“Mechs vs. Minions”-Game
Milestone 1

Marcus Wunderlich

1. The Plan for Milestone 1

The main goal of our first milestone was a proof of concept (PoC) using Vulkan. This included displaying a game board with dummy *Mechs* and loading 3D model files like .obj-files. The latter is crucial since we want to use the real *Mechs* from the board game later in our game as models. Besides the already mentioned points from the rendering part of this milestone, we also planned to implement the Blinn-Phong reflection model.
Similar to the rendering part of this milestone, the main goal for the game logic part was a PoC. In this, the *Mechs,* that are displayed by the rendering PoC, should be able to move alternately. We expected this to give us a better understanding of what will later be important for the architecture of our game logic. In the end, we want to be able to store a game state and reload it afterward. Since the game state will be a central part of our game logic, we do not want to change its structure dramatically. Therefore, we planned to make the game state reloadable as early as possible.

2. What we actually did

Rendering

- ~~Vulkan~~OpenGL-PoC in C~~++~~#: display the board with grids and Dummy-*Mech* (A minimal graphics pipeline) [MW] [IW]
- mesh loading from obj-Files (vertices and textures) [IW]

+ camera movement [MW]
+ resource-management and multi-instance-rendering [IW]

~~- Blinn-Phong materials & lighting (without textures & shadows) [IW]~~

Game Logic

- turn based game logic (let two players move their *Mech* alternating) [MW]

~~- start and end new game [MW]~~

Misc

+ build a scalable infrastructure for the game (GameModel, Controller, View, GameServer) [IW]

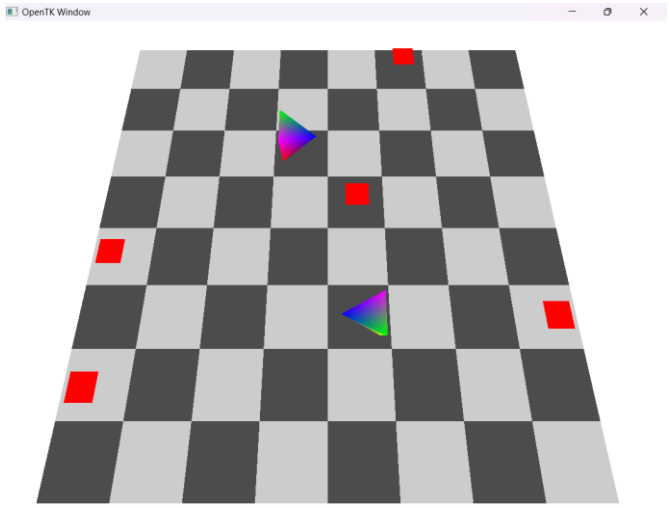
2.1. Failed PoC with Vulkan & C++

The first thing we had to do was to get to grips with Vulkan as we planned to use it for the rest of the project. Since neither of us had any experience with the Vulkan API, we agreed to practice pair programming, while we worked through the well-documented tutorial on vulkan-tutorial.com. In the end, we were able to display the `viking-room.obj`-file that is given as an example in this tutorial. Unfortunately, we were not able to extend this example to display several separate objects afterwards. As time was running out, we agreed that one of us tries to create a PoC using OpenGL, while the other one keeps trying to work with Vulkan. After the OpenGL-PoC was successful, we decided to use OpenGL at least for our prototype.

2.2. Successful PoC in OpenGL & C#

The change in the graphics API also brought a change in the programming language. Instead of C++, we are now planning to use C# as our main programming language. As we both had small but at least existing experience with OpenGL from the main lecture on computer graphics, it was easier for us to display one and later several different objects in OpenGL than it was for us in Vulkan. For the initial state of the OpenGL-PoC Ian used an OpenTK-Tutorial on Github[1]. Afterwards, we created models for the game board, the *Mechs,* and the *Minions*, and displayed them instead of the triangle used in the tutorial. The (Dummy-)*Mechs* are displayed as colorful pyramids pointing in the current line of sight of the *Mech*, and the *Minions* are small red squares.

[1] https://github.com/paulcscharf/genericgamedev-opentk-intro



2.3. Camera Movement

This feature was not part of our original first milestone, nor was it part of any other. As we found it useful to verify our created and loaded models from different angles, we decided to implement it in this milestone. For that, we use the keys W, A, S, and D to move the camera up, left, down, and right respectively. Additionally, one can move the camera target (alongside the camera position) to the left and to the right with the keys Q and E respectively. We capsulated everything that has something to do with the movement of the camera and/or its target in the class `CameraManager.cs`. The Math behind the camera movement is from the OpenTK documentation[2].

2.4. Turn-based Game Logic

Our first game logic component included the board state and the game state. The board state stored the pose of every *Mech* and the position of every *Minion*, while the game state stored a reference to the player whose turn it is, and the functionality to change that reference if the current player ends the turn. In this context, a player can be a *Mech* or an NPC like a *Minion*. If the player is an NPC, its move is calculated and applied automatically when it is the NPCs turn. For testing purposes, we added 2 *Mechs* and *5* Minions. Each *Mech* can be controlled by the keys U, O, and I, which make the current *Mech* turn left, turn right, and move in the line of sight respectively. We can show this functionality from an older commit in our presentation. Unfortunately, we could not integrate it into our preferable architecture (described in section 2.5) yet. In the current version, all *Mechs* move alternately one step forward, but cannot be controlled by the user yet. We will change this at the beginning of the next milestone.

[2] https://opentk.net/learn/chapter1/9-camera.html?tabs=input-opentk3%2Cdelta-time-input-opentk3%2Ccursor-mode-opentk3%2Cmouse-move-opentk3%2Cscroll-opentk3

2.5. Scalable Architecture

With an increasing number of classes and even more to come, it became necessary to think about the architecture of our game. One of the main design goals was to separate the Graphics from the Game-Logic, such that the visual representation of the game can be changed without changing its logic. Classes that are used by both packages are included in the Abstraction package. The App package is responsible for the communication between the Game-Logic and the Graphics package.

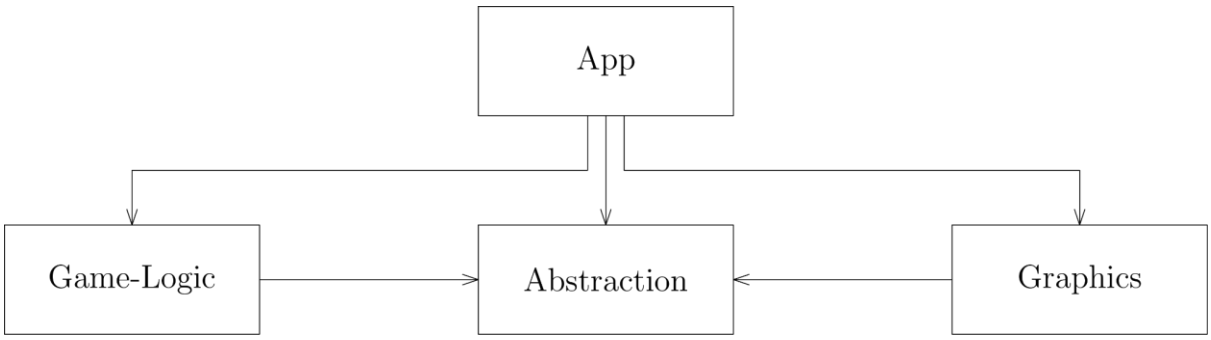


Figure 1: Architecture diagram

At the beginning the App package instantiates a `view` (from `Graphics`), a `controller` (from `App`), and a `model` (from `Game-Logic`). All these instances are passed to a `Server`-Object, which repeats the following loop until the window closes: the `view` asks the model if there are any state changes, since the last time the view updated. The `model` returns a list of all game states it had since the last loop iteration. Those game states are displayed by the `view`. After that, the loop awaits the next transaction from the `controller`. Currently, the (dummy-)controller simply waits 2 seconds and then returns the command to move the current *Mech* one step forward. Later, the controller could wait for user input, for example. The returned transaction from the `controller` is then passed to the `model`, which creates a new game state based on this transaction.