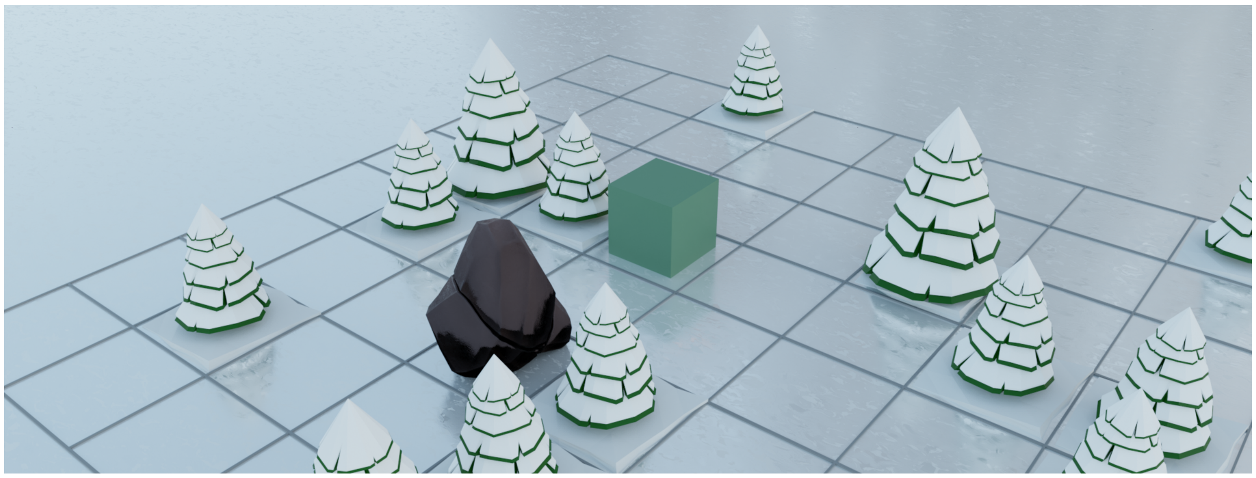
Interim Demo: qubi

Team *FünfKopf*:

Felix Brendel
Jonas Helms
Van Minh Pham

December 2020



Contents

1 Progress in Gameplay

For the interim demo we made the decision to focus on gameplay mechanics. As a lot of work went into building our own engine from scratch, we left the visuals at a basic level for the time being. In the following we elaborate on our progress so far.

In terms of gameplay we have been able finish up layer 1 and 2 mentioned in our proposal. This means that the player is able to move the cube on the field by using the W,A,S,D buttons. As intended the cube slides on slippery tiles without changing its orientation until it hits an obstacle or a wall. The main task then is to navigate the cube through the level in order to get to the finish tile. Inputs can further be buffered while the cube is moving. This makes the cube instantly move into that direction when it arrives at a location.

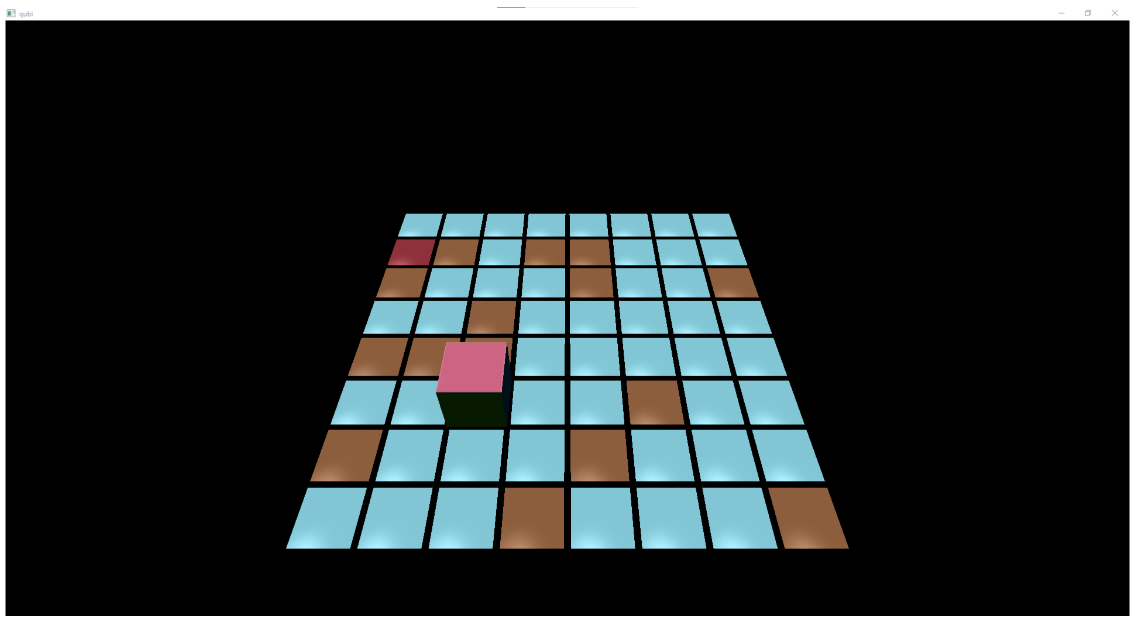


Figure 1: Sample Level 1 including the movable cube, blue slippery tiles, the red finish tile and brown obstacle tiles

When moving onto a dry tile, the cube will flip into that direction. In order to finish these levels the cube would have to reach the finish tile in one of three specific orientations. For that we color-coded the sides as red, green and blue with one color represented on two opposites sides of the cube. Thanks to the angle of the camera the player can always observe all different sides of the cube and plan ahead. Furthermore depending on the level the finish tile can be slippery or dry adding flexibility to level designs.

2 Engine Work

As the development of our own game engine makes up the major part of our project we will first go over our current progress on the specific components that we have implemented so far and how they relate to the layers of development. The engine was programmed in C++ while using the Vulkan Graphics API for the render pipeline.

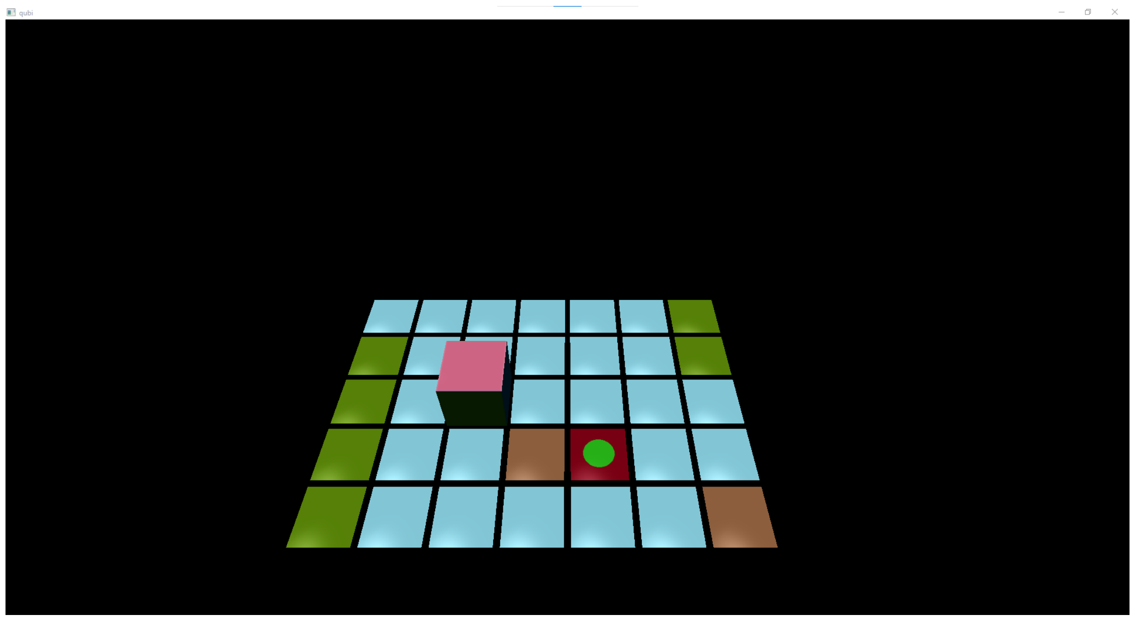


Figure 2: Sample Level 2 including the green dry tiles and red finish with green circle that requires the green side of the cube to be on top/bottom

2.1 Vulkan Initialization

The Vulkan Graphics API is a fairly young graphics API developed by the Khronos Group that was developed with the intent of leaving a higher amount of control to the developer. In turn this low level access means that the initialization and creation of the render pipeline is more extensive when compared to high-level level API's like OpenGL. Before we could start with the initialization we had to install a graphics driver that supports the Vulkan API. As Vulkan can be used on different types of devices and operating systems we had to some additional basic setup to use Vulkan in our engine beyond finding a driver that supports Vulkan and runs on our hardware.

2.1.1 Validation Layers

One one way for the Vulkan developers to increase the performance of the graphics API was to decrease the amount of error checking done by the driver. The Vulkan driver is mostly a direct abstraction except for cross platform functionalities of the hardware. As no type of error checking would mean that is very difficult to use this API the Khronos Group decided to allow the use of additional debugging layers called "Validation Layers" that work right in between the user's application calls and the API. These Validation layers are part of the SDK and allow validating parameters, validating texture and render target formats, tracking Vulkan objects and their lifetime and much more. The validation capabilities that we use are all packaged into a single layer called "$VK_{LAYERLUNARGstandardvalidation}$". More layers can be added and activated at the same time.

2.1.2 Loading the Vulkan Library

When you want your application to use the Vulkan API you have to use the vulkan loader which then connects the Vulkan API calls to the appropriate graphics driver. The Vulkan Loader is a dynamic library and is installed as part of the SDK provided by lunarg. To use the API functions in our engine we just had to use the LoadLibrary() function on the .dll file. After the library is loaded we can then use a Vulkan-specific function that provides the addresses to all Vulkan API functions. These functions can be divided into three levels: global, instance and device. Device level functions perform operations such as drawing, creating shader modules and data copying. Instance level functions create a logical device but before we can do that we first have to create an instance. To create a Vulkan instance we have to use a global level function.

2.1.3 Vulkan Instances & Extensions

A Vulkan instance is an object that gathers the state of the an application and allow us to create a logical device. Information that is part of the instance is the application name, name of the engine used to create an application and the enabled instance level extension and layers. Extensions are needed to allow extra functionality for the instance that they are enabled on but have to be hardware compatible. Example of extensions that we use are swapchain related extensions and to create an OS level window for our application. Extension can be enabled on the instance or the device level.

2.1.4 Creating a Vulkan instance

After preparing the application info, the number of extensions and the list of extensions we then save these values in the InstanceCreateInfo struct. The Vulkan function vkCreateInstance then just receives the adress to this struct in which all the needed information is stored. The other arguments for the instance creation are the address of the instance handle we want to create and an argement for allocation callback functions which we have not used so far. The function then returns a enum of type `vkResult` which provides feedback whether the function call was able to be resolved correctly. In general, all vulkan functions that can fail return a `vkResult`. We wrote a small macro that checks if the returned `vkResult` signals success and if not triggers a breakpoint.

```
vk_check_error vkCreateInstance(&instance_create_info, nullptr, &instance);
```

2.1.5 Vulkan structs and use

To create an instance we have to provde the aforementioned application info, extensions and validation layers we want to enable to the Instance CreateInfo struct. The application info is also saved as a struct which means that we use nested structs to incorporate all the info for the instance. This also represents the general philosophy used in Vulkan. All information that determines the functionality of your use of the Vulkan API is stored in structs. For better identification and to improve readability the first attribute of the Vulkan structs is always reserved to identify which information the struct stores.

2.1.6 Physical Device

After creating a Vulkan instance we can then use instance level functions to gather information about the physical devices (the GPU) available on the hardware. A Vulkan application can be run on many different devices which can each incorporate wildly different hardware that have different perfomances and different capabilities. In order of making sure to chose the correct hardware (onboard GPU and stand alone GPU card) and to check if the available hardware is capable of running our application we first gather information about it and confirm that all features are supported. To do this Vulkan provide three functions: EnumeratePhysicalDevices() which stores a representation of all available physical devices, vkGetPhysicalDeviceFeatures which stores the available features , and vkGetPhysicalDeviceProperties which stores general information about the physical device. We can then use this information to choose a suitable phyiscal and then create a logical device from it.

2.1.7 Logical Device

Logical devices perform most of the work in Vulkan: we can create resources, manage memory, record command buffers, submit commands for processing ect. Bottom line is that they include all the functionality we need to create a render pipeline. A logical device represents a physical device (GPU) but it is including all the features and extension we have previously activated and the information about the queues that can be requested from it.

2.1.8 Queue

The control of the hardware in Vulkan is implemented through queues. Commands to a queue are processed in the order they are submitted but there are different types of queues which are processed independently. Different types of Queues are not only for different functionality, not all operations are allowed to be performed on all queues.

2.2 Engine Structure

Obviously the graphics pipeline is only part of the engine. Every game engine needs to handle the games resources such as the scenes, game objects, etc. During the development a high importance was given to make the engine work as efficient as possible aswell as. In the following we will explain which systems are already in place and how they were implemented.

2.2.1 Resource Allocation

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.

2.2.2 Bucket Allocator

Meshes, textures, scenes all need to live in memory somewhere. But instead of heap allocating them all separately, we wrote an allocator to keep them together. The bucket allocator is basically a dynamic array of buckets, which are fixed-sized arrays. On startup the bucket allocator allocates itself a chunk of memory to hold the initial amout of buckets. When later all buckets are full, it allocates more buckets. No entries need to be copied, the only thing that needs to be updated is the dynamic array that holds the pointers to the buckets. Since no elements will ever move, it is safe to store and use pointers to them everywhere. When elements are freed, they are added to a free list, where they will be reused on the next allocation. The bucket allocator also provides functionality to iterate over all allocated elements. Bucket allocators are used for:

- Textures
- Meshes

- Scenes
- Materials
- Scheduler

2.3 Scheduler

The scheduler manages active animations and scheduled actions.

Animations are given by a start time, an end time, an aribitrary interpolant and an interpolation type. We can animate any variable in memory. One example for this animated field of view of the camera when finishing a level. It would also be possible to animate single vertices or material parameters but this is not in use at the moment.

The currently supported interpolant types are:

- vectors
- quarternions
- floating point numbers

More can be added later if the need arises. The basic interpolation functions where the ease functions just manipulate the variable t $\in [0;1]$ are:

type	adjustment for t
linear interpolation (also spherical)	
quadratic ease-in	`t = t*t;`
quadratic ease-out	`t = -(t*(t-2));`
quadratic ease-in and ease-out	`t = (t<0.5) ? (2*t*t) : (-2*(t*(t-2))-1);`

With this functionality, you can schedule even chains of animations in advance and continue with your game loop, as the scheduler will update the interpolants for all active animations every frame.

As an example, if you would want to animate a jump, where the horizontal movement is linear, while the vertical is quadratic you could split up the animations in three parts which are scheduled together:

- The upward movement, which is interpolated with ease-out
- The downward movement, which is interpolated with ease-in
- The horizontal movement, which is interpolated with linear interpolation

```
f32 from_z = qubi.transform.position.z;
f32 to_z   = qubi.transform.position.z + 1;
f32 from_x = qubi.transform.position.x;
f32 to_x   = qubi.transform.position.x + 2;

Scheduler::schedule_animation({ // upward movement
    .seconds_to_start   = 0,
    .seconds_to_end     = 0.6,
    .interpolant        = &qubi.transform.position.z,
    .interpolant_type   = Interpolant_Type::F32,
    .from               = &from_z,
    .to                 = &to_z,
    .interpolation_type = Interpolation_Type::Ease_Out,
});
Scheduler::schedule_animation({ // downward movement
    .seconds_to_start   = 0.6,
    .seconds_to_end     = 1.2,
    .interpolant        = &qubi.transform.position.z,
    .interpolant_type   = Interpolant_Type::F32,
    .from               = &to_z,
    .to                 = &from_z,
    .interpolation_type = Interpolation_Type::Ease_In,
});
Scheduler::schedule_animation({ // horizontal movement
    .seconds_to_start   = 0,
    .seconds_to_end     = 1.2,
    .interpolant        = &qubi.transform.position.x,
    .interpolant_type   = Interpolant_Type::F32,
    .from               = &from_x,
    .to                 = &to_x,
    .interpolation_type = Interpolation_Type::Lerp,
});
```

With this capability, it is easy to procedurally generate the animations that we need for our game. Of course in our case, the cube does not jump, but for more complex scenarios, like when flipping from dry tiles onto ice, start sliding and flip back on a dry tile, it is possible now to deterministically compute the resulting game state after every key input, and schedule the animations that lead to it.

If course, during the animations – so while the cube is sliding or flipping – player inputs should not impact it's trajectory. For that you can give the scheduler a "lock" which is just a pointer to a boolean for now, which will be set to `true` as soon as the animation is scheduled, and which will be set to `false` as soon as the animation finished. For now this is good enough as we expect to run the animation code on the same thread as the user input code. So with this we have a animation_locked boolean variable whaich we can check on user input, to check if we actually want to compute a player movement, or just keep the button in the player's input buffer, so it will be used as soon as animation_locked becomes false again.

Another thing that need to happen, is to check if the player finished the level as soon as the movement finishes. To do this, we don't check every frame for the finish condition, but rather schedule an action that checks for the finish condition on the exact time the animation finishes. Actions basically consist of a timer when they should run, and a functionpointer that will be called at that time; and since captureless lambdas kann be cast to function pointers we can even write them inline.

```
Scheduler::schedule_action({
    .seconds_to_run = animation_end_time,
    .lambda = [](){
      // check for finish condition
    }
});
```

C++ closures cannot be used as an action, as their size in memory varies, and thus cannot neatly be arranged in the bucket allocator holding all the actions (unless you use more levels of abstraction, like with `std::function` which themself heap allocate memory). On occasions we would need variable capture, actions have a fixed amount of space that can be used to store parameters to the function that should be scheduled.

Internally the Scheduler just consists of two bucket allocators, one for animations and one for actions. The scheduler gets called once per frame to update the animations and call the actions that are due.

The timestamps are stored as performance counters, since the easiest way to get a high resolution clock seems to be by calling `QueryPerformanceCounter` on Windows, and we wrote a similar function for linux.

2.4 Movement

Having a deterministic animation system is important for the player's movement, as our game is a puzzle game, where movements have to be exact. In our case, the game world consists of 2 tile types the player can be on: slippery and dry tiles.

We calculate the future gamestate for every input the user gives. This can be an iterative process, since one movement forces the cube into another one. This happens for example when standing on a dry tile and moving onto a slippery tile: The cube will flip onto the slippery ground and then immidiately start sliding in the same direction. So while simulating the future game state iteratively, we also at the same time generate and schedule the animations which manifest the movements to reach the calculated game state. This only works because we can calculate the start and end time of each movement and schedule the animations precisely to these times.

2.5 Game Logic

For the Game Objects that make up our scene we have right now implemented the following categories 'start pos', 'finish' and 'obstacles' and 'slippery tiles'. All tiles have a specific corresponding movement (sequence of animations) connected to them.

2.5.1 Different Tile Behaviors

Slippery tiles are the fundamental part of the game. When the cube reaches a slippery tile it will slide until it reaches an obstacle. The sliding animation is computed using the Lerp function on the position values of the transform matrix.

When moving on or onto a dry tile the cube flips over the bottom edge that corresponds to the direction that was input by the user.

When coming to a hold on a finish tile the camera will zoom out. Right after that, the next level is loaded. We further gave finish tiles different additional types. So a finish tile can either be slippery or dry. In levels in which the cube can flip, the finish tile also has a distinction depending on which of the three sides of the cube needs to be on top/bottom.

Hitting an obstacle leads to the cube stopping right in front of it. No further animation was necessary here.

2.5.2 Level Loading

A early Layer 3 goal for our project was the ability to load levels from a text file so we can streamline the level creation process that will be a major part for the alpha release milestone. The object and structure coordinates in the text file are grouped into categories and designated with 'begin category' and 'end category' which the map loader will then use to create a scene objects. Additionally the finish tiles have a extra keyword that determines whether they are slippery or dry and which color condition of the cube has to be fulfilled to finish the level. The rest of the tiles are automatically set to slippery.

3 Challenges & Design Revisions

When implementing the gameplay mechanics we encountered mostly minor issues which were resolved rather quickly. The win condition as well as the different behaviors of the cube when reaching specific tiles in itself were not our biggest challenges either. Our main concerns were all in regards to the implementation of the engine to make sure it runs smoothly.

- We spent a lot of time understanding theway to bind resources to the graphics pipeline and as a whole all the concepts and entities involved when rendering a scene on the gpu.
- Additionally we realized quickly that something like quaternions are really necessary if we do not want to keep track of the flips that occured, rotations are obviously not commutative.

However some concerns about our own goals arose:

- When we implemented input buffering, we noticed that once an animation is started, on the next frame the key will most likely still be pressed, and thus the key press would land in the input buffer, to be processed once the animation is finished. This is of course not as we envisioned. We made it necessary for the button to be be reset before it is eligible for the input buffer.
- And even though we implemented a method for loading levels from text files, we think for our purposes now, implemening them in C++ might actually have more benefits, as common aspects of the scene can easily be made default arguments or stuct members. With C++, there is the expressive power of a programming language to setup the scene, so that the C++ implementation is both more simple and concise. An advantage of loading levels at runtime however, is that you can edit the level while the game is running and just reload the level to instantly see the changes you made to the scene.