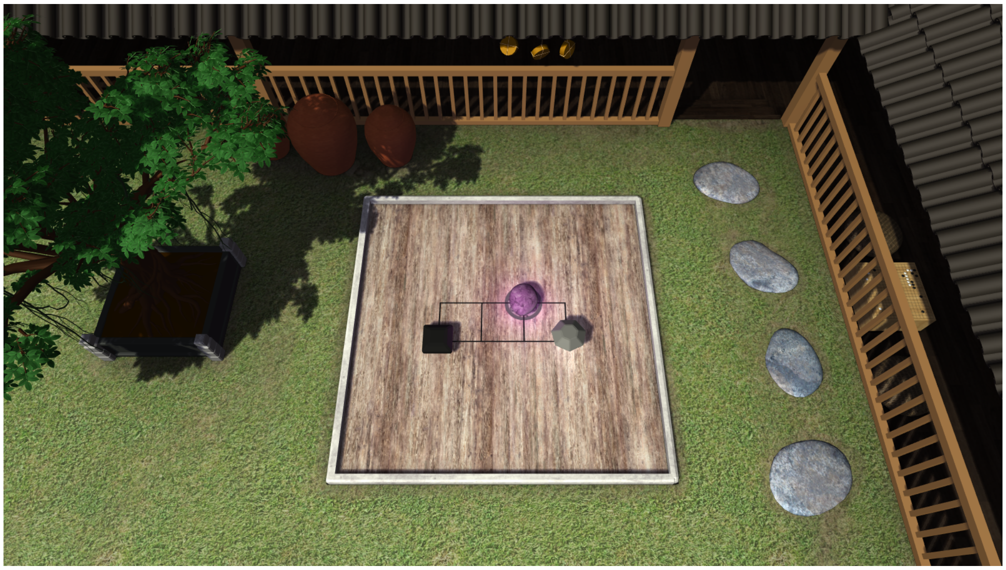


Figure 28: Gemji cursors

4. Scenery

   Since the game did not have a setting yet, we decided to put the playingfield in a 3d environment. We went for the asthetics of a man-sized chess board in the couryard of a building. And since our game drew some inspiration from the ancient chinese board game of Go, we decided on a east asian mood for the surroundings. The models for the scenery was done by us ourselves, except the tree and the textures, which we obtained from the internet. We used Blender for the 3d modelling and texture painting to draw some of the textures on some meshes.



5. Transparency

   To be able to render the ghost gems as an indicator where a gem will be moved to we also had to implement transparency. If a fully opaque gem would be rendered there it would be confusing for the player to see which gem is the preview gem and which gem is the "real" gem. The transparency was implemented using standard alpha blending. We thus also made sure to render the gems in the correct order so that the trancparency was rendered correctly.

6.1.3 Menus

The bring more structure into the game we added menus. When starting up the game, the player is greeted by a main menu (after the initial loading screen). From there they can either go to the campaign menu to play a level we designed, play a randomly generated level in the endless mode or quit the game.

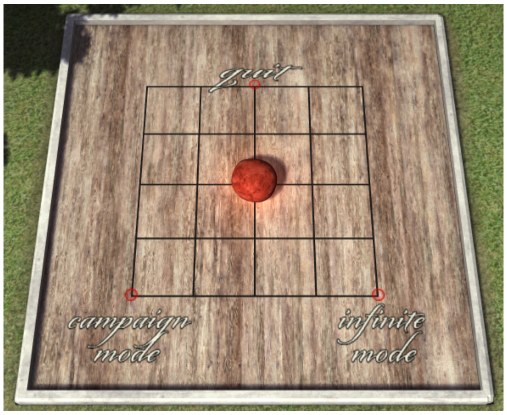


Figure 29: This menu opens at the start of the game and after the player presses escape. The campaign, the endless mode and the quit option are accessible from here.

In the campaign menu players choose between the levels they wish to play. It is important to note that players have to unlock later levels by beating previous ones. From the beginning the very first level is available. To achieve this availability feature we implemented a simple save system. When finishing a level, the game now saves the latest beaten level index in a .txt file. This way we can always convey the players' progress to them. Furthermore when pressing escape the main menu opens up offering flexible transitions between the options. The interesting aspects of our menus is that they themselves are levels. To navigate through them players have to move a gem across a board with the options being finish tiles. In the campaign menu the finish tiles leading to locked levels are sealed off with a black gem on them. This makes locked levels unreachable until they are unlocked.

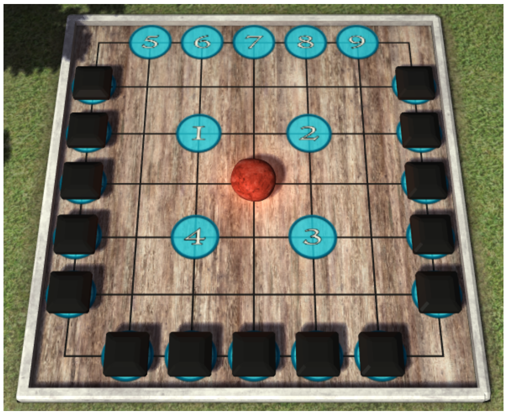


Figure 30: Levels are accessed via the campaign overview. Here the first 9 levels are unlocked. The rest are closed off with black gems.

6.1.4 Endless Mode

Apart from the main campaign with its 24 levels we implemented an endless mode. This mode procedurally generates new levels for the player to play for as long as the player wants to. To achieve this we integrated our generator from the Interim demo that we had lovingly called Bruce. The details of the implementation of Bruce are located in the report of the Interim demo. As the player solves the levels, the difficulty should gradually increase. So we had parameters for level generation change depending on the difficulty level ranging from 1 to 10. With each beaten level the grid size, number of gems and gem density of the level increases. Since the generator also returns a suggested solution for a generated level, we can also increase the number of steps that this solution needs to raise the difficulty.
Even though we have implemented a clear difficulty curve in the endless mode, we still encourage players to play through the campaign levels first and then move on the endless mode.

6.2 Experiences

Looking back on this term's project we have certainly gained a lot of experience in game development. These experiences including difficulties, successes and our experience with the theme are illustrated in the following.

6.2.1 Difficulties

Projects almost always come with a heap of difficulties, usually attributed to synergies within the group, technical nature or from setting the expectations too high from the beginning. This semesters projects difficulties were almost exclusively from a technical nature and something that comes from trying to write a game from the ground up without using a ready to use engine. These are factors the we had already anticipated and were used to due to last semester's project. This means that there is barely anything to mention in terms of difficulties except for the technical challenges we brought upon ourselves. One thing can be said for certain: Even though we took the challenge to not use an engine and managed to power through the issues: Developing everything on your own is very difficult and should not be taken lightly.

6.2.2 Working with the theme

When we first heard about this semester's theme we were really glad as we thought that this is something we could easily work with. After starting to iterate through ideas that could fit the theme we quickly realized that it might not be as easy as we initially thought it would be. We therefore took an extended period thinking about our initial idea that fits within our perception of Order and Chaos and were quite happy with what we came up with. During the other milestones our confidence in the idea started slowly to fade as the game is essentially not chaotic at all but after the playtest we were again reassured when we realized that for new players it does quite feel like a chaotic game.

6.2.3 Greatest success

The greatest success for this project was the implementation of the endless mode which was met with a lot of doubt initially. One aspect that we are especially proud of is that we are able to generate these levels in a reasonable amount of time so that there is no real interruption in the flow of the endless mode. This quick calculation of new and solveable levels is realized using a handful of mathematical/optimization tricks implemented by Felix Brendel who deserves recognition for this accomplishment. If you want more information on how he did it you can read it again in the Interim project report. Additionally we think that we were able to create a very fun and chill but at the same time deep puzzle game that relies on very basic mechanics. Considering that there are only 4 active gem effects for now we can see that the possibilities for this game are nowhere near exhausted. We are further very proud that we were able to add several new features to the already existing skeleton engine that we used in last semester's project. These features include:

- Scheduler instantiation
- Raycast mouse interaction
- Better tooltips and interface fading
- Particle System
- Point Lights
- Transparency

6.2.4 Overall sentiment

We are very happy with the final version of our game and are proud that we were able to implement everything we set out to do. One factor that was definitely influential for this aspect was that we had already taken part in last semester's GamesLab and were used to the report and development schedule. We additionally noticed that the workflow within our group has improved tremendously and that we were much better equipped to split up and manage the workload.