

Tank Simulator

Minor Skilled

Term 3 & 4 | 2022-2023

Nils Meijer – 466301



Table of Contents

Introduction	2
Learning goals	2
System design	2
AI behaviour trees.....	2
Custom editor/visualization design	2
Product Description	3
Project aspects I worked on:	3
Component-based system	4
Movement system	5
Camera controllers.....	6
Turret controller.....	7
Shoot system.....	7
Level design.....	8
AI behaviour tree	9
Damage registration system	13
Main menu	14
New input system	14
Progress comparisons	15
Evaluation 1.....	15
Evaluation 2.....	15
Evaluation 3.....	15
Final presentation	16
Workflow.....	16
Hour registration & planning	17
Quality Assurance	18
Future features & iterations	19
Self reflection	20
Conclusion.....	20
Appendices.....	21
Behaviour tree	21
Wheel collider.....	21
Mildots	21
Drag coefficient implementation	22
Bibliography	23

Introduction

In this report, I explain how I went about developing a tank simulator/game. Going over each component for the tank, such as a “Turret Controller”, a thorough description of the behaviour trees for the AI agents. The progress over the past 5 months is explained and shown, as well as how I had Quality Assurance done for my work. The report goes in-depth with various diagrams, visualizations and screenshots of the mechanics in the game. It’s explained why certain choices were made, what I learned after making those choices and what I should do better for my next game projects.

Learning goals

System design

One of the things I have always struggled with and want to improve on is designing/writing scalable systems in games. When a project starts with a poor codebase design, bugs will inevitably show up during development, costing time that can be better spent on actual features (and that’s money lost when working for a company). Therefore, I need to learn how to prevent making messy & hard-to-work-with systems.

I tried (and in my opinion, succeeded to a degree. Of course, things can always be improved) working on this learning goal by structuring my codebase into a “component”-based system, which is coupled to a Finite State Machine. Building the behaviour tree also taught me a fair bit about the concept.

AI behaviour trees

I worked on AI behaviour in previous projects, but those were always “Finite State Machine”-related. I wanted to expand my knowledge and skills on the general topic of AI, so I decided to learn about behaviour trees. They are another way of approaching AI design, and in my opinion more flexible.

Custom editor/visualization design

An essential skill for a tool programmer is to be able to design comprehensible and nice-to-use editor tools. Not necessarily just editor windows (think of scripts that have custom buttons, input fields, collapsible menus), but also gizmo visualizations to show how a mechanic works. This makes it easier to design behaviour as well as debug the system. Also keeping designer-colleagues in mind, as they are not able to easily change how the components work, so I should make it as accessible as possible.

Developing this skill increases my job potential for a tool programmer, since that’s one of the opportunities I am considering after graduation.

Product Description

For the past 5 months, I have worked on a tank game. The original idea was to develop a simulator close to how tanks operate in real life. However, this eventually evolved into a less realistic tank “game”. Because of time constraints and difficulties in developing certain components of the project, I couldn’t fully complete the idea I had while brainstorming in the first few weeks. In hindsight, it would have been better to have brainstormed a slightly smaller, less ambitious project so that I would have more time to focus on research and properly test & implement the features.

In the final product, the game starts in an interactive main menu, with an option to quit the game or to proceed to the next scene. In the next scene, the player can navigate around a desert-themed level and interact & fight with enemy AI tanks.

Project aspects I worked on:

Component-based system

- Movement system
- Camera controllers

Turret controller

Shoot system

My implementation of the formula can be found in the appendices (“

Drag coefficient implementation”).

- Level design

AI behaviour tree

- Damage registration system
- Main menu
- New input system

Component-based system

To have a structured codebase from the start to make the rest of the development of 5 months as easy as possible, I designed a component-based hierarchy. One of the requirements I set for myself was that I could reuse every component (if relevant. For example, a camera controller is not relevant for AI agents since they don't have a camera) for other tank entities. I should be able to attach a "shoot component" on both the player and an enemy instance, so it's important to make the component as generic as possible.

While developing the components, I found that it started to become harder to work with. Because there were different "states" the tank could be in, it was very hard to properly organize what input should be accepted (for example, to prevent the tank from being able to drive while it's exploded into a million pieces). I quickly realized there should be a system in place to handle these transitions, and so I started working on a State Machine. Again, it be as reusable as possible. I wrote the FSM into a template-format, so I create a "tank-FSM", a "camera-FSM", and a FSM". This made all logic incredibly easy to organize, access (but only need to) and prevents

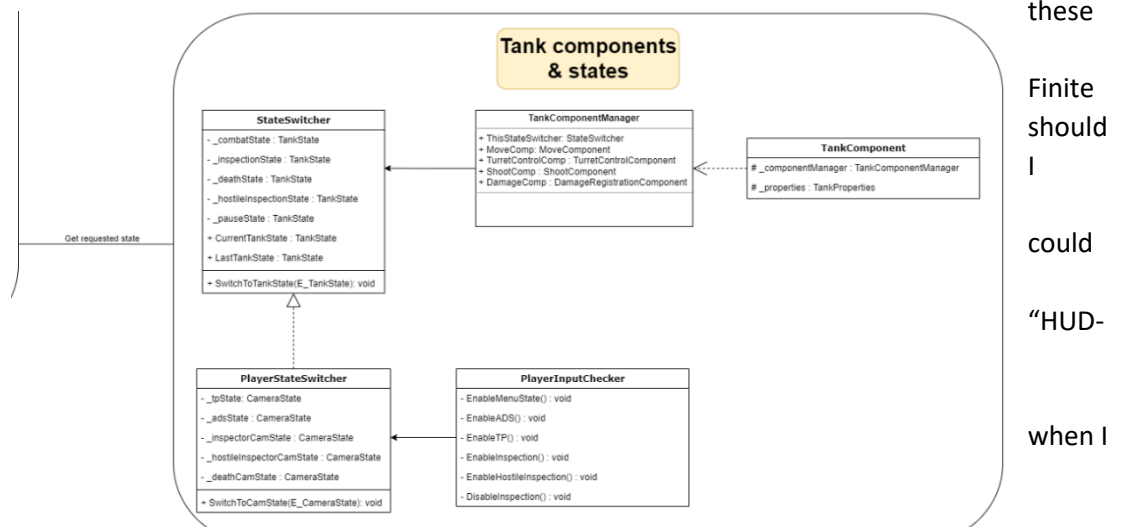


Figure 1 Tank components & states diagram. Shows communication between modules.

writing the same logic multiple times.

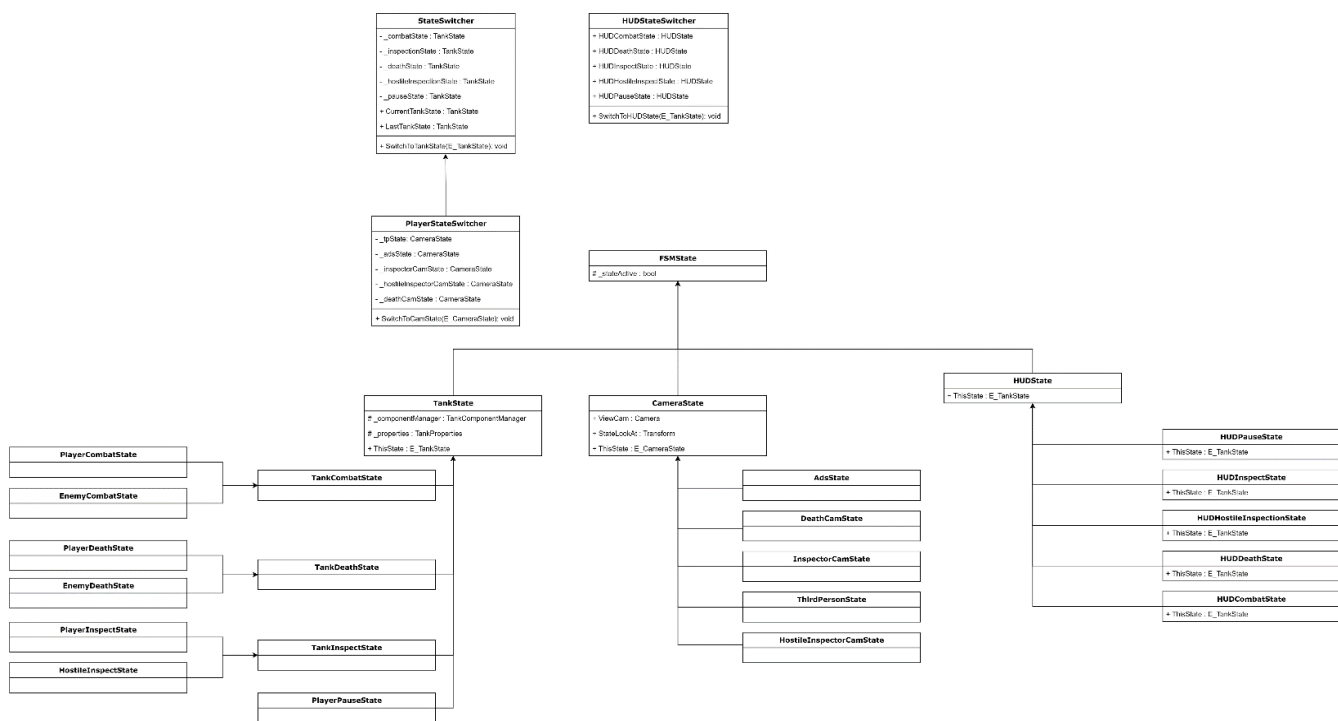


Figure 2 All FSM states

Movement system

I started out developing this system with the ambition to simulate real-life tank tracks as closely as possible. With a special type of collider in Unity, which is “WheelCollider”, I would be able to create a tank track similar to the example in Figure 3. The wheels inside the track would respond to terrain changes, due to the suspension system. The only thing different would be that the WheelColliders would also be the contact point to the ground, rather than the track. In a real tank, the front and rear sprockets are “powering” the track, which then makes contact with the ground. Due to that friction, the tank can move.

However, during a large part of the development, I struggled with making the WheelColliders behave the way I expected them to. The tank wasn’t even able to rotate properly. The way tanks handle rotation is to move the track opposite the tank is trying to move towards. Example: if the tank should rotate clockwise (so to the right), then the track on the left should be activated. Optionally, the track on the right can be reversely activated to create a stronger rotation force. This is how I tried it with the WheelCollider approach as well, but it didn’t work as expected. What happened was that it would either barely move at all, or the tank would need to travel a very large distance forward before having rotated the desired angle on the Y-axis.

Another reason why I struggled with implementing this WheelCollider approach was that, no matter how much research I did, I couldn’t wrap my head around the WheelCollider parameters that could be customized. Think of variables such as “Extremum Slip” and “Extremum Value”, and how they influenced each other. I spent too much time trying to find a balance in these values.

Since one of my learning goals was to improve my editor scripting skills, I worked on a custom editor window (so a separate window, not buttons and other GUI elements attached to a component). In “Figure 4 Custom tank editor”, you can see what the window looks like. As you can see, the most part is related to the Wheel Colliders.

However, I decided to greatly simplify the whole movement component, and just resort to just adding forces to the Rigidbody of the tank. When I look at how much time I spent on the movement part of the tank, I spent way too much time developing it. I suppose that’s a lesson for future projects.

Scrapping the Wheel Collider-related functionality also makes most of the editor window obsolete. I still learned a lot from designing/writing this editor window though.

When driving the tank, the FOV is widened to provide more visual feedback and add “juice” to the experience.



Figure 3 Tank tracks example (source: <https://edition.cnn.com/2023/03/21/politics/us-abram-tanks-accelerate-ukraine/index.html>)

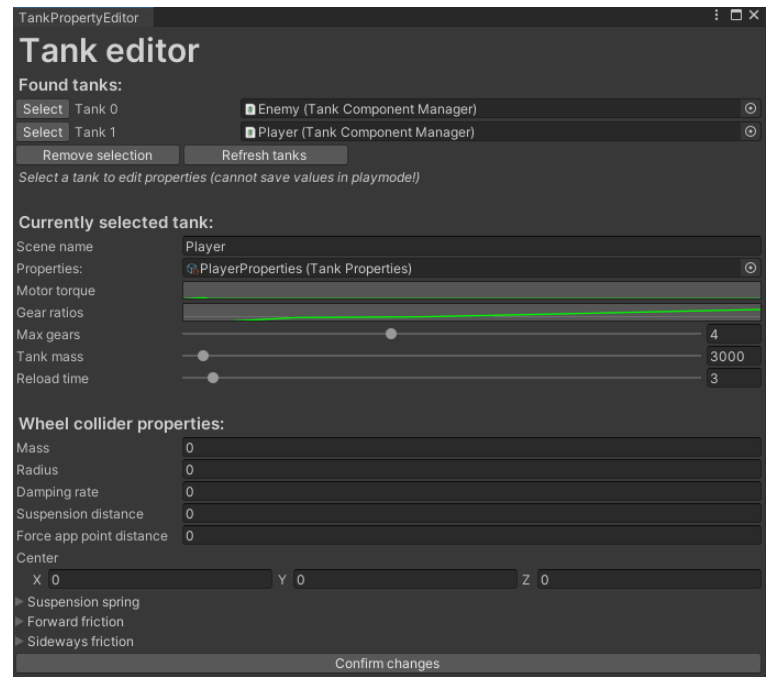


Figure 4 Custom tank editor

Camera controllers

As I was doing my research, I noticed many games had different (or multiple) camera views. In the beginning, I had 3 different views, which would be:

- **3rd person**

The camera follows the tank at a set distance, and can rotate a full 360° around the tank, as well as a limited range of motion on the x-axis. This will likely be the camera view that players use the most, to navigate around the scene and shoot targets that are relatively nearby. For this camera view, I did my best to visualize how the camera worked, by showing the boundaries for movement, and how it relates to the barrel

- **1st person**

A camera placed on any of the 4 positions marked in Figure 5, unable to move by itself. It was an attempt to mimic reality, but it wasn't a gameplay element I had any use for. That's why I decided to remove it, improving gameplay quality by decreasing complexity, at the cost of decreasing realism.

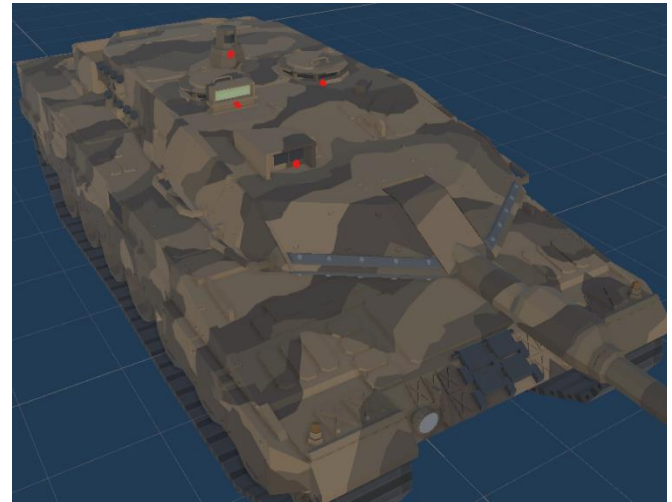


Figure 5, 1st person camera positions (red dots)

- **ADS**

A camera placed right above the barrel, which follows the y-rotation of the turret, and the x-rotation of the barrel. The player is able to zoom in using the right mouse button up to 5 times. This is done by decreasing the FOV, rather than physically moving the camera to a position in the distance. This would have introduced issues, such as moving into walls, being able to see behind walls and other edge cases that could not be solved which is why I changed it to FOV adjustments. To make zooming feel better, I let the camera lerp between its current FOV and the new FOV. This makes the transition feel less instant and much smoother.

This camera view should be used to target enemies at larger distances, since the zooming ability helps with hitting those targets with high accuracy.

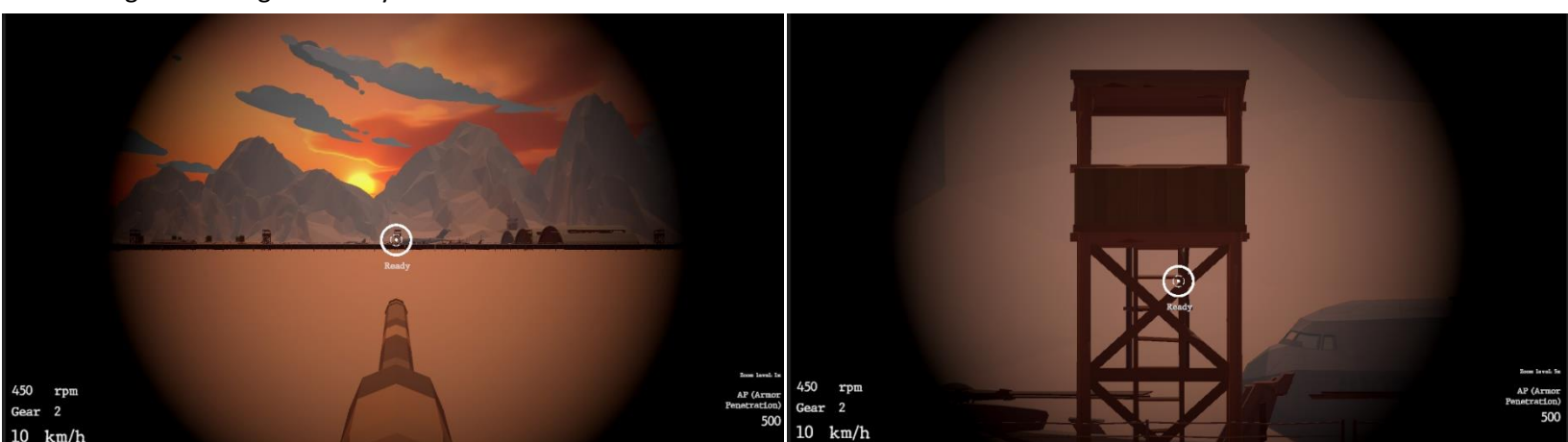


Figure 6 Min & max zoom levels

Turret controller

A system to control the rotation of the turret and barrel of any tank this component is attached to. The player can do this by rotating the correct input axis (mouse, joystick). This will instantly rotate the “target rotation transform”. The turret uses this as a target to which it will rotate towards. This component is intertwined with the camera component, as demonstrated in “Figure 7 Gizmos camera & turret control component”.

It’s visualized how the camera position relates to the “look at” position of the turret. If the camera is at its lowest position, the turret aims as high up as it can. When the opposite is true and the camera is at the max height, the turret aims straight forward.

These boundaries are of course easily editable by a designer, so they have complete freedom of how far the camera and turret can move up and down.

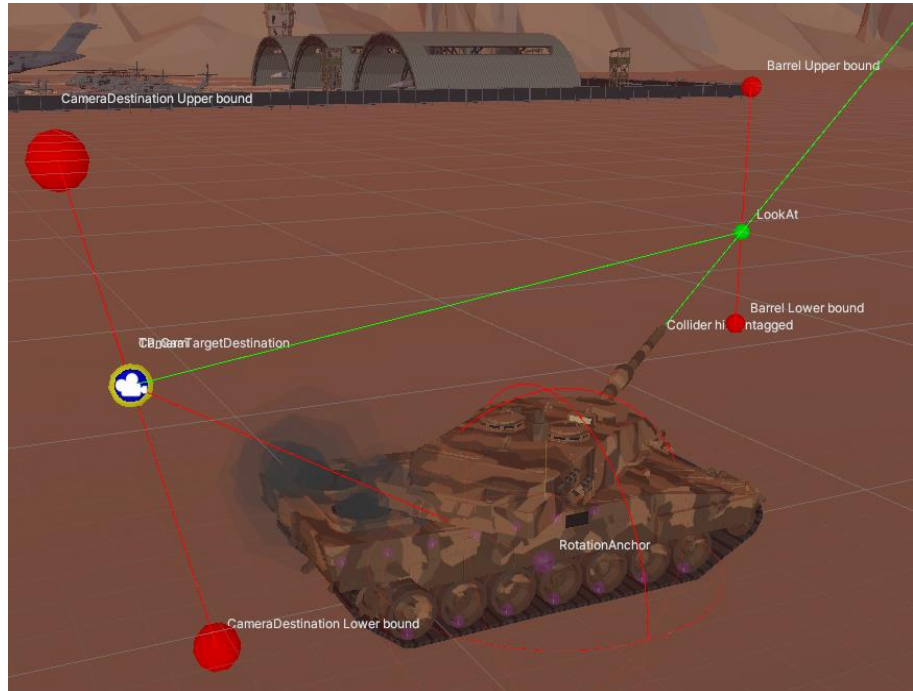


Figure 7 Gizmos camera & turret control component

The speed of the camera & turret rotation (different variables!) can be customized as well, so the rotation can be instant, or more realistic and the turret will slowly follow the camera rotation.

Shoot system

Developing this component was another way to try and make the game a bit closer to reality. When using the proper input, a shell is spawned at and shot from the tip of the barrel.

During early/mid development of this feature, I spent a lot of time on a “ranging” system. During the research phase of my minor, I found that a lot of games had some kind of range finding way. For example, you would manually measure the distance to a point using UI called “mildots” (or the distance was measured automatically), and you would have to adjust the rotation of the barrel on the x-axis to account for shell drop, making sure that the shell would hit the target in the distance accurately. Refer to “Mildots” in the appendices for an explanation.

Unfortunately, this didn’t work out. I decided to entirely scrap the range-finding feature, and to let the player rely on their own intuition. While playing, I found that it’s mostly a matter of finding the right camera view. If the selected camera view is ADS, then the player only has to use the correct zoom level.

One thing that has worked out, is the shell drop-off. Some distance after firing a tank shell, the shell will start to lose velocity & therefore, height. This is implemented with the Drag Coefficient [1], although I haven’t gone too in-depth on this. I spent a lot of time on the mildots system and there were still other time-intensive features I had to work on.

I developed this feature after having a conversation with a

With this shell drop-off, the game is a bit closer to a simulator (assuming I used the correct values in the formula). The formula for this mechanic [1] is the following:

$$\text{Drag force} = \text{DragCoefficient} * 0.5f * \text{AirDensity} * \text{currentVelocity}^2 * \text{frontalArea}$$

My implementation of the formula can be found in the appendices (“

Drag coefficient implementation”).

Level design

In order for the player and AI (which I still had to start working on at that point) to have an engaging environment, one of the tasks I gave to myself was to design a moderately-sized level. It had to be large enough to have enough movespace for the entities in the level, but small enough so the minor doesn’t become a “how to design levels” minor. In Figure 8 The level I designed, you can see a overview.

While designing the behaviour tree (see “AI behaviour tree”), I found that the agents don’t handle the slopes/hills in the level well at all. I used them to bring some more variety to the desert environment. However, the agents weren’t able to reach certain patrol/cover points due to these slopes. Even through tweaking the NavMesh (& agent) properties, I couldn’t manage to have them smoothly go over these hills. Therefore, I had to compromise and remove any height differences. Still, the agents occasionally have some trouble moving through the level (even though the NavMesh is clear and there aren’t any obstacles).

With feedback from artist fans and my QA, I added some much-needed postprocessing. This makes the scene more interesting to look at, rather than a dull, flat-coloured boring level. On his suggestion, I also tried to add volumetric lighting using a 3rd party asset, but as of writing this paragraph, I couldn't get it to work unfortunately.

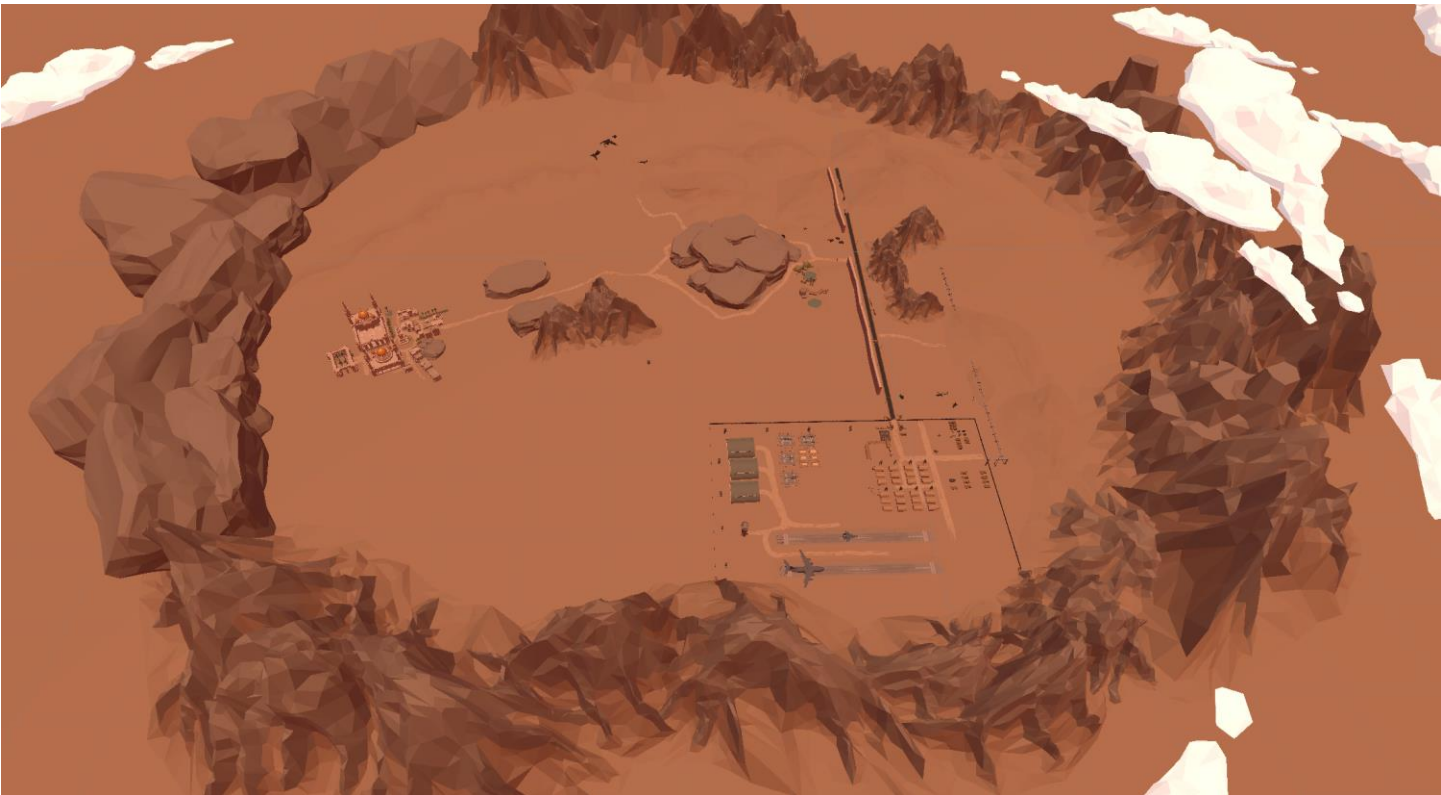


Figure 8 The level I designed

AI behaviour tree

To blow more life into a game, in my opinion, it needs a presence other than the player itself. That's why I decided to add AI agents. They are able to move around the world and make their decisions by the use of a behaviour tree. To design this behaviour tree and convert my thoughts into usable AI behaviour, I started with 4 "core nodes", from which all other behaviour would derive. The 4 nodes are described next. You can see a visualization of every node (kind of like pseudocode, including what type of node it is. For a detailed explanation of what a behaviour tree is and how it works, refer to the "Behaviour tree" appendix.

Behaviour tree branch	Description	Visualization
-----------------------	-------------	---------------

<p>Cover</p>	<p>If the armor stats of any tank part reaches a certain threshold, the AI will generate a set of points on the NavMesh. These points are all candidates for positions of cover. After generating, the AI will filter out the best options, and then choose the closest option. The AI will move to the chosen cover point.</p>	<pre> graph LR Root[Selector - Cover node] --> S1[Selector - Check armor stats] Root --> S2[Selector - Find new cover position] Root --> A1[Action - Move to cover position] S1 --> C1[Condition - Check armor Check if is currently repairing (w returns FAILURE) Returns SUCCESS if armor < certain percentage Returns FAILURE if armor > certain percentage] S1 --> C2[Condition - check if already in cover Returns SUCCESS if is in cover (can start repairing) Otherwise returns failure] S1 --> A2[Action - Repair armor until full Return SUCCESS when armor is fully repaired for all parts] S1 --> A3[Action - Set CanGenerateNavPoints to true (cover is needed so generation is enabled)] S2 --> I1[Inverter] S2 --> C3[Condition - check if already in cover Returns SUCCESS if is in cover (exit cover finding loop) Otherwise returns failure] S2 --> C4[Condition - check if has no valid moveposition. Returns FAILURE if is not valid Otherwise returns SUCCESS (exit cover finding loop)] S2 --> S3[Sequence - Sample and filter cover position] S3 --> A4[Action - Sample x generic points in node range] S3 --> A5[Action - Copy generic points to cover points collection] S3 --> A6[Action - Filter out positions that are behind the player compared to agent (dot product)] S3 --> A7[Action - Optional - Filter out positions that are too close to allied AI agents] S3 --> A8[Action - Raycast to all remaining positions to see if the player can see the agent at that cover position. Set point that's closest to agent as CoverPoint] S3 --> A9[Action - Set selected CoverPoint as MoveToPosition] S3 --> A10[Action - Set CanGenerateNavPoints to false (cover is found so generation is not necessary anymore)] A1 --> S4[Selector - on reached destination] S4 --> C5[Condition - check if point has been reached] S4 --> A11[Action - Set IsRepairing to true (inspector)] </pre>
--------------	---	---

Figure 9 Cover node visualization

<p>Patrol</p>	<p>The AI generates a set of points on the NavMesh. This time, these positions are possible patrol points. Again, the AI will filter out the best patrol points, and then moves to the chosen patrol point.</p> <p>At first, I was planning to manually place patrol points (empty transforms) around the level from which the AI would filter a proper point, but that wouldn't scale well at all, if I decide I want to adjust the level design or create new levels. So I went for "random" point generation on the NavMesh.</p>	<pre> graph LR Root[Sequence - Patrol node] --> S1[Sequence - Execute patrol] Root --> S2[Sequence - Handle patrol point generation] S1 --> C1[Condition - Check if can generate navpoints Returns FAILURE if not possible else returns SUCCESS] S1 --> A1[Action - wait for seconds Returns FAILURE if currentTime > 0 Returns SUCCESS if currentTime = 0] S1 --> S3[Sequence - Get target patrol point] S1 --> A2[Action - Set transform.position = transform.forward as turret target] S1 --> A3[Action - Rotate Turret to point] S1 --> A4[Action - Move towards current patrol point] S1 --> C2[Condition - Check if point has been reached Returns SUCCESS if point has been reached] S1 --> A5[Action - Set ShouldCountdown to true] S2 --> C3[Condition - HasValidPatrolPoint Returns success if a valid patrol point (can reach it, path is assigned, otherwise failure)] </pre>
---------------	---	---

Figure 10 Patrol node visualization

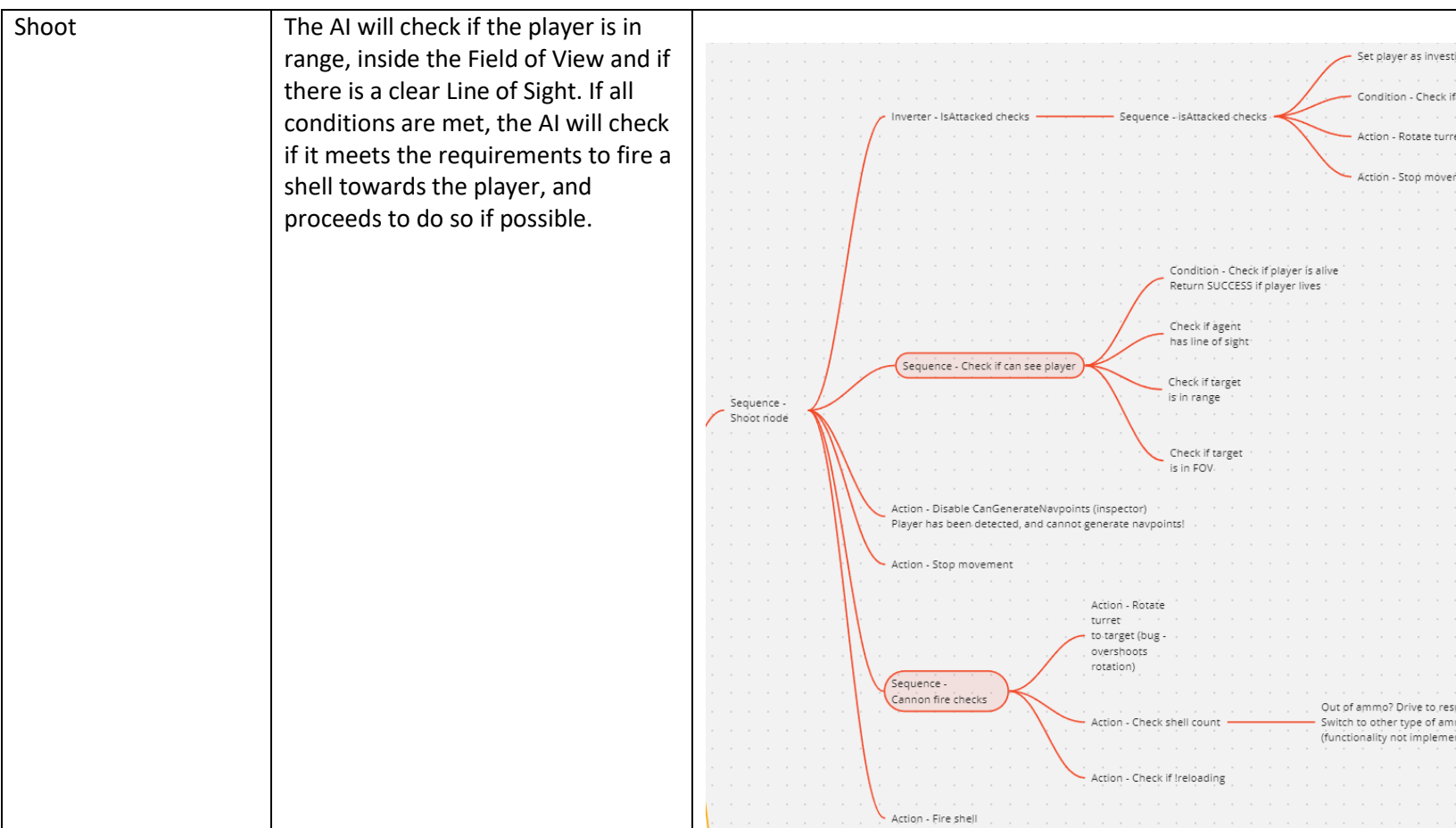


Figure 11 Shoot node visualization

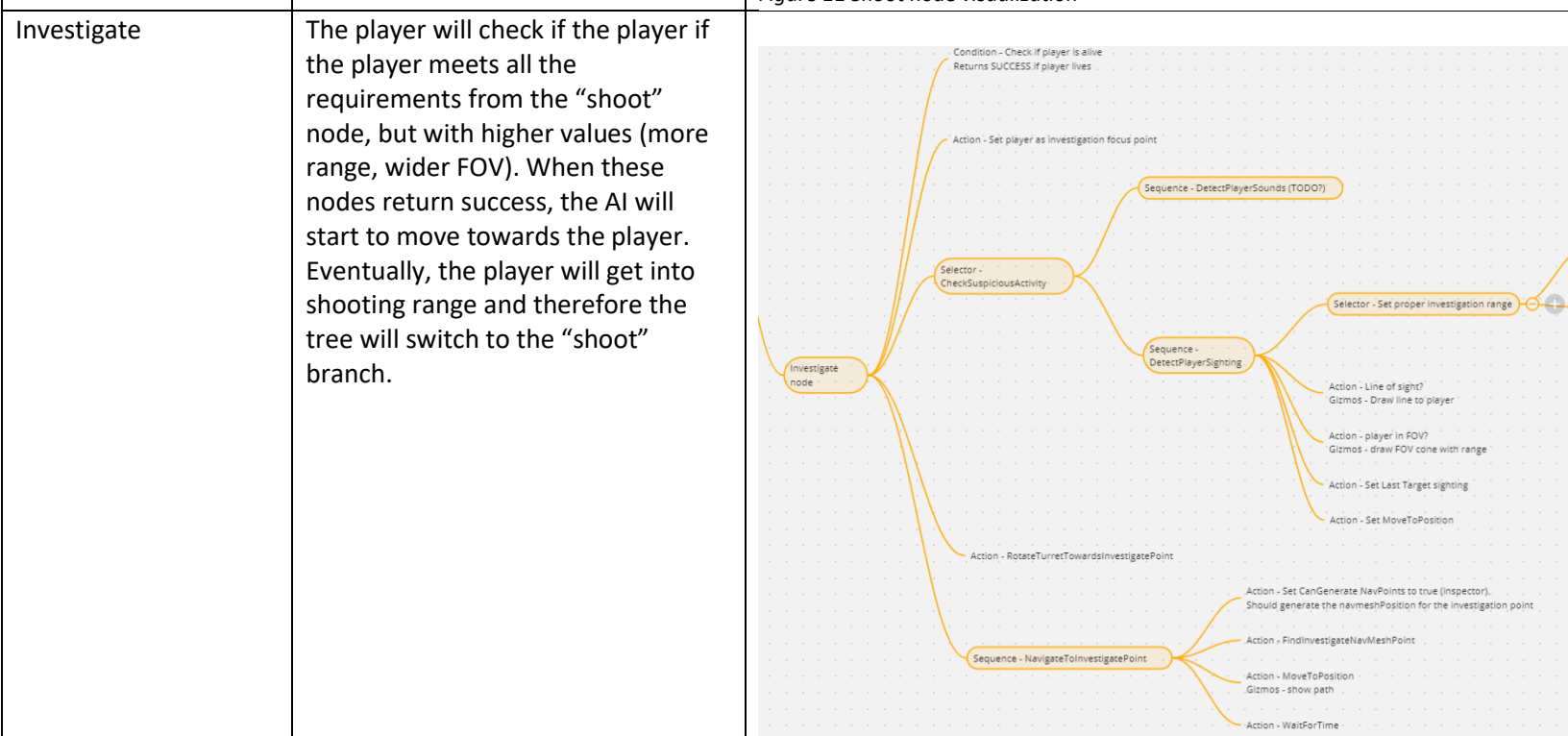


Figure 12 Investigate node visualization

Sometimes it would be rather difficult to debug what the cause of a certain problem was because I don't have a visual node editor (something to develop in the future, so it's possible to release this behaviour tree code as a plugin).

Designing the tree with the website <https://miro.com/> made it more doable to get an understanding of how I would imagine the final behaviour. I used this method to structure the tree in Unity as well, as can be seen in “Figure 15 Behaviour tree hierarchy”. In “Figure 16 Behaviour tree inspector”, the *base setup* for every node can be seen. Any logic node has this setup, as well as possible additional variables for that node.

Another way to understand why the behaviour didn’t work as expected and desired, was by logging the “nodeState” of all (or just the selected) nodes, as can be seen in “Figure 14 Behaviour tree console logging”.

To take one of the logs as demonstration: “SEQUENCE: **Branch** [2] Action – Set Player As Turret Focus – [log count]”.

SEQUENCE = Clarification on the type of parent node: selector, sequence (these are called “composite nodes”), or an inverter (which is a “decorator” type node).

Branch = is either **red**, **orange**, or **green**. Red means FAILURE, orange means RUNNING, green means SUCCESS.

Action = the type of child node. Can be either an Action or Condition. Although this type is just for debugging & organizing the tree. I haven’t made concrete “action” and “condition” nodes for them, unlike the composite nodes.

The advantage of a behaviour tree compared to a finite state is that I don’t have to manage a ton of transition conditions. Instead, I only have to check conditions that are relevant to *that* specific branch. Example: to transition into the cover branch, I only need to check if the agent needs to repair (armor < certain percentage). If that condition fails, I know the entire branch doesn’t need to be executed.

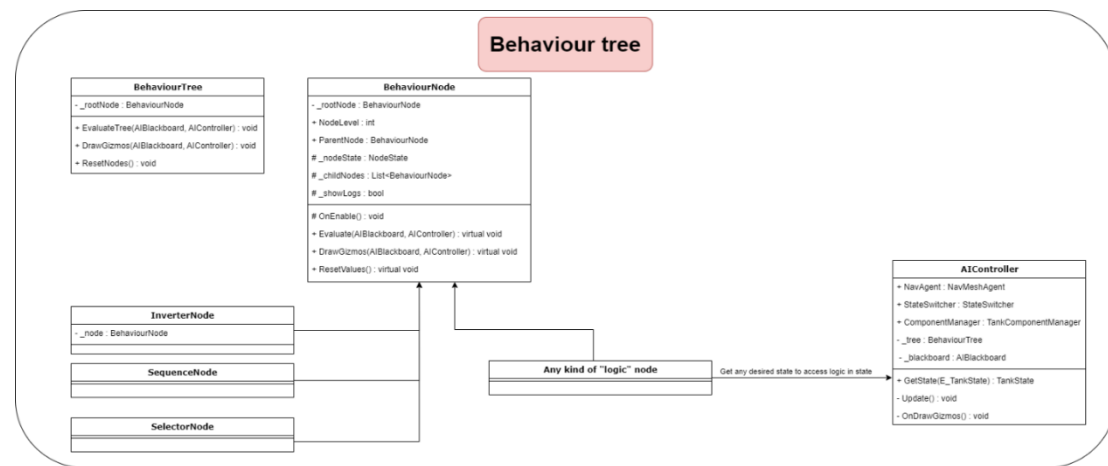


Figure 13 Behaviour tree diagram

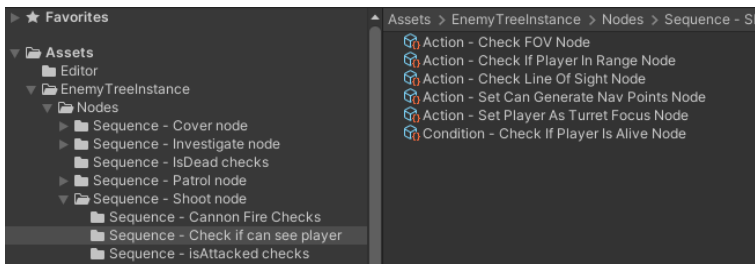


Figure 15 Behaviour tree hierarchy

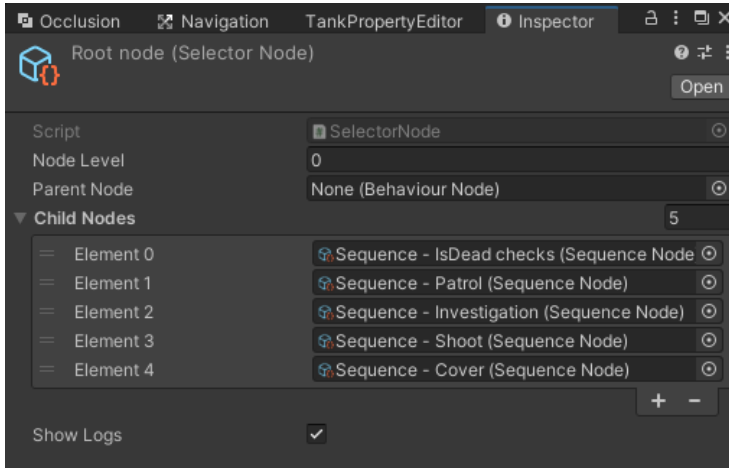


Figure 16 Behaviour tree inspector

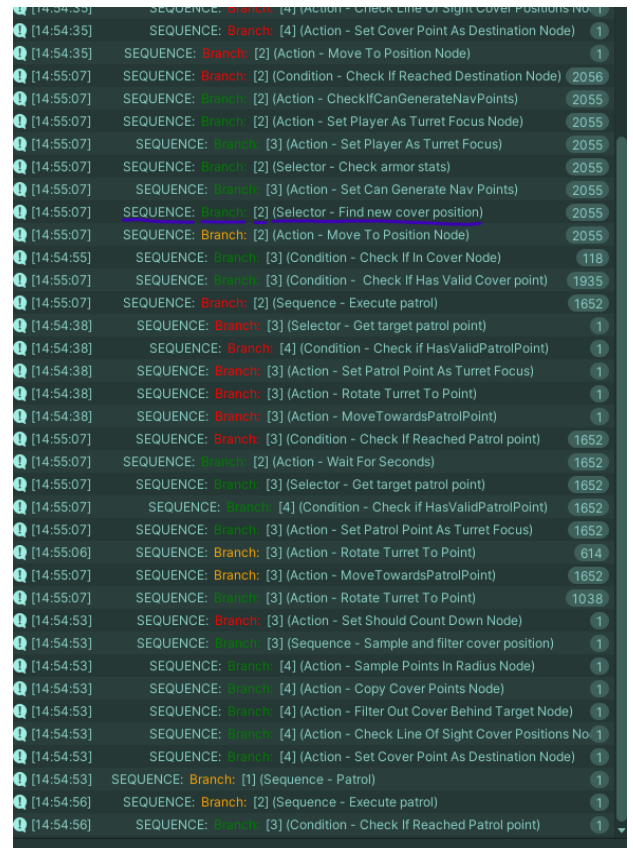


Figure 14 Behaviour tree console logging

Damage registration system

In a tank combat game, the tanks obviously need to be able to be destroyed. That's why I spent a few weeks on developing a damage registration/health & armor system, as well as a way to repair that damage.

With this component, the shells that hit the tank get registered and damage is applied to the part it hit. Once a part (can be the left track, right track, hull or the turret) has been destroyed (no armor and no health left), the tank explodes & therefore has been defeated. If a part has not been completely destroyed but its armor has been depleted, the armor can be regenerated. The same cannot be said for the health bar, as that's considered permanent damage.

The player can inspect their tank by pressing the corresponding input. The camera view will then switch to an "inspection view" (Figure 17 Tank inspection view), where the properties for each tank can be seen, and repaired if necessary. The camera can rotate around the tank in a full 360° circle, as well as zoom in to a certain depth.

This exact same component is being reused for the enemy AI, as I have made it as generic as possible. This way, I didn't have to write the same logic twice in a slightly different manner. For hostile tank inspection, I did make a "hostile tank inspection view".



Figure 17 Tank inspection view

Main menu

Every game of course needs a main menu to start the gameplay. I've always been a fan of these interactive menus. It makes the game feel more alive and adds to the atmosphere. For example, in my menu, I added a tank of which the barrel follows the position of the mouse. When I click any of the buttons, it shoots a shell, playing an explosion particle system. When loading the main menu scene, the gameplay scene is loaded asynchronously in the background. This prevents any loading screen or hickups/lag when clicking "play".



Figure 18 My main menu

New input system

At first, I was still using the input handling system I was used to. However, during one of the Guided Work meetings and one of my QA sessions, I received the feedback I might want to check out the new input system, designed with several input devices in mind. This way, it's easier to add new ways of input later on in development. It took some time and getting used to the new approach of handling input, but I got at least the basics of how it works down. Overall, it's more robust and efficient. In "Figure 19 New input system implementation", you can see my implementation of the new input system. It currently doesn't accept input from a controller, but it shouldn't be too difficult to implement that in the future. I didn't implement controller support because other tasks had a higher priority, even considering the expected low amount of hours needed to support controllers.

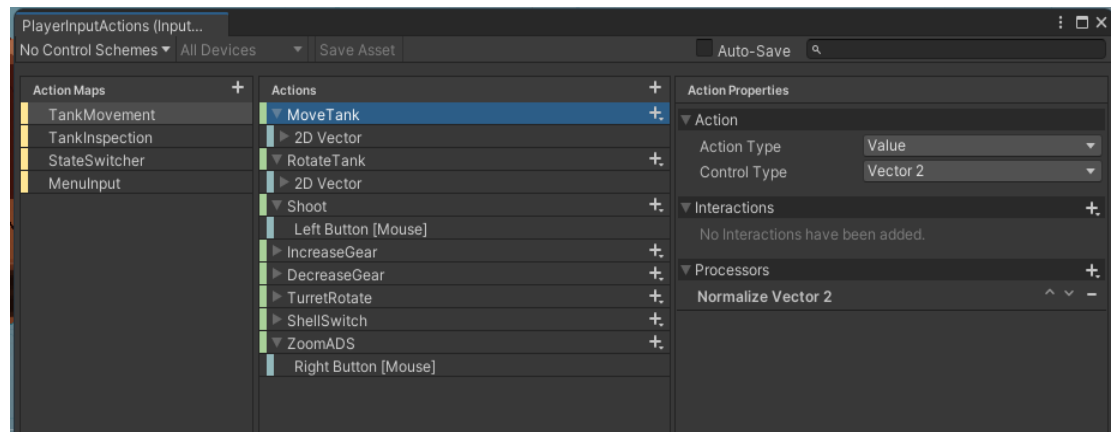




Figure 19 New input system implementation

Progress comparisons

Evaluation 1	Evaluation 2	Evaluation 3	Final presentation
			

Evaluation 1

During the first 4/5 weeks, I brainstormed and expanded on several concepts, from which I eventually chose the tank simulator. After doing research, I started working on a very rough prototype. I made a simple camera controller with 3 (later to be reduced to 2) camera views. The turret can be controlled by moving the mouse, and the barrel by scrolling the mouse wheel.

The move component & physics values of every tank in the scene can be edited through the custom editor. I also worked on a mechanic that allowed the firing of tank shells, but it was still very much in the early stages and the shells didn't have any effect on the surface they impacted with.

On <https://github.com/ngmeijer/Minor-Skilled-Notes/tree/main/Obsidian%20Mind/Minor%20Skilled>, you can find all the concepts, research, design documents and QA feedback I have written down.

Evaluation 2

At this point, I had designed a small level in which the player was able to move around. I also enormously improved the aiming system, as well as the turret control component. Rather than instantly rotating the turret, the camera now looks at an invisible "target look at", which rotates instantly. That's also the target the turret is rotating towards. This way feels and looks better than rotating the turret at an instant.

In this period, I worked on a range finding system which ended up getting scrapped. It didn't work out the way I planned so I decided to not waste any more time on trying to fix it.

Another main aspect I spent a lot of time on during these weeks was a damage registration component system. Refer to
“

Damage registration system” for a detailed explanation on this component.

To organize all these components and streamline the logic to a centralized place, I converted the existing (messy) codebase into a Finite State Machine. This allowed me to organize logic much better into classes that only get activated if certain conditions are met.

Evaluation 3

These weeks, I almost exclusively worked on an AI behaviour tree. I combined the behaviour tree with the previously mentioned Finite State Machine, so that I could reuse all of the component logic I made for the player. For example, a Shoot Component is attached to both the player and the agent. The agent can access that component through the state returned by its state switcher.

Final presentation

In the last part of my minor, I spent almost all of my time on polishing, fixing bugs and adding missing functionality. And of course, writing this report took a lot of time as well. I did my best on designing comprehensible flowcharts/UML diagrams and other visualizations, so it’s easier for “outsiders” to get an understanding of how my project is built, and how the classes communicate with each other.

Workflow

I did all my hour planning, brainstorming, functionality planning and documenting in a software called “Obsidian”. I hadn’t used it in a (for me) large scale project before, so I wanted to see how it would affect my workflow for such a long-term project. There are some things that can be improved, but overall I liked working with it.

During the first few weeks, I would go to school when I knew some of my friends were there as well, so I’d force myself to get out of the house more and make sure I’d get some social interaction. However, apparently it’s required to “reserve” the monitors/PCs in the XR lab and I’d often not know if I would be at school. There were a few occasions I managed to get a 2nd screen to work on, but often I just had to work on my small 17” laptop screen. So after probably the 1st evaluation, I really only came to school for the Guided Work meetings and just stayed at home the rest of the week so I could work more comfortably at my own PC. I’d often sit in a Discord voice channel with some friends to get some social interaction.

Every day, I would work from ~8.30-9.00 to around 16.00 so that equates to ~35 hours ($5 * 7$) a week. I usually keep working during lunch. In the last quartile however, I struggled with keeping up motivation and hours (together with combining working on redos, going to the gym, working at my sidejob, keeping my place clean and seeing my parents during the weekends) so those hours are more in the range of 30-32 hours.

Hour registration & planning

I tracked my hours using the software “Toggl Track”. The program is basically just a stopwatch, which remembers the time entries so it’s easy to see how much you worked every day and on what aspects of the project. In the desktop view, it’s possible to export a visualization of this data, which is what I used during the evaluation presentations and the final presentation to justify my hour registration.

Around every evaluation, I would look at my hour planning and check where it needed readjusting. During my internship, I also had to estimate how many hours I would need for a certain task, and review that planning after completion (and also update it during development, of course). That experience helped with doing the same for this minor. But as with all things, my hour estimation skills can still be improved (and will happen as I grow as a programmer).

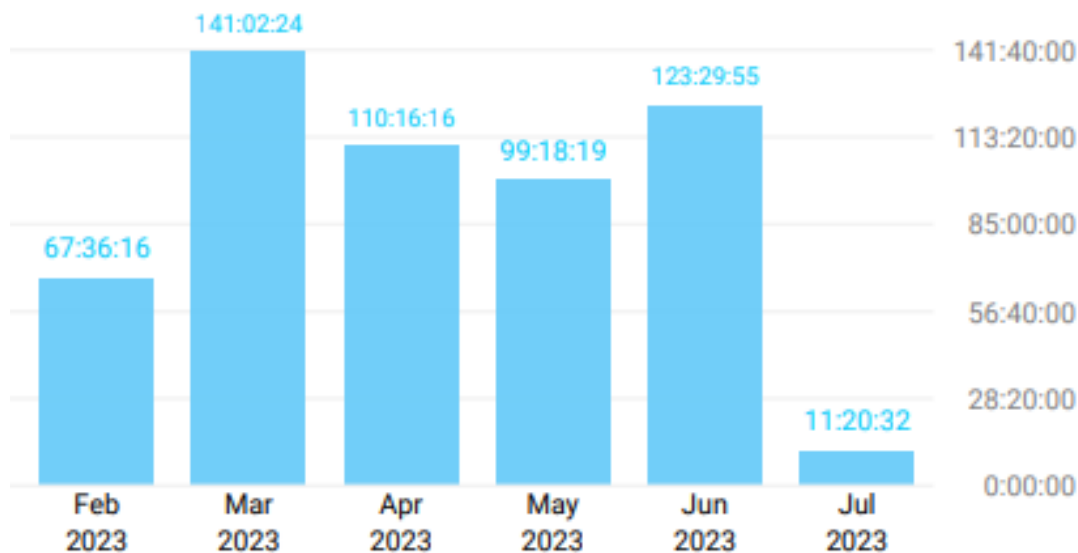


Figure 20 Total hours of the minor

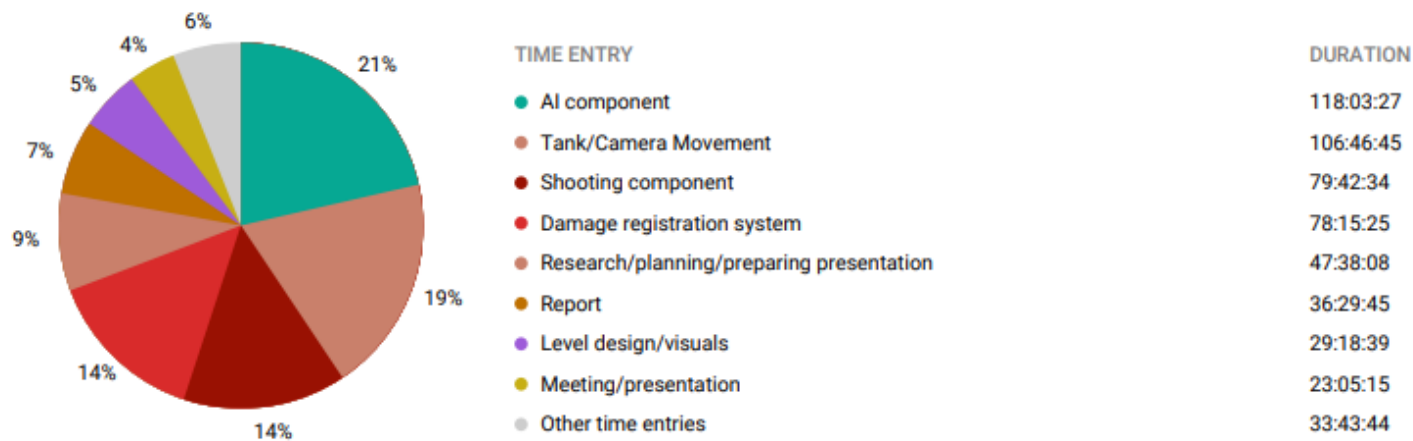


Figure 21 Hours spent per task

In “Figure 22 Trello logbook”, you can see the Trello logbook (<https://trello.com/invite/b/Dk4Xi2Uf/ATTIac958599d4bb543cf94510dddbb14241E2E2262D/minor-skilled>) I kept. I specified per week what I did on each day and how much time I spent on it. I could improve on documenting this, as I would usually do this every few days/weeks, so it sometimes became blurry as to what I did on a specific day.

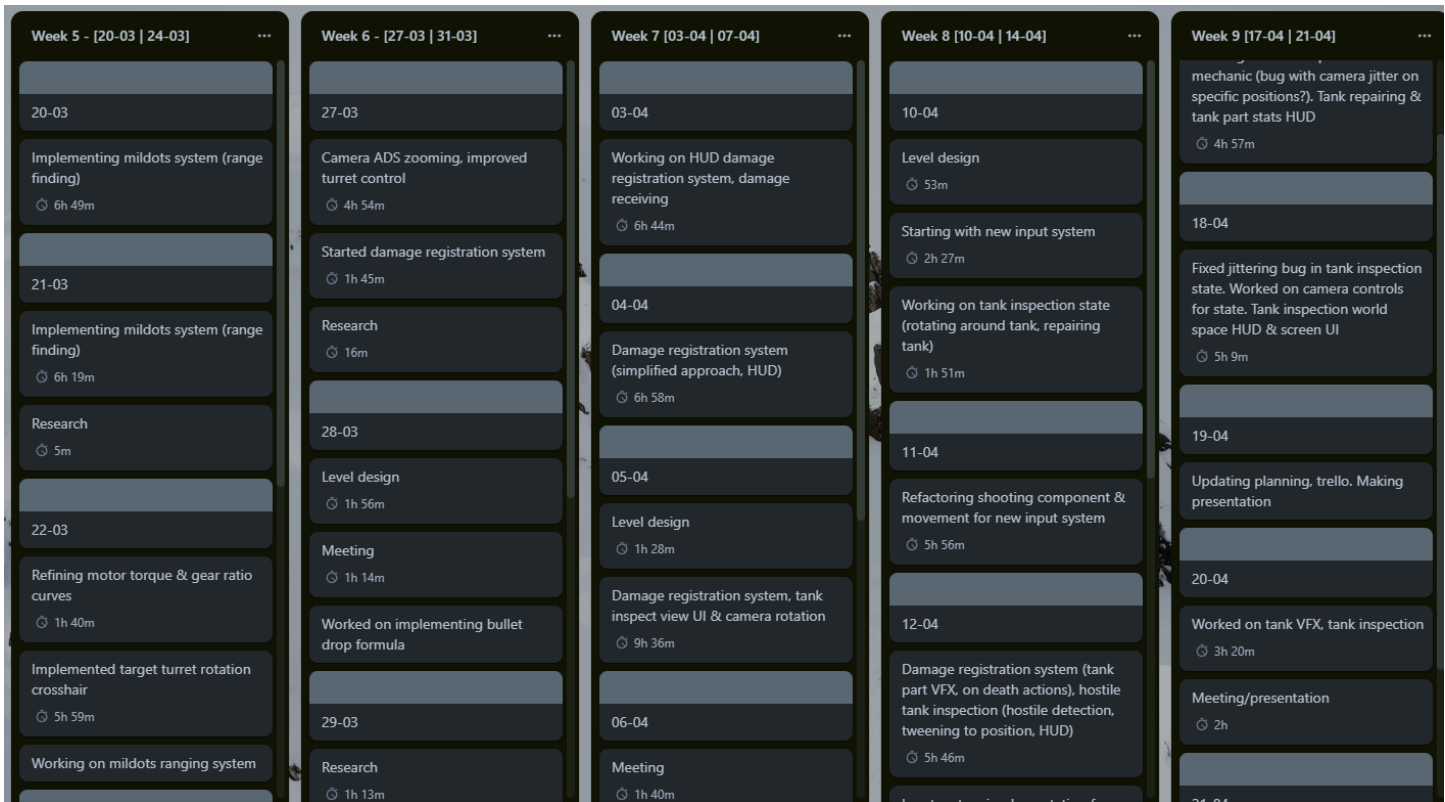


Figure 22 Trello logbook

Quality Assurance

Before any QA meetings, I was planning on doing a QA with professional programmer in the game development industry. For the first meeting, I messaged and emailed several studios working on a tank-related game. I also made posts in several Reddit game development groups. Unfortunately, I got 0 responses (and in the reddit groups, I even got downvoted). I ended up asking a former colleague from my internship at Total Reality as the first Quality Assurer. He replaced the previous lead developer (who was my internship coach), but left the company again after about 2 months.

For the 2nd, 3rd and 4th Quality Assurance meetings, I asked Hans Wichman and Yvens Serpa. They gave valuable feedback about the way I developed my behaviour tree, how I should handle feedback, visual design of the game and setting proper ambitions for myself.

I also made some use of ChatGPT during development, although it doesn't count as QA in my opinion. Rather as a way to help me get out of being stuck with a certain problem.

Future features & iterations

- Tank customization
 - o Originally a feature I was planning to develop for this prototype. I wanted the player to have more control of the tank they were playing with. Both in a gameplay aspect and a visual, eye-appealing aspect. For example, purchasing a light source that can be attached to the tank to make it easier to navigate

through a level when it's dark. This would be a gameplay customization. A visual customization would be buying a different skin for the tank (another camouflage colour).

- Behaviour tree node editor
 - o An upgrade for the current behaviour tree designing process. It will be easier to design and debug behaviours. The tree itself would be visualized, rather than
- More sound effects
- Placing landmines
 - o This would improve the combat dynamics, as both the player and the AI can decide to place landmines in a strategic location, which enables more complex combat scenarios.
- A dynamic day-night cycle.
- Other types of ammo
 - o Having several types of ammo (which was the plan at first but has been scrapped due to time constraints) would make the combat more dynamic, forcing the player to choose about the way they approach a combat scenario. For example, if you inspect an enemy tank and see it has exceptionally high armor properties but relatively low health properties, you would want to choose an armor-piercing shell round so it's a way around the heavy armor.
- More levels
 - o Combined with the quest system, this would make the game more immersive, because there could be a story-telling aspect if there were several levels.
- Quest system
 - o A feature that allows for an actual "purpose" in the game, rather than just driving around and killing enemies.

Self reflection

In retrospect, there have been plenty of mistakes I made, from which I can learn/have learnt from. Aspects in both development and organization areas could be improved. To name a few points of attention:

- Find QA from the industry instead of QA from teachers. Preferably even from industry developers working on similar projects, in my case a tank simulator/game.

- Lower the bar in terms of features. It's hard for me personally to find a balance between my ambitions (what's good enough for me to be satisfied?) vs a realistic workload (what can I do realistically, in a given timeframe with the knowledge I have at that moment?). This means it's also a dangerous to set the bar *too* low. However, when that's the case, I can always decide to ramp up the challenge a bit and add some features/go more in-depth on a certain task.
- Make the decision to scrap mechanics earlier on in the development cycle. If a feature isn't working the way I expect and want it to or too many hours will be/have been spent, I should revise my planning & ambitions.
- For future projects, I should try to spend less time at home alone, and more time with other people, possibly at school. During the lockdowns last year(s), I noticed my productivity went down fast, after a while, and now again during the minor. I did spend the occasional workday in voice chat with some friends, but it's not enough social contact to sustain myself.

Conclusion

When listening to the QA feedback and objectively looking at my work (trying to do so without thinking about the aspects that are missing/should be better), I can conclude I still made a good product. I learned a lot over the past 4.5 months, my AI development skills being one of the things I improved on. One step closer to a job in the industry.

Taking my self-reflection points into account, I can improve on myself during the next (large) project I'm going to work on, which will be IMT&S next semester.

Appendices

Behaviour tree

A way to design AI behaviour, using a tree-like structure. The tree is built from a set of different node types. There is one "root node", from which all other nodes are run. In my implementation, there are 3 different node types being used.

- Sequence node: all child nodes have to return SUCCESS, otherwise if any child returns FAILURE, the node fails and stops the checking the child nodes, returns FAILURE itself.
- Selector node: if any child node returns SUCCESS, the node stops looping through the children and returns SUCCESS itself.

Wheel collider

A special collider for vehicle wheels [2]. Wheel collider is used to model vehicle wheels. It simulates a spring and damper suspension setup, and uses a slip based tire friction model to calculate wheel contact forces.

Wheel's collision detection is performed by casting a ray from center downwards the local y-axis. The wheel has a radius and can extend downwards by suspensionDistance amount.

The wheel is controlled with motorTorque, brakeTorque and steerAngle properties.

Wheel collider computes friction separately from the rest of physics engine, using a slip based friction model. This allows for more realistic behaviour, but makes wheel colliders ignore standard PhysicMaterial settings. Simulation of different road materials is done by changing the forwardFriction and sidewaysFriction based on what material the wheel is hitting.

Mildots

A MIL-Dot reticle refers to a standard, specific pattern of duplex crosshair reticles with four small 0.25 mil diameter dots placed along each axis.

These dots are arranged to allow for range estimation. A trained user can measure the range to objects of known size, determine the size of objects at known distances, and compensate for both bullet drop and wind drifts at known ranges with a MIL-Dot reticle-equipped scope [3].

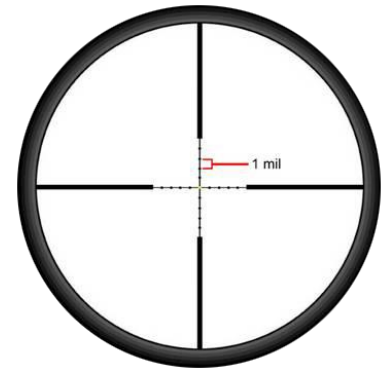


Figure 23 Mildots calculation example.
Source = [3]

Drag coefficient implementation

```
private void UpdateShellsVelocity()
{
    foreach (Shell currentShell in _firedShells)
    {
        double deceleration = GetDeceleration(currentShell.RB) * Time.deltaTime;
        currentShell.RB.velocity -= currentShell.RB.velocity.normalized *
(float)deceleration;
    }
}

private const float AIR_DENSITY = 1.2f;
private const float SHELL_FRONTAL_AREA = 0.035f;
private const float DRAG_COEFFICIENT = 0.1f;
```

```
private static double CalculateDragForce(Rigidbody rb) => DRAG_COEFFICIENT * 0.5f *
AIR_DENSITY * Mathf.Pow(rb.velocity.magnitude, 2) * SHELL_FRONTAL_AREA;

private double GetDeceleration(Rigidbody rb)
{
    double dragForce = CalculateDragForce(rb);
    Vector3 currentVelocity = rb.velocity;
    Vector3 inverseVelocity = currentVelocity.normalized * -1 * (float)dragForce;
    double deceleration = currentVelocity.magnitude - inverseVelocity.magnitude;

    deceleration /= rb.mass;

    return deceleration;
}
```

My implementation of the drag coefficient formula. For every shell that has been fired and is still active, the drag force is calculated and applied to the current velocity. A mechanical engineering student/friend explained the formula as I am not very skilled in physics/math so I asked him for help, after which I did my own research and attempt at implementing it. Eventually he helped me a bit with the implementation.

Bibliography

- [1] The Engineering ToolBox, "Drag Coefficient," [Online]. Available: https://www.engineeringtoolbox.com/drag-coefficient-d_627.html.
- [2] Unity, "Wheel Collider component reference," Unity, 23 06 2023. [Online]. Available: <https://docs.unity3d.com/Manual/class-WheelCollider.html>. [Accessed 29 06 2023].

- [3] Trijicon, "Advanced Mil-Dot: Estimating Distance Using Your Scope," 28 08 2019. [Online]. Available: <https://www.trijicon.com/community/post/how-to-use-trijicon-accupoint-mil-dot-riflescopes#:~:text=A%20MIL%2DDot%20reticle%20refers,to%20allow%20for%20range%20estimation..>
- [4] C. Simpson, "Behavior trees for AI: How they work," 18 07 2014. [Online]. Available: <https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work>. [Accessed 05 2023].
- [5] S. Castro, "Introduction to behavior trees," 17 08 2021. [Online]. Available: <https://robohub.org/introduction-to-behavior-trees/>. [Accessed 05 2023].
- [6] M. Pêcheux, "How to create a simple behaviour tree in Unity/C#," 2 11 2021. [Online]. Available: <https://medium.com/geekculture/how-to-create-a-simple-behaviour-tree-in-unity-c-3964c84c060e>. [Accessed 5 2023].

Credits for assets

Synty Studios military pack: <https://syntystore.com/collections/frontpage/products/polygon-military-pack>

Synty Studios Battle Royale pack: <https://syntystore.com/collections/frontpage/products/polygon-battle-royale-pack>

Typewriter font: <https://www.dafont.com/jmh-typewriter.font?text=500m&back=theme>

Crosshair: <https://void1gaming.itch.io/free-mega-crosshairs-pack/download/eyJleHBpcmVzIjoxNjc3MjM1MzgyLCJpZCI6OTM1OTI3fQ%3d%3d.3JodmL6zWeY91zoMy9ECRUvNEWo%3d>

Target indicator icon: [Frame icons created by Royyan Wijaya - Flaticon](https://www.flaticon.com/free-icons/frame "frame icons")

Repair icon: [Gear icons created by Freepik - Flaticon](https://www.flaticon.com/free-icons/gear "gear icons")

Target icon: [Sniper icons created by Freepik - Flaticon](https://www.flaticon.com/free-icons/sniper "sniper icons")

Audio: humble bundle – creators are "Dark Fantasy Studio".