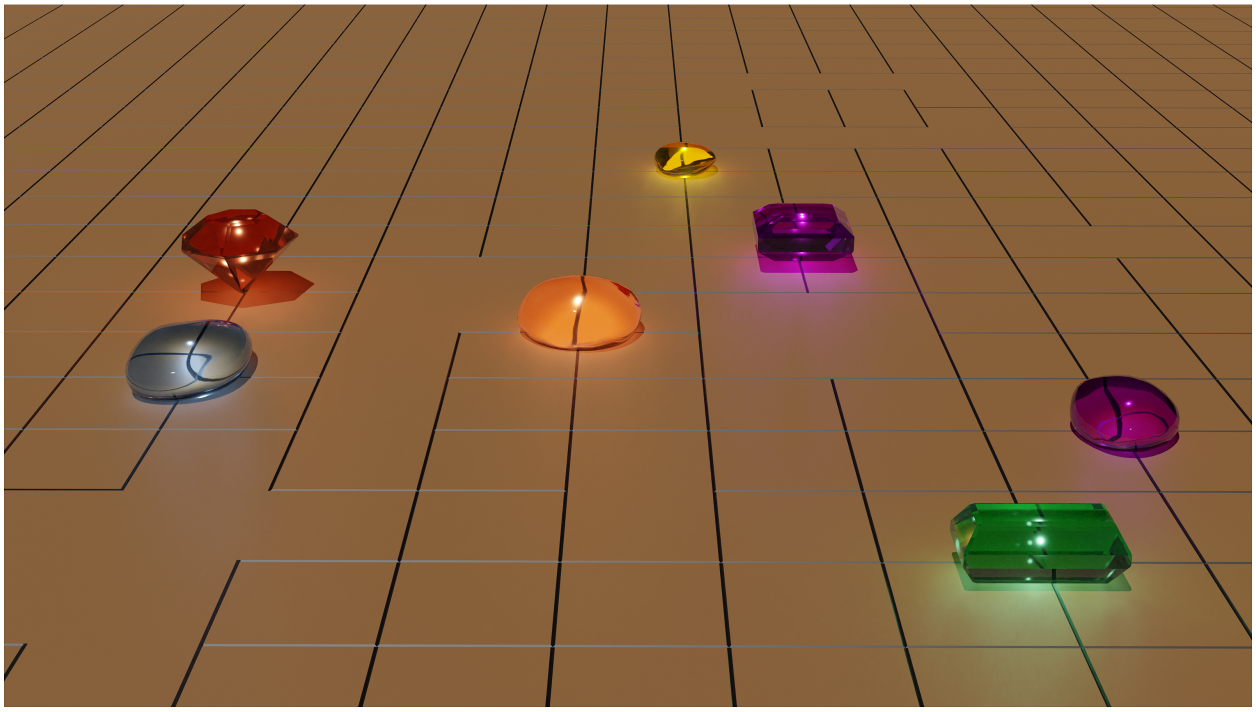
Interim Demo: *Gemji*

Team *DreiKopf*:

Felix Brendel
Jonas Helms
Van Minh Pham

May 2021



Contents

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.

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.

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, 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.

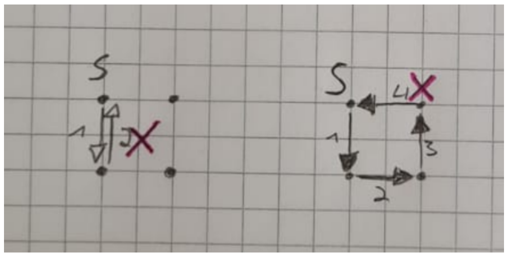


Figure 1: Board positions that were encountered during a search path, will not be considered for the search as they cannot possibly lead to the shortest solution

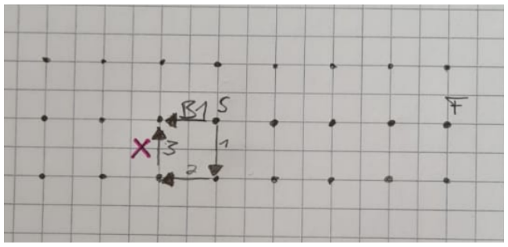


Figure 2: After all possible solutions (max moves = 4) starting with B1 have been checked, later solutions do not need to consider moving to B1 because it is a dead end.

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.

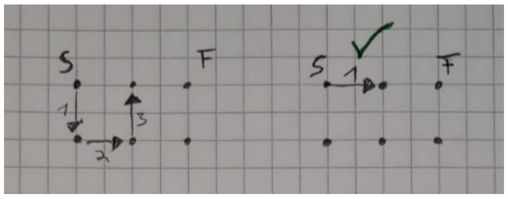


Figure 3: Even though the board position (max moves = 3) after [3] on the left has seen before, it does not necessarily mean that it is a dead end, it could be on the path to the finish, just not on the shortest one

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.

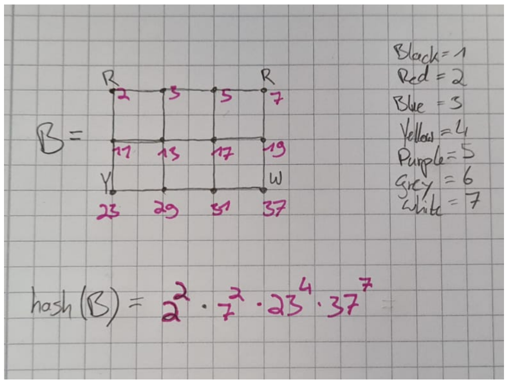


Figure 4: The construction scheme to generate an integer from a board configuration using integer factorization; each position on the board is assigned a prime number while the gem types are the enumerated exponents.

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 manhattan-distances 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.

1.3 Heuristic for generated Levels

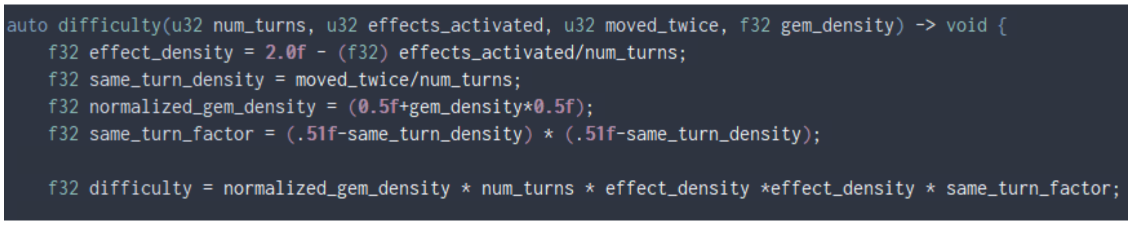
The aforementioned aspects for the random generation are not all elements that we use to control the generated boards with. Deciding to use randomly generated levels from the get go is something that introduces its own type of issues, mainly that we cannot directly control and determine how fun or interesting 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

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.

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".

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.

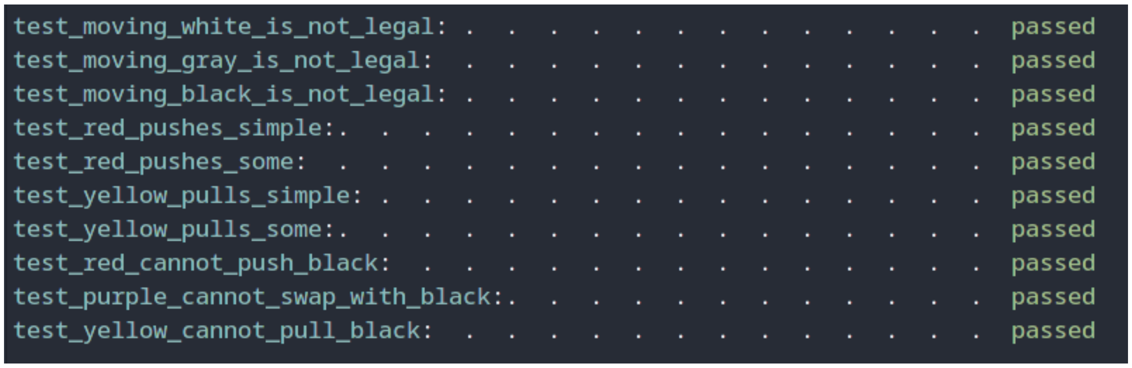
Additionally *Gemji* features white and grey Gems that cannot be moved by the player and only move from other Gem effects. Furthermore grey Gems are the only Gems that do not trigger other effects while white Gems do. Finally we included black Gems that serve as immovable obstacles.
Along with the effects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

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.

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.



```
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

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 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.

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.