FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2023 Open / ~~Confidential~~
Authors: Jesper Søberg, Magnus Tysdal, Simen Graue Kase	
Programme coordinator: Erlend Tøssebro Supervisor(s): Erlend Tøssebro & Naeem Khademi	
Title of Bachelor's Thesis: Further Development of 3D Self-Rescue Game for Tunnel Fire	
Credits: 20	
Keywords: Game development, emergency response training, tunnels, traffic simulation, procedural 3D-modelling, data processing	Pages: *89* + attachments Stavanger, 15th of May / 2023 date/year

Further Development of 3D Self-Rescue Game for Tunnel Fire

by

Jesper Søberg
Magnus Tysdal
Simen Graue Kase

Supervisors: Erlend Tøssebro & Naeem Khademi

A thesis submitted in partial fulfillment for the bachelor's degree of Computer Science

at the

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

May 2023

Abstract

This thesis provides a system that grants access to the data in Nasjonal Vegdatabank (NVDB) for a game simulation of a tunnel. The primary objectives were to create a more realistic game experience, by having the tunnel resemble its physical counterpart, enabling the player to drive vehicles, and creating a realistic traffic simulation in the tunnel.

To achieve the first objective, a system was implemented that allows the game to gather data from NVDB. The second objective was achieved by allowing the player to interact with the car by entering and exiting, and using user inputs to drive the car. The third objective was accomplished by retrieving data from NVDB to simulate traffic that adapts to real traffic data.

All objectives were achieved with sustainability in mind. The project is built to allow for easy updates and changes in the future, rather than be replaced later in the project by new implementations.

Preface

This bachelor thesis was written at the Department of Electrical Engineering and Computer Science at the University of Stavanger.

We would like to give a special thanks to our supervisors Erlend Tøssebro & Naeem Khademi, Associate Professors at UiS, who have given us valuable guidance and contributed with insightful and nuanced perspectives.

We would also like to thank Benjamin Mydland and Ove Oftedal for giving us insight into the project and helping us with problems we had during development.

Contents

List of Figures

Listings

Acronyms

API Application Programming Interface

CFD Computational Fluid Dynamics

FDS Fire Dynamics Simulator

GUI Graphical User Interface

ITSRG Information and Communication Technology(ICT)-based Tunnel Safety Research Group

LFS Large File Storage

NPC Non-Playable Character

NPRA Norwegian Public Roads Administration

NVDB Nasjonal Vegdatabank

RPM Rotations per minute

UI User Interface

UiS Universitet i Stavanger

VCS Version Control Systems

VR Virtual Reality

ÅDT Årsdøgntrafikk

Norwegian Glossary

Norwegian:	English:
Assosierte Signalpunkt	Associated Signal Point
Assosierte Skiltpunkt	Associated Signpost
egendefinert	custom defined
egenskap	property
egenskaper	properties
egenskapstype	property type
enhet	unit
fartsgrense	speed limit
innhold	content
lengde	length
liste	list
navn	name
overlapp	overlap
relasjon	relation
relasjoner	relations
sideposisjon	side position
signalpunkt	signal point
skiltplate	sign plate
skiltpunkt	sign point
slutt	end
start	start

stedfesting	location fixing
stedfestinger	location fixings
trafikkmengde	traffic volume
trafikkmengder	traffic volumes
tunnelportal	tunnel portal
type	type
vann- og frostsikring	water and frost protection
veglenke	road link
veglenker	road links
vegobjekter	road objects
verdi	value
ÅDT, andel lange kjøretøy	ÅDT, share of long vehicles

1 Introduction

1.1 Motivation

Norway has started several large scale tunnel projects across the country, which is great for many reasons, such as not having to rely on ferries for transport. However, these tunnels also expose travelers to new security risks, with the main one being fires developing inside of a tunnel. In the event of an accident, it is important that individuals who are caught in such an accident know what actions to take to rescue themselves, as the rescue personnel may not arrive in time.

Despite enormous amounts of effort to enhance tunnel safety and decrease the risk of people being injured, such as adding lighting and alarms to guide people to safety, the danger still persists. Especially tunnels such as Ryfast (the longest sub-sea tunnel for cars in the world) or Rogfast (which will claim the title in the future) are very cost-inefficient to close down for safety drills, not to mention that doing so would significantly limit the flow of traffic or even stop it completely.

For these reasons and many others, Universitet i Stavanger (UiS) (University of Stavanger) established the Information and Communication Technology(ICT)-based Tunnel Safety Research Group (ITSRG). Their goal is to make digital versions of the tunnels so that the public and professionals can prepare themselves for emergency scenarios.

1.2 Objective

The aim of this thesis was to further develop the 3D self rescue game project, building upon the work done by previous theses. The focus was on improving existing functionalities and introducing new implementations. Specifically, the objective was to create a solid foundation within simple tunnels, which can be built upon later for more complex tunnels and game logic.

To achieve this, three concrete objectives were set:

- To have the tunnel more closely match its physical counterpart.
- Allowing the player to drive a car, so as to better participate in traffic.
- Simulating traffic within the tunnel, meaning the player will have to take their surroundings into account.

All of these objectives had an overarching objective of focusing on future oriented and sustainable code. This is now the fifth thesis in a chain of theses related to this project. It is therefore important to ensure the code is easy to understand, change and build upon seeing as the project is likely to continue on in the following years. This has been achieved through restructuring and decoupling of old code, whilst applying similar principles to the new features.

Overall, the main goal was to bring the game closer to a finished and useful product.

1.3 Thesis Outline

Chapter 2 - Background

Literature review of key concepts needed to understand the project and related works.

Chapter 3 - Updates

Updates that were made to previous implementations.

Chapter 4 - Implementations

New features implemented in this thesis.

Chapter 5 - Evaluation

Evaluation of the new implementations and their limitations.

Chapter 6 - Conclusion

Conclusion and recommended future works.

2 Background

2.1 Work Done at UiS

This is a further development project, and as such, several previous works have been done at UiS, all focusing on different aspects of the project. The following sections will present a brief timeline of these theses and summarize what was or was not used in this thesis and why.

2.1.1 Procedural 3D Modeling of Road Tunnels: a Norwegian Use-case

The bachelor thesis [21] where it all began. Written in 2020, it managed to develop a system that could procedurally generate a 3D model of any given road tunnel in Norway using a software known as Autodesk Maya. The purpose of this was to be able to do safety training inside said tunnel. Data was retrieved from NVDB, a public database of Norway's tunnels created by the Norwegian Public Roads Administration (NPRA) to create 3D models of the given tunnel. The application is capable of modelling both lone tunnel tubes as well as some larger tunnels consisting of multiple tubes, although the functionality of the latter option was limited. This thesis therefore primarily builds upon the tunnel tubes.

The thesis also began working on a method of placing objects inside the tunnel, but this too was not completed. As such this thesis will further explore this topic as well.

Finally, the thesis developed a VR demo allowing the user to move around inside the tunnel model. The team decided to focus on the game itself and has therefore not focused on developing the VR functionality any further.

2.1.2 3D Self-Rescue Game for Tunnel Fire

A bachelor thesis [9] written in 2021 used the tunnels generated by the previous thesis (2.1.1) to create a self-rescue game. In short, the player would appear next to a burning car inside a tunnel and be tasked with finding the nearest exit. The game included a fire extinguisher that could be used to extinguish the car fire as well as some support for multiple difficulty settings. Adjusting the difficulty would in turn adjust the rate at which smoke will kill the player or how many vehicles appear inside the tunnel. The thesis laid the groundwork for this thesis, hence most of its functionality is still present although some it has has been disabled or removed for reasons which will be discussed.

2.1.3 Digital Tunnel Twin Using Procedurally Made 3D Models

A master thesis [18] also written in 2021. The goal was to be able to create digital twins [10] of the tunnels generated within the original thesis (2.1.1). Creating a digital twin would mean that it would listen to changes made in the data for the tunnels and update the models accordingly. It may also be able to gather data for things such as temperature or climate to mimic the physical tunnel as closely as possible.

Seeing as there were two theses written in 2021, they are both missing functionality implemented in the other. The Digital Tunnel Twin thesis did not directly work on a game and has therefore unfortunately been mostly left behind by later theses.

2.1.4 CoolEngine : Simulating Realistic Fire and Smoke in a Unity3D-based Tunnel Fire Rescue Game

A bachelor thesis [17] with the main goal of creating a realistic simulation of a tunnel fire. The team used a simpler Computational Fluid Dynamics (CFD) software known as Fire Dynamics Simulator (FDS), and used Plot3D to get the output from the FDS simulation to Unity. From there Unity's particle system was used to implement the fire and smoke within Unity. The researches decided to build upon the preceding bachelor thesis (2.1.2) rather than the master thesis (2.1.3). As such, it also managed to expand the game with some smaller improvements such as a traffic implementation and the ability to create set scenarios.

Seeing as it was the newest version of the self-rescue game, this thesis will continue developing the game using the coolengine as its foundation. As will be discussed however, many features had to be changed or remade. Although impressive, the fire simulation has not seen much use as it currently requires too much processing power to justify its benefits.

2.1.5 The Current Thesis

In summary, this thesis consists of two separate applications: The procedural modelling developed in 2020, and the rescue game developed in 2021. The latter was later expanded with additional functionality in 2022. This thesis seeks to further expand both of these applications to develop the game further.

2.2 NVDB

NVDB is a digital database system, which stores detailed information about Norway's road network [42], including information about tunnels. The database is used actively by the NPRA for Norway's road management, but it has also been made available to the public.

One can interact with NVDB using the Vegkart website (vegkart.atlas.vegvesen.no) or by interacting with the Application Programming Interface (API) directly using HTTP requests. As an example, the following URL will retrieve (the ID of) every tunnel in NVDB:

https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/581

Developers can interact with this database to retrieve data about different tunnels. This data includes the chosen tunnel's geometry, properties and more. The data retrieved from the database presents a lot of opportunities for this project, as will later be explored in this thesis.

2.2.1 NVDBv3 Data Structure

Working with any code related to NVDB will require a basic understanding of how the returned data is structured. Note that the structure described in this section is the third and most recent version of the API. The second version will be briefly discussed in section 3.1.

Figure 1 below shows the basic structure of the data that is returned when requesting data for a tunnel tube.


```
{
   "id":489792809,
   "href":"https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/67/489792809/2",
   "metadata":{ },
   "egenskaper":[ ],
   "geometri":{ },
   "lokasjon":{ },
   "vegsegmenter":[ ],
   "relasjoner":{ }
}
```


Figure 1: The basic structure of data returned from NVDB

This contains some simple information such as the ID of the object or the URL used to retrieve the data. Of particular interest is the field "properties (egenskaper)". This field contains a long list of "egenskaper", each with their own "name (navn)" and "value (verdi)" keys. Although this may bring a dictionary data structure to mind, it is important to understand that "egenskaper" is not a dictionary, but rather a list where each "property (egenskap)" is a dictionary. Said dictionaries contain the keys "navn" and "verdi" among some other keys. Some "egenskaper" are shown in figure 2 for demonstration purposes.


```
{
   "id":8356,
   "navn":"\u00c5pnings\u00e5r",
   "egenskapstype":"Heltall",
   "datatype":"Tall",
   "verdi":2012
},
{
   "id":1317,
   "navn":"Lengde",
   "egenskapstype":"Flyttall",
   "datatype":"Tall",
   "verdi":295.0,
   "enhet":{
      "id":1,
      "navn":"Meter",
      "kortnavn":"m"
   }
},
```


Figure 2: Two "Egenskaper" of a tunnel tube

Note that Norwegian characters are not encoded correctly. What should have been "åpningsår" has instead become "\u00c5pnings\u00e5r". Thankfully this has not been a problem for the project as the data only appears this way when saved to the file system. Within code, it will be encoded correctly. Nevertheless, the issue is mentioned here as some of the older code appears to have struggled with it and implemented some UTF-8 conversion to avoid it.

To retrieve the desired data for a tunnel, one would usually have to loop through the list of "egenskaper" to find the correct "navn". The desired data will then usually be found within the "verdi" field. For example, the length of the tunnel tube in figure 2 is 295 as can be seen by the "verdi" of the "egenskap" where "navn" is "length (lengde)".

This method will not always be sufficient. For example, one would have to use the "unit (enhet)" field to determine the unit in which the length of the tunnel tube has been measured. In other words, there will be some "egenskaper" with additional fields and useful data outside of the "verdi".

Finally, as seen in figure 1, there is a field called "relations (relasjoner)". This contains all parent and child objects of the data. For a tunnel tube, the parent object will be a tunnel whilst the child objects may include objects such as traffic signs or emergency stations. There is however little data available about these objects as seen in figure 3.


```
"relasjoner":{
   "barn":[
      {
         "listeid":220201,
         "id":200201,
         "type":{
            "id":69,
            "navn":"Tunnelportal"
         },
         "vegobjekter":[
            376392270,
            376392271
         ]
      },
      {
         "listeid":220205,
         "id":200205,
         "type":{
            "id":70,
            "navn":"Vann- og frostsikring"
         },
         "vegobjekter":[
            657578358
         ]
```


Figure 3: Two example relations. Contain little information beyond the type of object and their IDs

The fields of interest here are "type (type)" and "road objects (vegobjekter)". "type" identifies the type of object in question whilst "vegobjekter" contains a list of all such objects within the tunnel tube.

In the case of the example above, the tunnel tube contains two "tunnel portal (tunnelportal)" objects with IDs 376392270 and 376392271. Furthermore it also contains one "water and frost protection (vann- og frostsikring)" object with ID 657578358. These IDs may be used to request additional information about the corresponding object, as will be explored in section 4.3.4

2.3 Autodesk Maya

Autodesk Maya, often called Maya, is a professional 3D software for creating realistic characters and blockbuster-worthy effects [2]. It offers various tools and features that allow users to model, animate, and render 3D scenes with realistic textures, lighting, and dynamic simulations. It is widely used in the entertainment industry and is considered one of the most powerful and comprehensive 3D animation softwares available[3].

Maya is a commercial product, but includes a student and educators licence deal. The licence is valid for one year, and can be renewed as long as one is eligible.

It was the chosen 3D-modelling software for generating road tunnels for a previous thesis (2.1.1). Thus this thesis continues to use Maya.

2.4 Unity

Unity is the chosen game engine for this project. This was chosen by a previous group (2.1.1), and the team did not have any reason to change engine now. The following sections will briefly explain some of the basics of Unity and how the team has been working with it.

2.4.1 Unity UI

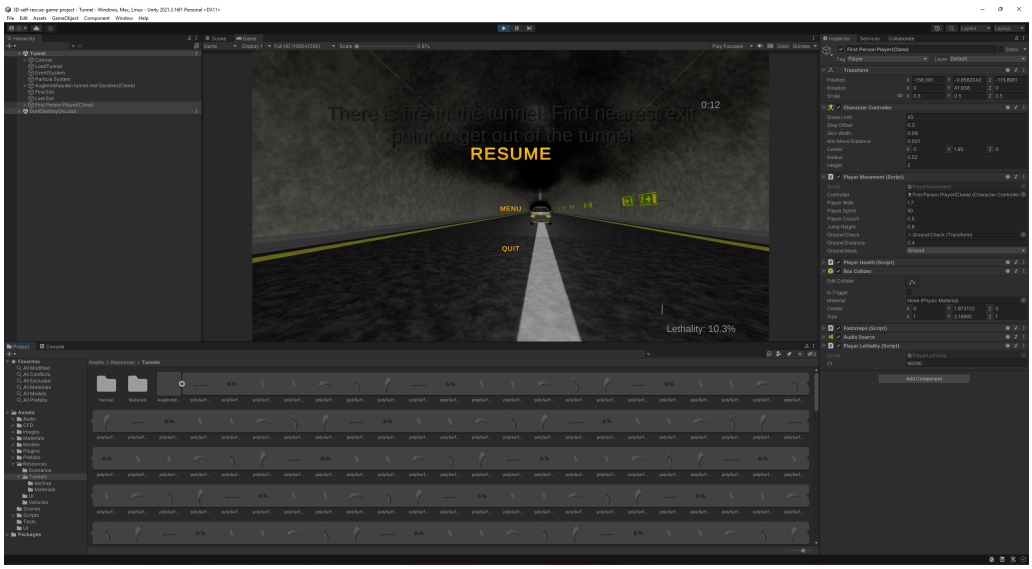


Figure 4: A screenshot of the Unity UI. It shows the hierarchy on the left, the game view in the middle, the inspector on the right, and the project overview at the bottom

The User Interface (UI) can be divided into four important sections. In reference to figure 4 the left pane contains the hierarchy. This displays the gameObjects present in the current scene and their relationship to one another. Clicking on an object here will display more detailed information about it in the inspector.

The inspector is the pane on the right. It shows all of the components that make up the object. These components will be further discussed in section 2.4.2. The inspector is very useful during testing as it allows for making temporary changes to variables whilst playing the game. Thus letting one quickly examine the impact of these changes.

The middle pane is the game view where one can play the game. Note that whenever the game is using the mouse for inputs, one will have to press the esc key to move the mouse outside the game. The scene tab should also be noted. This allows one to move freely around the scene, clicking on objects of note to see more details in the inspector. When moving around the scene, one can hold down right click; whilst doing so, the mouse may be moved to look around and the wasd keys may be used to move. Furthermore, one may use the q and e keys to move up or down.

Finally, the bottom pane shows an overview of all the assets, scripts and so on within the project. This comes in handy when you already know what you want to look at without having to play the game and find it within the hierarchy or scene. This pane also has a tab for the console. In addition to Unity's various error messages and warnings, you can also create your own logs for the console [24].

2.4.2 GameObjects

Everything within Unity is a GameObject. By default, they all have a transform component. This allows you to manipulate the object's position, rotation and scale. You may then add components to further expand the logic of the object. Some noteworthy components are listed below.

- **Script:** Allows you to write your own scripts in C# and attaching them to one or more objects, effectively making this the most flexible component available.
- **Rigidbody:** Makes the object susceptible to Unity's physics engine. The object will hence have a mass and can be affected by gravity.
- **Collider:** There are many different colliders such as BoxCollider and SphereCollider to name a few. They are all used to check whether two objects collide or overlap. This will often be combined with the rigidbody to let Unity handle realistic collisions between physical objects.

2.4.3 Prefabs

Prefabs in Unity is a fully configured gameObject that is saved in the project for reuse [12]. This makes them useful as they can be shared easily by different scenes, without having to be configured again. The main benefit of prefabs, is that the prefabs in different scenes work like linked copies of the original asset, meaning that a change in the original prefab will be propagated to all other instances of the prefab. A prefab can also be edited on a per-object basis, where a single instance of the prefab is changed for a specific purpose, instead of all instances being changed.

Figure 5: A screenshot of some Unity prefabs. Note the .prefab asset type at the bottom

2.4.4 MonoBehavior

When creating a new C# script from the Unity project window, it will automatically have the MonoBehaviour class included. Every Unity script has the MonoBehaviour class by default.

The class is needed to be able to attach the script to a GameObject in Unity's editor. Furthermore, it enables the script to access various Unity methods such as Start() and Update() which will be explored in section 2.4.5. There is a plethora of other methods usable with MonoBehaviour that one may find within the documentation [27].

2.4.5 Important Methods

Unity has some important methods built in which have been used many times throughout the code. Some of them are briefly explained below.

- **Awake:** This method is called by Unity when the object containing the script is first initialized.
- **Start:** This method is quite similar to awake. It will be called once, but not before the script is enabled and always after the call to Awake().
- **Update:** The call to Update() happens once for every single frame, making this method useful for anything that might need to change between frames. The method will often check the value of Time.deltaTime which returns the elapsed time since the previous frame, and use it to scale different values. If for example a force is being applied to an object, it should be scaled according to Time.deltaTime to prevent variations in frame rate from applying the same amount of force over varying time intervals.
- **FixedUpdate:** This should be used whenever Unity's physics engine is involved. It functions like the Update() method, but rather than being called every frame, it is called at fixed intervals. These intervals correspond to whenever Unity's physics engine is making an update. This fixed interval can also be accessed and altered using Time.fixedDeltaTime.

It is important to note when the different methods are called. A detailed flowchart for the Script lifecycle can be found in Unity's documentation [28].

2.4.6 Raycast

Raycast [5] is a Unity method that draws a vector originating from a certain point. Within this project it is used for enabling the player gameObject to interact with other gameObjects, the car being one example. One may also use it to shoot an object with a weapon or check what the player is standing on.

A ray is created in the scene using the Unity physics engine. If the ray hits a gameObject the function will return a boolean value. Information about the hit gameObject can be gathered such as position, its tag and the object's transform. It can also be used to trigger certain scripts such as the interactable objects in the game.

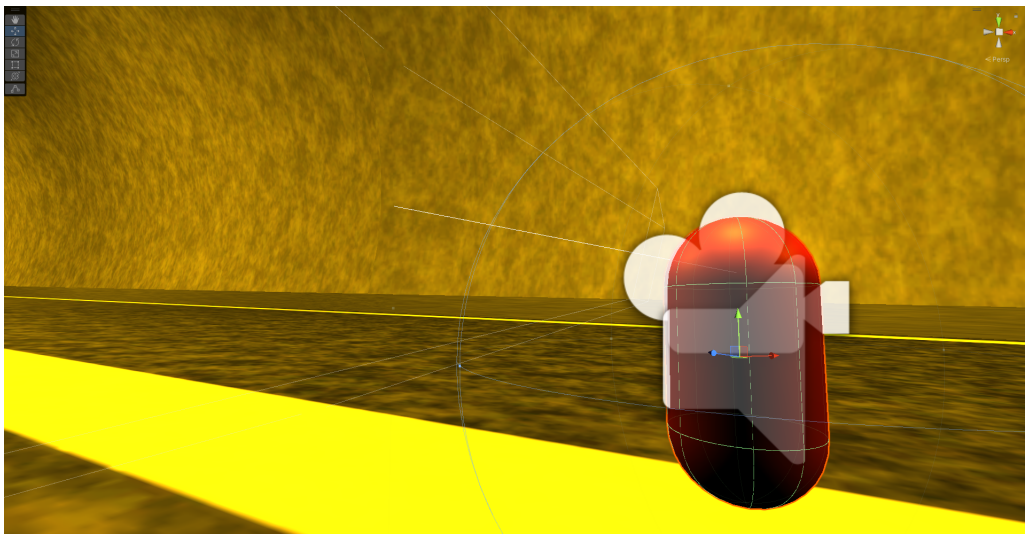


Figure 6: Raycast drawn from the player cameras origin

One thing of note is that the raycast is not visible by default. In order to be able to see the raycast a Debug.DrawRay() for the raycast needs to be added in order to see it.

```
1 Ray ray = new Ray(cameraTransform.position, cameraTransform.forward);
2 Debug.DrawRay(ray.origin, ray.direction* maxDistance);
```

Listing 1: Raycast Debug.DrawRay() code from Cameracontroller.cs

The ray is being cast in line 1. Line 2 draws the raycast with the length specified by the variable maxDistance, making it visible in the scene viewer.

2.4.7 Static Keyword

Several methods use the "static" keyword [15]. It is a modifier used to declare for example a field that will be shared between all instances of a class. The modifier can also be added to classes, interfaces, properties, and so on. The main reason it has been used so much is to have a static and common instance of certain fields or methods, rather than instantiating them for every gameObject. It has also commonly been used for classes which should never have more than one instance, thus avoiding having to instantiate it.

2.4.8 SerializeField

If it is desirable to make a private variable visible in the inspector, one may add the [SerializeField] decorator to force Unity to make it visible in the inspector. With the variable visible one is able to drag and drop gameObjects in the fields, thus adding a reference to them in the script.

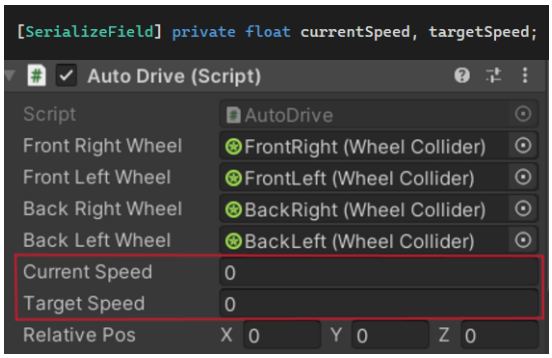


Figure 7: SerializeField defined on top in C#; the fields in the inspector at the bottom in Unity

It is also possible to make a variable visible with the public keyword, however one might not always want to do that if variable access is of concern [13]. Then it might be preferable to use "[SerializeField] private" for the variable instead.

2.4.9 Visual Studio

Unity code is written in C#. The team has therefore been using the code editor Visual Studio for writing code, since it offers great support for C#. More importantly, it also has packages for support with Unity. Meaning it can recognise various Unity types and classes such as gameObjects, whilst also allowing Unity to easily open any scripts or projects in Visual Studio.

2.4.10 Debugger

Unity allows you to attach a Visual Studio debugger to a project. This is a great tool for inspecting the flow of the program as well as when or why things happen. This section will therefore explain how to attach the debugger.

Switch to debug mode in Unity by clicking on the icon in the bottom right as highlighted in figure 8.

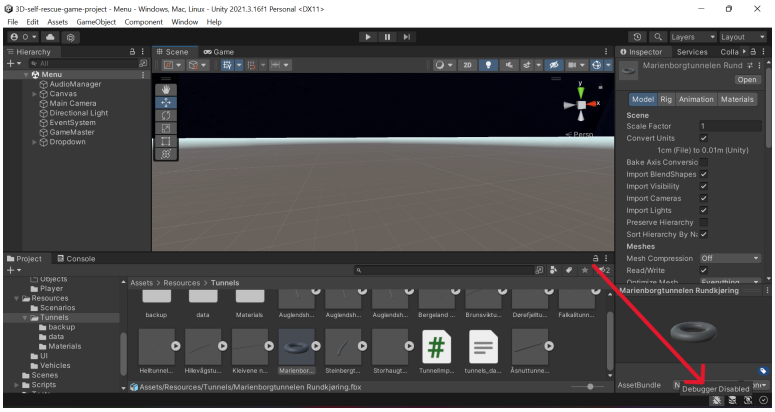


Figure 8: Switch to debug mode using the icon in the bottom right. The debugger is currently disabled

Once debug mode has been enabled and Unity has recompiled everything, Visual Studio may be opened. From there select "Debug", then "Attach Unity Debugger" from the top. Doing so will open a menu to choose a project to attach to. After selecting your current project, the debugger should be ready as indicated by the aforementioned icon in figure 8 turning blue.

2.4.11 Unity Teams

Unity Teams is a collaborative tool designed for teams working on Unity projects. Unity Teams allows team members to save, share and sync project files, track changes and collaborate in real-time. It should be noted that Unity Teams was not used in this thesis, previous theses have utilized this feature, meaning some traces of it remain within the file system of the project, hence why it is explained here.

Unity Teams makes use of Version Control Systems (VCS) and Build Automation to allow for collaboration. Unity Version Control tracks ever change made to the project and these changes can be reversed, providing enhanced error control [34]. This also ensures that every team member is working on the latest version of the project, and fixes any compatibility issues. Unity Build Automation, previously called Cloud Build, enables automation of the process of building, testing and deploying Unity projects to the cloud [23]. The cloud is a virtual environment that builds the game in that environment, and when the build is complete, the team gets notified that its ready to use.

2.5 Git LFS

The project directories are being stored on GitHub. Some files however may become too big for GitHub storage to be ideal. This is especially true for games that are storing audio files or large prefabs. As such, the project has implemented Git Large File Storage (LFS).

Git LFS allows Git to track larger files. This is done by storing the actual file on a different server, whilst only storing a pointer to that file on GitHub [6]. It should not require any further setup after cloning the project beyond running the following command:

```
git lfs pull
```

The command will use the pointers retrieved through a normal "git pull" and replace them with the actual file contents.

3 Updates

3.1 NVDBv2 to NVDBv3

The procedural modelling of tunnels was previously based on version 2 of the NVDB. NVDB has since been updated to version 3 and the previous version is no longer supported. Hence a small update to the code was necessary to accommodate these changes.

First of all, the URLs had to be updated to be able to retrieve any data. Furthermore, the format of the returned data has changed, meaning the code would often crash. As can be seen in figure 9, the data of a specific tunnel as retrieved from the NVDB is placed in a dictionary with various fields. Of particular interest is the "egenskaper" field. Inside this field is a list as shown in figure 10.

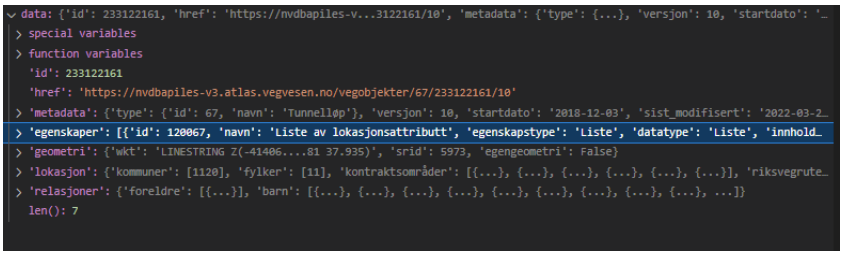


Figure 9: Data from NVDB with the "egenskaper" field highlighted

Figure 10: Contents of "egenskaper". In this case a list with 32 elements. (Only 10 elements shown.)

In version 2, every element in the list would contain a field known as "verdi". In version 3 this is no longer true. The code would therefore naïvely look for a field which may not exist, leading to a crash. A KeyError to be specific. This was handled by having the code check that the "verdi" field actually exists before trying to read its value.

Another change on top of this is that some of the elements in the "egenskaper" list (figure 10) are themselves a list. This is indicated by their "property type (egenskapstype)" having the value "list (liste)". In this case, they will have a field called "content (innhold)" containing another list. The elements within that list may contain a "verdi" field. This is shown in figure 11.

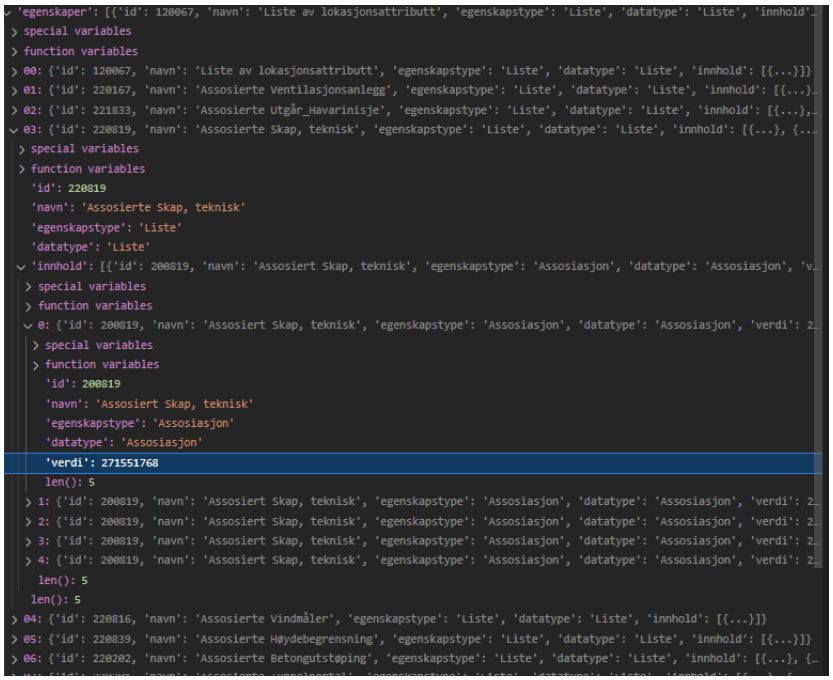


Figure 11: "Verdi" inside an element of the "innhold" list which itself is inside an element of the "egenskaper" list.

In summary, the code will loop through the "egenskaper" list and check if each element has a "verdi". If not it will check if the element has the "egenskapstype" "liste". If yes it loops through the "innhold" list and handles any "verdi" found inside its elements. The code is in other words written with previously encountered data structures in mind. If any unexpected "egenskapstype" were to show up, it could potentially break, or more likely be overlooked by the code. No such structure has been encountered during months of modelling however, so the code should work as long as NVDB does not receive another major update.

3.2 Fixing the GitHub Repository

Cloning the project to the computer and running the project went smoothly. However a problem arose when it came to adding, committing and pushing to GitHub. When writing the "git status" command, a massive output of files that had been either deleted or modified was printed to the terminal. The problem was so far out of control that the "git status" command was rendered useless. This due to there being no way of checking which files were staged for a commit, as the output was large enough to cap the terminal by a good margin.

The main culprits of these 1000+ file outputs were located within the Library folder, namely: Artifacts, PackageCache and ShaderCache. These files, amongst many other files which should never be on GitHub, had nevertheless been pushed to the repository by previous groups.

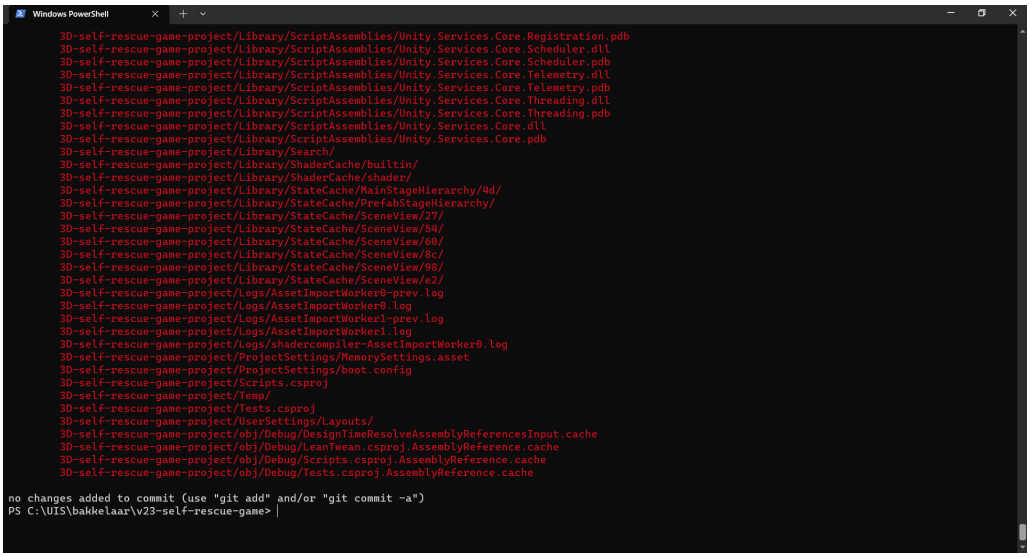


Figure 12: Git status output. Note the scrollbar as even scrolling to the very top will not show all the output

Any file causing trouble for GitHub has now been deleted from the repository and a .gitignore file has been added to ignore them in the future. The file is a standard Unity .gitignore file [1] with some additional items added by the team, including comments on why said items are ignored. The directories were massive, so although the .gitignore is believed to be complete, there are likely still some leftover files already on GitHub warranting a deletion.

What follows is a list of the more noteworthy directories which have been added to .gitignore and a brief explanation of why:

- Artifacts, PackageCache, ShaderCache: The reason these three directories in particular were a problem is that they would automatically generate and delete files as the game was running, thereby continuously generating more and more files for Git to watch. They should therefore never be pushed to GitHub again unless strictly necessary. Note however that they themselves are not ignored, but rather the entire Library directory. Library is auto generated by Unity and is therefore not needed on VCS.
- Collab, CFD: These contain code from previous projects that are no longer in use. They are however still present on GitHub. Collab is a directory used by Unity Collaborate (A previous version of Unity Teams (2.4.11)). Collaborate was used by a previous group (2.1.2) but is no longer being used in the current version. CFD is a directory used for more realistic fire simulation (2.1.4).
- Tunnels: Contains data and models of tunnels. Some have been pushed to GitHub so that one does not have to model a tunnel to play the game after cloning, but tunnels should generally not be on GitHub.
- Bee, Search: The purpose of these directories is not known. They have been added to .gitignore for the purpose of reducing output, but can be safely removed if needed.

For the sake of completeness, the following will detail what Unity directories and files *should* be on GitHub:

- Assets: Contains all assets used by the game, such as scripts or prefabs. Obviously useful to be able to share with the team.
- Packages: Tells Unity which packages are needed and their dependencies. Without the newest version of Packages, Unity will not be able to automatically resolve dependencies.
- ProjectSettings: Various settings for the project, including the Unity version used. Without this information, Unity may ask to update the project due to believing it is using an outdated version.
- .meta files: Whenever Unity compiles, it will create a .meta file corresponding to each asset file. The tunnels for instance, have both a .fbx file containing the actual model as well as a .fbx.meta file of the same name. These .meta files contain a GUID that Unity uses to locate files, meaning that without these .meta files on GitHub, Unity will not always be able to locate files on its own (if a prefab references a script for example). Note that all folders will have a .meta file as well.

Beyond what is described above, everything within 3D-self-rescue-game-project should be safe to ignore.

3.3 Removal of Burning Vehicle

In the original version of the game, the player spawned next to a burning vehicle in the middle of the road. The fire would grow larger and larger as the game went on thus generating more and more smoke. Coming in contact with this smoke is what would eventually kill the player.

Figure 13: A burning car

As the project went on, the authors found the burning vehicle implementation to be assuming too much about the final state of the project, as well as simply being implemented too early. It had no functionality other than standing in the way and killing the player. Until other objects are able to respond to the burning vehicle, there is not much reason to reimplement it. Therefore it is currently removed from the project.

3.4 Removal of LeanTween

LeanTween [19] is an asset from the Unity Asset Store used for simple animations. In previous projects it was used to add a short animation to text as it appears in the player's UI.

The issue with this was that it created numerous files and took a lot of space within the game directories (as well as GitHub). Considering these problems and the fact that the package was barely used, it was not worth keeping for some simple animations that could easily be recreated without LeanTween. Hence the decision was made to completely remove LeanTween from the project.

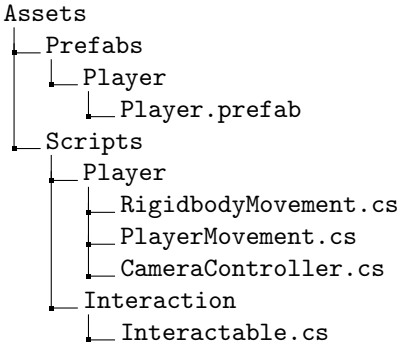
3.5 Restructuring of TScene

The original implementation of TScene had an abundance of responsibilities it should not have, leading to a lengthy and disorganized class. Thus changes had to be made, mostly restructuring involving moving methods out from TScene and into their own scripts. Afterwards improvements were made to the structure of the remaining code in the class, and all unused code was removed from the class.

The new TScene is intended to create the scene and help instantiate objects within the scene, rather than be responsible for any of the logic or interactions within the scene. It now functions similarly to a typical program.cs or app.py file, in that it mostly does the boring busywork of initialising classes, rather than any concrete tasks such as spreading fire. Furthermore, the new TScene will now only load the tunnel tube itself, rather than all of the objects inside of it such as signs and emergency doors. Ultimately these changes have resulted in a more efficient TScene capable of loading the tunnel scene faster than before.

3.6 Player Prefab

To start with the old player prefab worked fine. However it proved to be difficult to work with and numerous issues were discovered during development. The prefab was therefore ultimately replaced with a new prefab. All features of the original prefab have been carried over, and will be explained in the following sections. Below is an overview of the folders and scripts discussed in this section.



3.6.1 New Player Prefab Structure

The figure below shows the old player prefab with a red capsule body, a camera, and an audio source (the white speaker). Around the player is a blue ball indicating the range of the audio clip as well as some white lines extending from the player to indicate what is within view of the camera (and thus visible to the player).

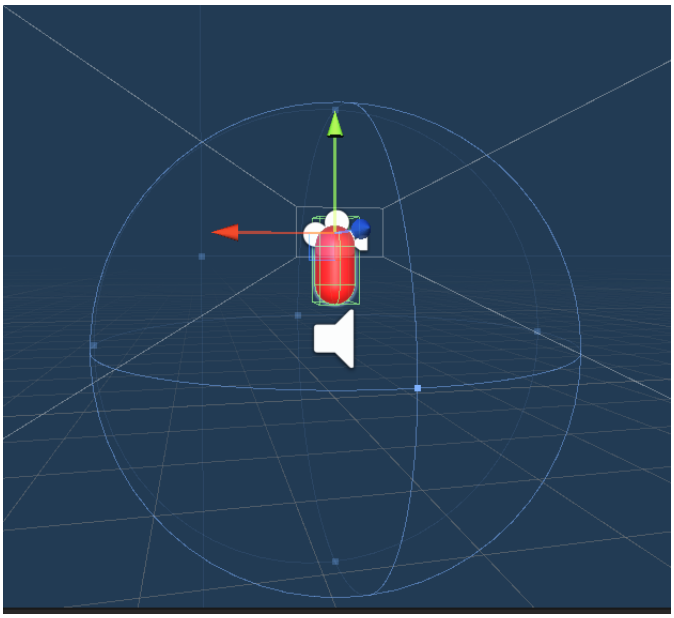


Figure 14: The old player prefab "First Person Player" as seen from the prefab viewer

The figure shows that the origin point for the player is not located in the center of the capsule body, but instead at the same position as the audio source (illustrated by the white speaker). This design caused several issues when dealing with the player, most notably when implementing the car interaction discussed in section 4.1.5. Consequently, the players origin point has been changed in this new iteration. The old player prefab hierarchy is shown in the figure below for reference.

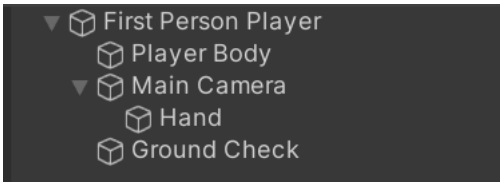


Figure 15: The hierarchy for the old player prefab

When inspecting the hierarchy, the presence of the "First Person Player" object took notice. This was the gameObject which contained most of the components that makes the player, such as scripts and a character controller. This adds an unnecessary level of complexity when dealing with the player prefab (additionally, no documentation was found that explained this design choice), therefore it has been removed to simplify the prefab. The figure below shows the new player prefab in the Unity editor.

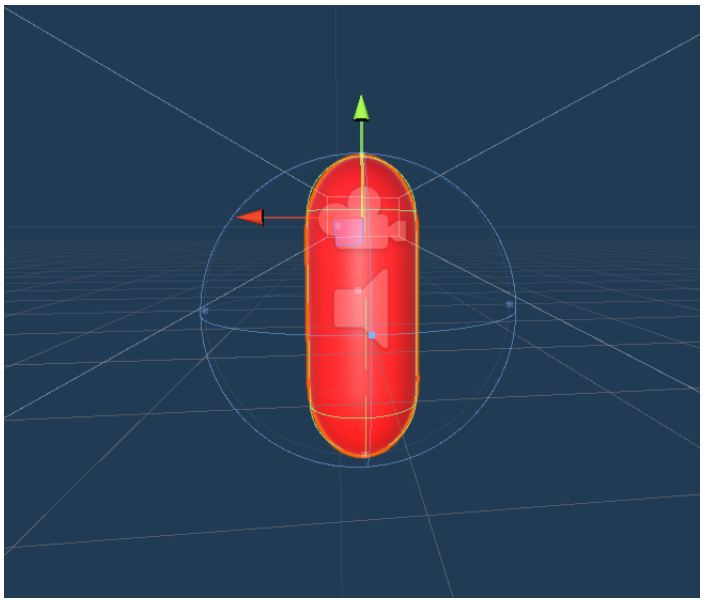


Figure 16: The new and more compact player prefab called "Player"

The new redesign for the player is more compact and easier to deal with. Additionally, placement of gameObjects such as the camera are more intuitive and manageable, since the origin is where you would expect it to be. The figure below shows the new player prefab hierarchy.

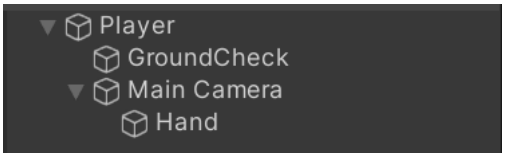


Figure 17: The hierarchy for the new player prefab

The major change to the prefab in terms of the hierarchy is the removal of the "First Person Player". Components like the scripts has been added to be a part of the player body now called "Player" and gameObjects like "Main Camera" are now a child of the "Player".

3.6.2 Player Interaction

The interaction logic is in the player camera and the script CameraController.cs within said camera. This script is responsible for drawing a Raycast, thus allowing for interaction between the player and interactable gameObjects. The sequence of events goes as follows:

1. The player presses the interact key, drawing a Raycast in the game.
2. If the Raycast hits an object that inherits from the Interactable class, the Interact() method of the class will be called upon. (This class is found in the Interactable.cs script.)
3. The Interact() method has been decorated with the keyword "virtual[16], allowing other classes inheriting the method to override it using the "override"[14] keyword. Doing so allows different

classes to run their own implementations of the Interact() method to better suit the needs of a specific gameObject.

The way the Raycast line knows it hit an interactable object, is if the gameObject's LayerMask [26] is set to "Interactable". The Raycast line will then try to run the Interact() method of the target hit.

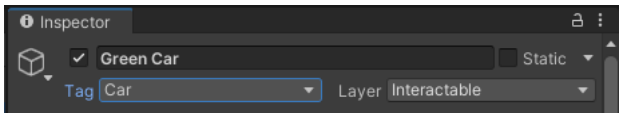


Figure 18: The tag and LayerMask can be changed in the gameObject's inspector

A gameObject's tag [33] is useful for collision scripts. Using the GameObject.FindWithTag(TagName) method, one is able to find the gameObject with the TagName supplied to the method.

In the hierarchy shown in figure 17, there is also a child object called "Hand". Originally, the purpose of the Hand object was to be able to pick up a fire extinguisher and use it to put out fires, although this feature was disabled in 2022 as explained in the report [17]. The hand gameObject is nevertheless kept in the new model should a reimplementation of the fire extinguishers or any other object pickups.

The way this feature worked, was that the player would look at it and press the interact key, causing the extinguisher gameObject to be placed within the Hand object. Once picked up, the player could press interact again to use the extinguisher. Pressing "G" would drop the extinguisher.

Finally, a note regarding the Update() method within CameraController.cs. It contains two if-statements: One that checks if the interact button has been pressed and another one that checks if the variable "shown" on line 3 in listing below is false.

```
1  void Update()
2  {
3      if (!shown)
4      {
5      }
6
7      if (Input.GetButtonDown("Interact"))
8      {
9      }
10 }
```

Listing 2: The two if statments in the Update() method from CameraController.cs

If the Draw.Ray is placed inside the "if (Input.GetButtonDown("Interact"))" statement on line 7 in listing above, the line drawn will only be seen for a split second. To see the ray constantly, simply place the Draw.Ray() inside the "if (!shown)" statement.

3.6.3 Rigidbody Movement

The old player prefab contained both a Rigidbody component as well as a Character Controller component. However, a single gameObject should not contain both of these components as explained

in the Unity documentation [22]. As per the documentation, a Character controller is generally only used for objects that do not need to be affected by Unity's physics engine. Seeing as the goal of the project is to make a realistic game, it would be advantageous to let Unity handle physics for gameObjects like the player, rather than writing the behavior all from scratch. The Rigidbody was therefore kept as it is needed to interact with the physics engine, whilst the Character Controller was removed, thus necessitating a new script to handle player movement.

RigidbodyMovment.cs is the script made to make the player move around using the Rigidbody instead of a character-controller-based movement script. This makes it easier for the player to interact with other gameObjects using the Unity physics. Interact in this case meaning the player physically moving around and pushing or being pushed by objects within the game.

Some changes have also been made to how the script is structured. The old movement script "PlayerMovement.cs" had all the movement logic inside the Update() method, making it difficult to read. Hence methods such as Jump() and MovePlayer() were made to move some lines of code out from the Update() method, thus separating logic and increasing readability.

```
1  void Update()
2  {
3      if (disableScript)
4      {
5          playerBody.velocity = Vector3.zero;
6          this.enabled = false;
7          return;
8      }
9      PlayerMovementInput = new Vector3(Input.GetAxis("Horizontal"), 0f, Input.
       GetAxis("Vertical"));
10
11     isGrounded = Physics.CheckSphere(groundCheck.position, 0.1f, groundMask);
12     isMovingForward = (Input.GetAxis("Vertical") > 0);
13
14     if (Input.GetButton("Crouch") & isGrounded)
15     {
16         playerBody.transform.localScale = new Vector3(1f, 0.5f, 1f);
17         MovePlayer(crouchSpeed);
18     }
19     else
20     {
21         playerBody.transform.localScale = new Vector3(1f, 1.4f, 1f);
22         MovePlayer((isSprinting ? sprintSpeed : walkSpeed));
23     }
24 }
```

Listing 3: The update method for RigidbodyMovement.cs

The way the movement works now is that the player movement is registered at line 9 in listing 3, then applies a Vector3 to the Rigidbody after calling the MovePlayer() method on lines 17 or 22. The speed is adjusted depending on whether the player is crouching, running or walking.

3.6.4 Disabling Player Movement

When the player goes inside the exit colliders (discussed in section 4.2.5), it takes a couple of seconds for the game to end. If the player wants, they could use this time to sprint out of the tunnel and fall

down indefinitely. The old method FreezeMovement() prevents this from happening and stops the player in their tracks by checking for a "freeze" variable in the update.

However it was noticed that the player inputs were registered before the boolean check of the "freeze" variable was done. Thus in RigidbodyMovement.cs this check was moved so that this is the first check that happens before any input is registered shown in line 3 in listing 3. Additionally it also disables the script completely to make Unity run less code, and the variable names have been changed to reflect this difference in behavior.

4 Implementations

Below is a directory tree providing an overview of the relevant directories for this chapter. The following sections will further specify which files are being discussed.

```
root
├── 3D-self-rescue-game-project
│   └── Assets
│       ├── Prefabs
│       │   ├── DrivableVehicles
│       │   └── Player
│       ├── Resources
│       │   ├── Vehicles
│       │   └── Tunnels
│       │       └── data
│       ├── Scenes
│       └── Scripts
│           ├── DrivableCar
│           ├── Player
│           ├── Traffic
│           └── Tunnel
│               └── Placement
└── proceduralmodeling
```

4.1 Drivable Cars

An early idea for this project was to create cars the player is able to drive themselves. The intention was to make the experience more realistic as in a real scenario one would have the option of driving rather than walking inside of a tunnel.

To achieve realistic car behavior, some functions were identified as the most important to simulate how a real car behaves. The following functionalities were identified as the most important to implement:

- The ability to drive a car
- The ability to enter a car
- The ability to exit a car

The implementations of these three functions will be discussed in the following sections. Below is an overview of the prefabs and scripts that have been created to make this possible.

```
Assets
├── Prefabs
│   └── Drivable Vehicles
│       ├── Drivable Green Car 1
│       ├── Drivable Silver Bus
│       └── Drivable_Purple_Truck
└── Scripts
    └── DrivableCar
```

```
CarController.cs
CarInteract.cs
CarWheel.cs
TruckController.cs
TruckInteract.cs
```

4.1.1 Prefab Structure

The first step to implementing a car would be to choose a car model to modify to become a drivable car for the player. Luckily, the project already contained car models downloaded from the Unity Asset Store [35] to expand upon. Of the models available, the model called "Car_1_Green" was chosen arbitrarily as it is a family car which was well suited for being the initial implementation before going forward with other types of vehicles like buses and trucks.

There are 6 main components that make up this prefab:

- The car model itself
- The Wheel Meshes (think of them as the visible wheel models)
- The Wheel Colliders (the invisible wheels that handle the physics)
- The camera present in the car
- The Scripts
- An empty gameObject called "Drivable Green Car Rigidbody"

The actual car model, called "Green Car" in the hierarchy, contains the Mesh Renders for the car model itself as well as the Mesh Colliders. The scripts are also attached to this object, making it the anchor for the prefab. Below is a figure of how the prefab looks and a figure showing how the prefab is structured in the hierarchy.

Figure 19: The car as seen from the prefab viewer

The figure below is the car prefab's hierarchy showing all the gameObjects contained in the prefab.

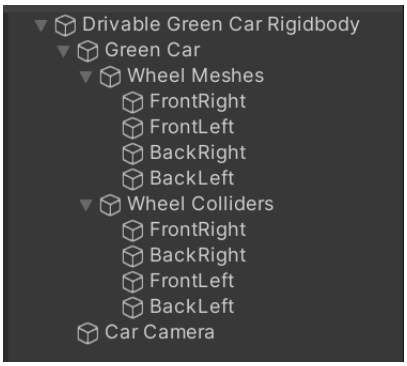


Figure 20: The hierarchy of the car prefab

The gameObjects "Wheel Meshes" and "Wheel Colliders" are simply empty gameObjects meant to group the Wheel Meshes and Wheel Colliders together in the hierarchy, thus making it more organised.

Additionally it may be confusing to see that there are two colliders for the wheels, those being the Mesh Colliders and Wheel Colliders. The difference is that Wheel Colliders despite their (official) name, are nothing more than some circular rays used for physical interactions such as rotating objects or in this case pushing a car in a certain direction. The Mesh Colliders are therefore created to mimic the complete 3D-shape of the visible wheel Mesh, and is what will be used to actually collide with other objects.

Both the CarController.cs and the CarInteract.cs scripts are attached to the "Green Car" gameObject in the prefab. The reason for attaching the scripts to "Green Car" and not "Drivable Green Car Rigidbody" is because the latter only exists to solve two problems: The main problem relates to the Wheel Collider rotation as will be discussed in section 4.1.2. The other problem is in regards to the scaling of the player after entering the car as will be discussed in section 4.2.4.

4.1.2 Wheel Colliders

In order to simulate the driving force of the car, Wheel Colliders were chosen as the driving force of the car. When used with the Unity physics engine, the colliders rotate and move the car forward in a realistic manner. The car can turn by adjusting the steeringAngle parameter of the Wheel Colliders, with their max turn angle set to 45 degrees to ensure the wheels don't rotate more than a real car would. The value of 45 degrees was arbitrarily chosen, and it may be changed later if needed. The appearance of a Wheel Collider is shown in the figure below.

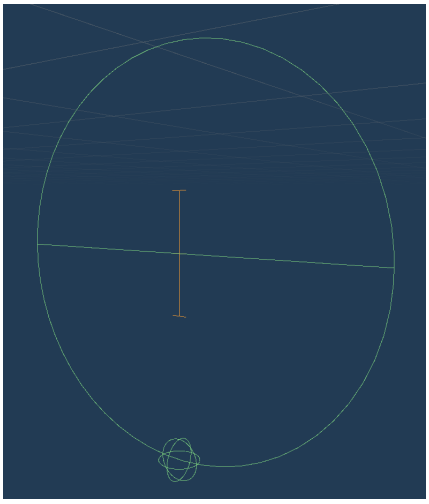


Figure 21: A Wheel Collider as it appears within the Unity editor. It is invisible outside the editor, unless selected in the Scene viewer

During the implementation of the colliders, it is important to keep in mind that the Wheel Colliders are fixed and cannot be rotated in the editor. They are always aligned and fixed to the Z axis, which can cause problems when implementing them, since they were not aligned correctly with the car. The reason for this design choice by Unity is unclear, but it is possible that it is for scripts to be able to adjust the steering, though this is not confirmed. The figure below depicts the problem.

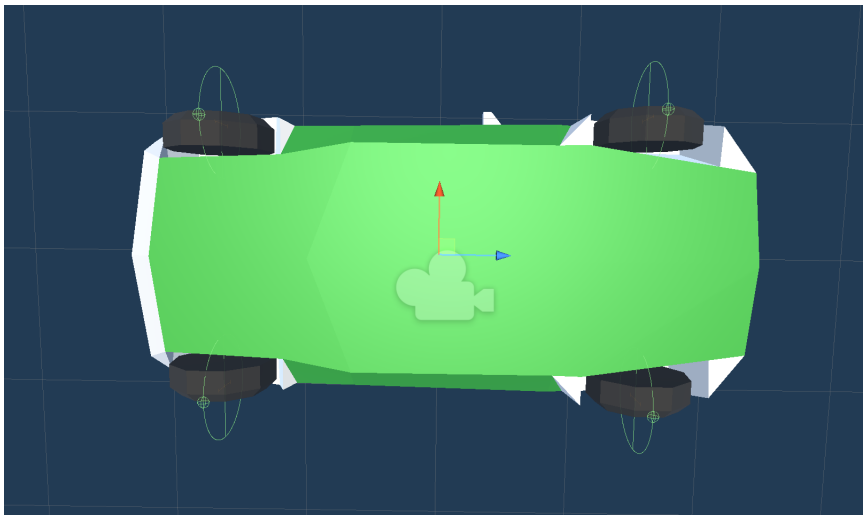


Figure 22: This is what it looks like when the Rigidbody is attached to "Green Car" instead of a gameObject like "Drivable Green Car Rigidbody"

To fix this, there are two possible solutions:

- Edit the model using software like Blender or Maya to align the model to the Z axis.
- As was done in this thesis, one can create an empty object and make the "Green Car" model a child of that empty object ("Drivable Green Car Rigidbody" in this case). "Green Car" may then be rotated to align the Wheel Colliders with the car model.

Attaching the Rigidbody to the "Green Car" gameObject will always result in a 90 degrees misalignment of the Wheel Colliders with the car model. This is due to the Wheel Colliders aligning themselves with the Rigidbody. Thus the Rigidbody needs to be attached to the empty gameObject "Drivable Green Car Rigidbody" instead. Then, the "Green Car" can be rotated -90 degrees to correctly align with the Wheel Colliders.

After the necessary changes mention to implement the Wheel Colliders, the car is still only able to move forwards and backwards. In order to enable steering, a Wheel Colliders property called SteerAngle was utilized. By setting this property, the Wheel Colliders can be made to turn, and as a result, the car turns as well.

maxSteerAngle is a variable that exists to prevent the Wheel Colliders from rotating indefinitely. The maxSteerAngle has been set to 45 as an arbitrary value. This goes for the values of motorForce, breakingForce and the acceleration variables in the script CarController.cs.

4.1.3 Driving the Car

CarController.cs is responsible for applying acceleration and brake force to the Wheel Colliders, which in turns makes the colliders push the car forward or slow down its speed.

```
1 private void Accelerate()
2 {
3     currentAcceleration = acceleration * verticalInput;
```

```
4       Wheel.motorTorque = currentAcceleration;
5
6       Wheel.brakeTorque = currentBreakForce;
7   }
```

Listing 4: Applying acceleration and break force to motorTorque and brakeTorque in CarController.cs

When player input is detected, the script multiplies an acceleration force with the player input for the front, and applies it to the Wheel Colliders' motorTorque property, which results in the car being pushed forward with the desired speed.

Similarly, when the player brakes, the currentBreakForce is applied to the brakeTorque property of the Wheel Collider. Breakforce is a bit different as it is first registered in the FixedUpdate() method as shown in listing 5 below.

```
private void FixedUpdate()
{
    if (Input.GetKey(KeyCode.Space))
        {
            currentBreakForce = breakingForce;
        } else
        {
            currentBreakForce = 0f;
        }
}
```

Listing 5: Setting the breakingFroce in CarController.cs

Within FixedUpdate() in CarController.cs, it is checked if the player is pressing the brake button. If it is pressed, then the currentBreakForce variable is set to be the value of breakingForce (shown on line 6 in listing 4), which is a constant statically set. On the contrary, when the break button is released, the currentBreakForce is set to 0, which means that the Wheel Colliders will no longer break. During the next FixedUpdate() call, the brakeTorque value will be set to be equal to currentBreakForce as shown on line 6 in listing 4.

The CarController.cs script also takes horizontal inputs into account, giving the player the ability to turn the car. The steering is applied through the Steer() method, as shown in listing 6.

```
private void Steer()
{
    steeringAngle = maxSteerAngle * horizontalInput;
    Wheel.steerAngle = steeringAngle;
}
```

Listing 6: Applying the steering angle to steeringAngle in CarController.cs

By registering the horizontal inputs from the player, it is possible to adjust the steerAngle values for the Wheels Colliders. This allows the Wheel Colliders to turn according to the player input and apply the driving force in the direction they are "looking". As a result, the car is pushed in the same direction as the Wheel Collider's orientation. Enabling the player to be able to turn left or right and change the car's trajectory.

Serialized fields are used to obtain a reference to the Wheel Colliders and allowing the scripts to manipulate the gameObjects as needed. The figure below shows how the serialized fields are used in

the CarController.cs script to obtain references for the Wheel Colliders as well as display the current break force, acceleration and steering angle which are useful when debugging.

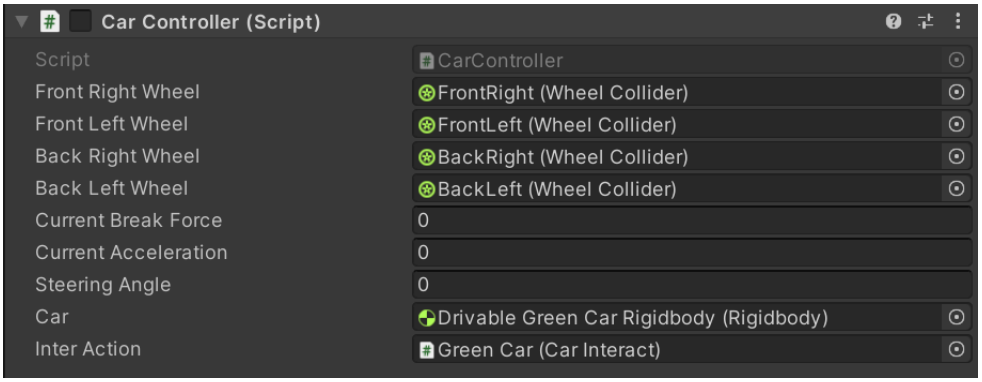


Figure 23: The inspector for Carcontroller.cs

If a reference to a script is needed in order to call a method of said script, serialized fields may be used to gain this reference through some simple drag-and-drop. However, since the serialized field expects a gameObject, the gameObject that contains said script needs to be added to this field. In the case of figure 23 the game object "Green Car" is inserted into the field. The reasoning behind this reference will be discussed in section 4.1.5 regarding the exit function.

4.1.4 Moving Car Wheels

To make the car movement look more realistic, it was sought-after to not only move the car forward, but also rotate the visible Wheel Meshes correctly like a real car would. To achieve this, the CarWheel.cs script is added to each Wheel Mesh in the car hierarchy (see figure 20). The script updates the wheel's position and rotation based on the movement of the car's colliders, as shown in the code snippet below:

```
private void UpdateWheelPosision()
{
    targetCollider.GetWorldPose(out wheelPos, out wheelRot);
    transform.position = wheelPos;
    transform.rotation = wheelRot *  Quaternion.Euler(0,-90,0);
}
```

Listing 7: The UpdateWheelPosision() method in CarWheel.cs

The UpdateWheelPosision() method's purpose is to synchronize the rotation and position of a Wheel Mesh in accordance to its corresponding Wheel Collider. The method "GetWorldPose()" calls on the "targetCollider" object to obtain the target Wheel Colliders world position and rotation. Then, the obtained position and rotation values are assigned to the gameObject the script is attached to, which is the Wheel Mesh representation.

The "transform.rotation" line applies a correction to the rotation, as it rotates the wheel by 90 degrees around the y-axis. If this correction is not applied, the wheels will appear to be rotated incorrectly, as shown in figure 24.

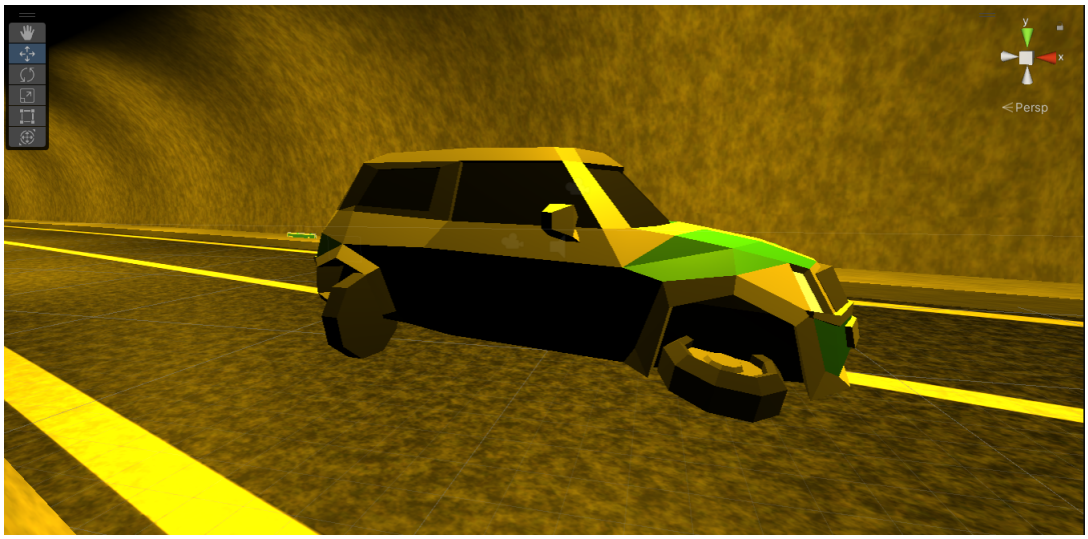


Figure 24: Wheel Meshes incorrectly rotated

Note however that the invisible Wheel Colliders are still rotated correctly. There are two issues with having the Wheel Meshes rotated incorrectly. Firstly, it doesn't resemble a real car, which affects the realism of the game. Secondly, the Mesh's Colliders are affected when they're incorrectly rotated, which can make it more challenging for the car to drive as expected. This is because the Mesh Colliders' orientation as well as their shape, plays in when detecting and responding to collisions. By correcting the Wheel Mesh's rotation, not only is the car's physical appearance enhanced, but it also improves its handling by allowing the Wheel Colliders to function with less interference from the Mesh Colliders.

4.1.5 From Pedestrian to Driver

CarInteract.cs is responsible for enabling the player to enter and drive any vehicle with this script attached. As for the exit functionality, it was decided to have CarController.cs call the ExitCar() method in CarInteract.cs, given that CarController.cs handles the player inputs. This cooperation between the two scripts will be discussed later.

Becoming The Driver

The interaction is initiated when the player approaches the car and presses the interact key. If close enough for the Raycast to hit the car, the Interact() method in CarInteract.cs will be called and insert the player inside of the car whilst also enabling or disabling the necessary scripts and components.

The script makes the player a child of the car within the Unity hierarchy and places the player gameObject inside the car gameObject. The player will be flipped 90 degrees to ensure their body doesn't stick out of the car. Certain scripts, such as RigidbodyMovement.cs are disabled so as to not interfere with the CarController.cs script, which is enabled to control the car.

The placement of the Player object inside the car allows for the players Capsule Collider to trigger the ExitCollider.cs scripts OnTriggerEnter() method, when the player drives into one of the

exit colliders. This removes the need to edit other scripts to enable the end game functionality, as the current OnTriggerEnter() method in ExitCollider.cs script checks for the "Player" tag.

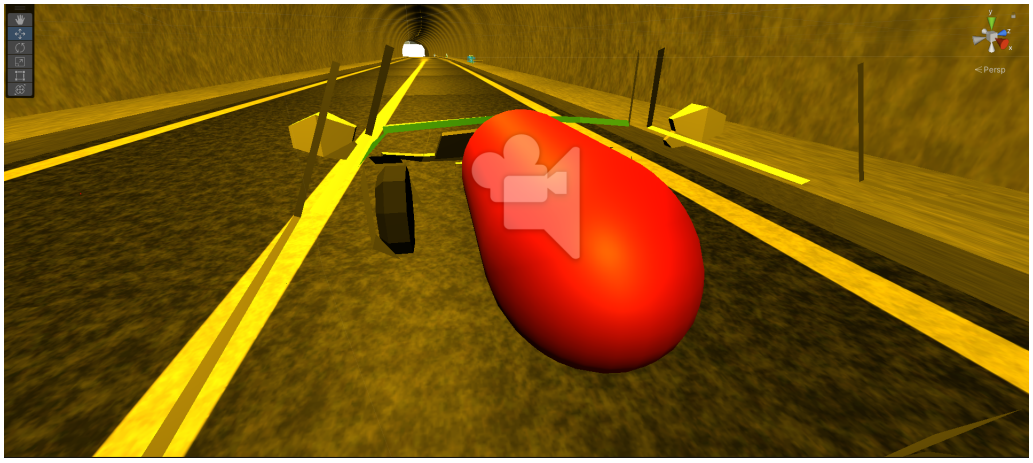


Figure 25: The player is placed laying on its side in the car

Returning to Pedestrian Form

For the exiting functionality to work, some cooperation with CarController.cs was needed to react to the player's key inputs.

In order to exit the car, some of the exit functionality has been placed inside the CarController.cs script as it handles player input. To make the explanation of the exiting process more understandable, the script of whatever event will be in a parenthesis behind the points below. The sequence of events is as follows:

1. (CarController.cs) The Player presses the exit button when driving the car.
2. (CarController.cs) Once pressed, the call to the ExitCar() method in CarInteract.cs is made.
3. (CarInteract.cs) ExitCar() is run and removes the player from the car.

This call from CarController.cs to CarInteract.cs is the reason for the reference shown in the figure 23 of the CarController.cs inspector.

An issue not mentioned previously was that upon entering the car, the player would lay on the ground and stay in place such that the car is able to drive away from the Player gameObject. This was due to the Rigidbody having an effect on the Player gameObjects placement. To solve this, the Rigidbody Component was simply removed with a Destroy() call.

Removing the Rigidbody obviously caused issues as the player was unable to move after exiting the car since RigidbodyMovement.cs uses it to move the player around when walking.

The fix was to reinsert the Rigidbody component again, but the CarInteract.cs script will lose the Rigidbody reference. To solve this issue, the method GetRigidbody() was added to the RigidbodyMovement.cs script to reobtain the reference after the Rigidbody is added to the player again. The listing below shows the process of the reinsertion.

```
 public void ExitCar()
{
    player.AddComponent<Rigidbody>();
    Rigidbody playerRigidbody = player.GetComponent<Rigidbody>();
    playerRigidbody.freezeRotation = true;
    player.GetComponent<RigidbodyMovement>().GetRigidbody();
    player.GetComponent<RigidbodyMovement>().enabled = true;
}
```

Listing 8: Rigidbody relevant code in CarInteract.cs

The Rigidbody is added and the GetRigidbody() method is called, which is a simple GetComponent<Rigidbody>() call to have the player regain the reference so that the RigidbodyMovement.cs script will work when the player exits the car.

Finally, a note regarding the scale of the car. "Green Car" currently uses its default transform scale of (100,100,100). "Drivable Green Car Rigidbody" on the other hand, has the transform scale (1,1,1). The reason for this configuration is due to the player. When the player enters the car, the player gameobject inherits the parent's size. If "Drivable Green Car Rigidboy" has a size of (100,100,100), the player will become massive and bug the game until the player leaves the car.

4.1.6 Car Camera

When entering the car, the camera isn't taken into consideration and the player does not get the point of view desired. To address this issue a camera called "Car Camera" was statically placed into the car.

That way the camera would be activated when the player enters the car, while the player camera (called "Main Camera") is still in the player but disabled. Unity will then automatically show the point of view from the Car camera.

The reason for this solution is that it is a lot simpler to just turn off the player camera and turn the car camera on as Unity will then automatically give the car camera's view. This removes the need to move the player camera back and forth between two positions which has been more difficult to pull off compared to the current solution.

4.1.7 Other Vehicle Prefabs

In addition to the car shown in the preceding sections, a bus and a truck prefab have been implemented, both of which can also be driven by the player. They work in the exact same way, with the truck having different scripts, namely: TruckInteract.cs and TruckController.cs. The figure below shows what the truck looks like.

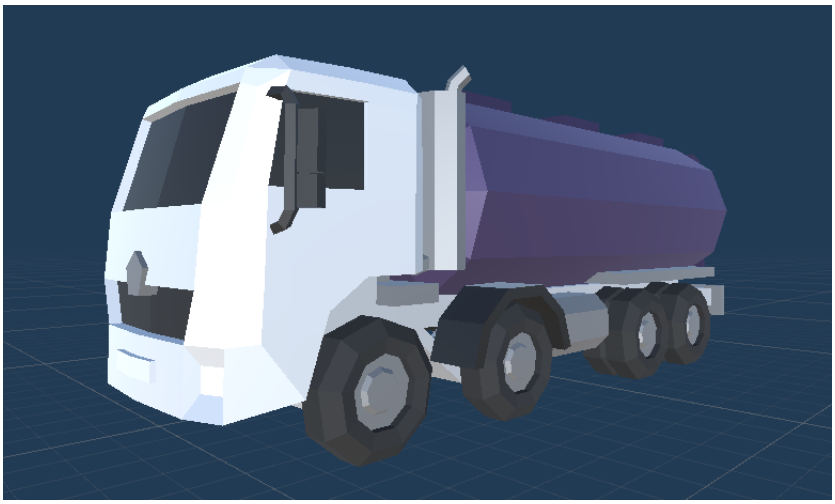


Figure 26: The truck prefab

In the same vein as the car and bus prefabs, the truck's scripts have the same functionality as CarController.cs and CarInteract.cs, that being letting the player drive the truck and interact with it. The difference being that they take the additional four wheels into consideration.

These scripts were added to create more realistic truck driving but the truck is fully functional using the CarController.cs and CarInteract.cs scripts, although using said scripts would requite some simplification by treating pairs of wheels as one wheel.

4.2 Rescaling the Tunnel

From the perspective of the player, the tunnels in the game appeared to be quite claustrophobic compared to actual tunnels. The player appeared to be about the same size as a driving lane, and could easily manage a jump from one end of the road to the other. This motivated the team to rescale the tunnels as well as replace the player prefab as has been discussed in section 3.6. Figure 27 shows how one of the tunnels looked like before.

Figure 27: A tunnel before rescaling. Appears quite small

The tree below show the files mentioned in this chapter.

```
Assets
└── Tunnel
    ├── TConfig.cs
    ├── TScene.cs
    ├── GenerateLights.cs
    └── Placement
        └── ExitCollider.cs
```

4.2.1 Maya or Unity

Since the tunnels are created in Maya, a question naturally arises of whether the rescaling should be done within Maya or handled by Unity. The team ultimately decided to do so within Unity for a few reasons:

- Solving the issue within Unity gives the game more control over how the tunnel should appear, rather than being dependent on a separate process scaling the tunnel exactly as needed.
- The game and tunnel modelling are ultimately separate processes, and it is important to keep in mind that the tunnels may be used in future applications beyond just this game. Therefore, it would be unreasonable to expect Maya to implement all of their unique requirements.
- The relative scale of the tunnels appeared to be mostly accurate in terms of the proportion between the walls and the roads. The main issue was the disparity between the size of the tunnel and objects within the Unity engine, and should therefore be handled by Unity. If however the assumption of relative scale turned out to be false, then the problem should be handled by Maya seeing as such an issue would affect any application making use of the tunnels.

4.2.2 TConfig

Seeing as the game still is in its relatively early stages, there could be a need to change the scale of the tunnel again later on. There has therefore been implemented a new class called TConfig. The intent behind this class is to gather some configuration parameters in one place. Among them is a constant called Tunnel_Size. Doing so allows anything dependent on the tunnel size to dynamically adapt according to the tunnel size. Objects may for example alter their own size or placement using the Tunnel_Size constant.

This solution enables one to easily change and test different tunnel sizes without having to navigate through the code and make many small changes. The class should also store other such variables which may benefit from a single place to change them.

```
 public class TConfig
{
    public const int TunnelSize = 4;
}
```

Listing 9: TunnelSize constant in the TConfig class

By adjusting the TunnelSize constant, one is able to increase or decrease the size of the tunnels. If desired, it can also be changed into a float for more precise adjustments.

```
private void CreateTunnel()
{
    tunnel.transform.localScale = Vector3.one * TConfig.TunnelSize;
}
```

Listing 10: TScene tunnel scaling

A simple multiplication on the tunnels transform is done to scale the tunnel to its preferable size. With this implementations the goal is to have a centralized way of determining the size of the gameobjects within the game.

4.2.3 Reference Frame

Since scale is relative, a frame of reference will always be needed in order to scale an object accordingly. This role was quickly delegated to the tunnel object, as it effectively serves as the entirety of the game world. Furthermore, the tunnels are all generated based on data from NVDB and were therefore thought to closely match their actual size. Other objects such as cars only exist within the game and do therefore not need to match any data. This assumption will be further discussed in section 5.2.

In the end, a tunnel size of 4 was chosen rather arbitrarily. This size was primarily chosen based on how the tunnel appears within the Unity Editor. When navigating a tunnel at this size using the editor, the tunnel was found to be easy to navigate, without making objects inside of it too small. Once a size for the tunnel had been determined, all other objects could be scaled according to the tunnel. The details will be discussed in the following sections.

4.2.4 Car & Player Scaling

Cars were first on the agenda. When scaling the cars, the road of the tunnel was used as the primary reference. The goal was to have the car fit within a single driving lane of the road, just as a real car should.

With this goal in mind, an important adjustment had to be made within the tunnel hierarchy. Rather than having the cars and tunnel be separate gameObjects within the hierarchy, the cars will instead be treated as children of the tunnel, as demonstrated in the figure below:

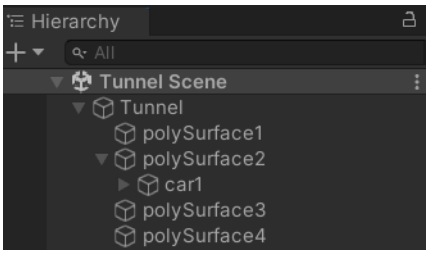


Figure 28: The hierarchy in the tunnel scene

Specifically, the car is a child of polySurface2, which refers to the first road segment of the tunnel. This is done due to objects in Unity scaling according to their parent object. If polySurface2

were to increase in size, then the car would also scale along with it. If the car were a separate object from polySurface2, then a size change to the tunnel would have no effect on the car, resulting in either too large or too small cars.

After the car model, it was time to scale the player. To ensure consistency, the player has been scaled using the car as its primary frame of reference. The goal was to have the player slightly taller than the car. Just as the car, the player is a child of the tunnel object, meaning the it will scale along with the tunnel.

4.2.5 Exit Collider Scaling

Every tunnel contains a gameObject known as an "exit collider". Whenever the tunnel scene is first loaded, two of these colliders called "First Exit" and "Last Exit" are generated and placed at at each end of the tunnel. They are a massive white box, meant to look like bright light entering the tunnel from outside. Figures 29 and 30 demonstrate their appearance within the game.

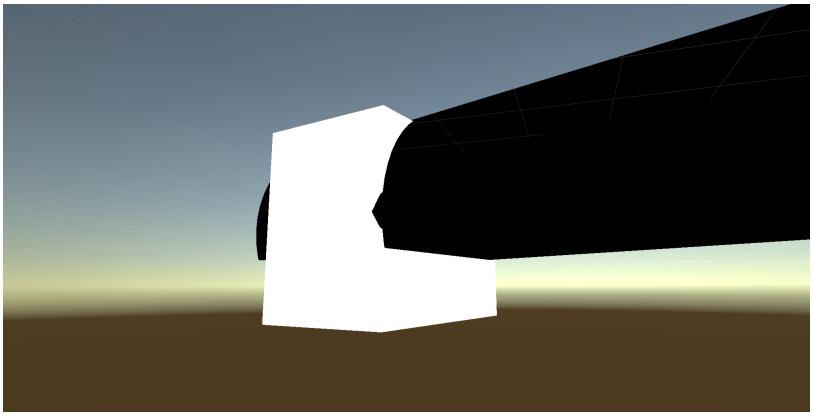


Figure 29: The exit collider seen from outside a tunnel

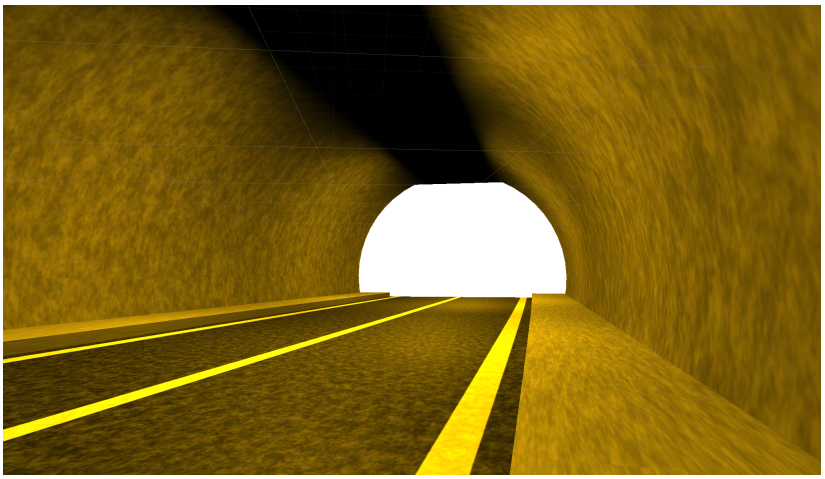


Figure 30: The exit collider seen from inside a tunnel

Their function is to stop the player in their tracks by calling the Disable() method from the RigidbodyMovements.cs script discussed in section 3.6.4.

These colliders will also have to scale according to the tunnel size, lest they will not be able to cover the entire opening. Unlike the preceding section 4.2.4, these are not children of the tunnel, thus meaning they have to be scaled through the code instead. The reason behind the different approach is that the exit collider does not currently have its own dedicated prefab, but is rather generated from the ground up within the code. It was therefore easier to change its scale through code as shown in listing 11.

```
private static void SetScale()
{
    Vector3 newScale = new Vector3(TMath.roadWidth * TConfig.TunnelSize * 1.25
    f , TMath.roadWidth * TConfig.TunnelSize , TMath.roadLength * TConfig.
    TunnelSize);
    exitBox.localScale = newScale;
}
```

Listing 11: Exit collider scaling

The code makes use of the TunnelSize within TConfig to adjust the size of the exit colliders. It also makes use of a roadWidth and roadLength constant from TMath.cs. (TMath is not used much and not currently relevant beyond these constants.) As such, the exit colliders will be able to grow along with the tunnel.

It should be noted however that this was simply a quick fix. There is no good reason to generate the exit colliders from the ground up within code, as it only results in more code and longer loading times. In the future, a prefab should be made for the exit colliders, allowing the game to simply instantiate said prefab. With the prefab in place, it should be instantiated as a child of the corresponding road segments just as the cars. Doing so would allow them to scale automatically without requiring any extra code.

4.2.6 Lighting Adjustment

Finally, the lights in the tunnel also have to change according to the tunnel size. Static lighting would lead to small tunnels being blindingly bright whilst larger tunnels would be pitch black. Unlike the objects in the previous sections however, the lights cannot be adjusted by simply scaling them. Rather, their range and intensity needs to be altered so as to ensure their light reaches far enough and is strong enough. (Their colour has also been adjusted, which is why it varies between images throughout this thesis.) These parameters do not automatically change when the size of the lights change, meaning the same trick regarding children of the tunnel utilised in the previous sections, will not work. They instead have to be adjusted within code, as seen in listing 12.

```
private void GenerateLighting()
{
    light.range = TConfig.TunnelSize * 15;
    light.intensity = TConfig.TunnelSize * 2;
}
```

Listing 12: GenerateLighting() light scaling

One should note that the numbers 15 and 2 are magic numbers, and are chosen based on nothing more than trial and error. They work well with tunnel sizes between 0.5 and 8. Beyond that however,

the light is will quickly grow too strong for the player to be able to see anything but yellow in the tunnel.

4.3 Integrating NVDB Into the Game

In earlier versions of the project, data from NVDB was used to generate a model of a tunnel. Said model was then used as the setting within the game. The problem was that these two applications were wholly separate, meaning the tunnels had to be manually imported into the game. Furthermore, the game did not have any access to data from NVDB beyond the tunnel models provided to it. This section will explain how this problem has been rectified, how to use the new implementation, as well as guidelines for how to scale the code in the future.

4.3.1 Directory Overview

Recall from section 2.1 that the project consists of two main components, the modelling application and the game. The process begins within the proceduralmodeling directory. When a tunnel model is generated as explained in appendix C, some other code will run to retrieve the desired data from NVDB. Said data is then stored as JSON within the game's directories in order to later be used by the game. Specifically, the target location is the data directory shown below, although this can be changed later through config.py.

```
root
├── 3D-self-rescue-game-project
│   └── Assets
│       └── Resources
│           └── Tunnels
│               └── data
└── proceduralmodeling
    ├── config.py
    └── addData.py
```

It should be noted that this code currently only supports tunnel tubes as that is what the team primarily has been working with. The modelling of entire tunnels has proven too unstable and should therefore be improved before there can be significant benefits from gathering data directly from parent tunnels.

4.3.2 addData.py

The code responsible for the data harvest is called addData.py and will henceforth be referred to as addData. The old code responsible for collecting the data needed for modelling tunnels is also still intact and will henceforth be reffered to as oldData. The reasons for recreating it from the ground up in such a fashion are as follows:

- oldData is heavily coupled with the modelling code, meaning removing or reworking oldData is a very difficult and complicated process, requiring some knowledge about both the modelling and the data retrieval processes. Creating a new implementation from scratch was therefore easier. To avoid the same problem in the future, addData has been kept in a separate file which

is then called from main.py. Should one wish to update addData in the future, one could easily do so within that file without worrying about breaking the modelling process.

- Ultimately, oldData and addData serve different purposes. oldData was written for the purpose of generating 3D models (2.1.1) and was never concerned with creating a game within these tunnels. It therefore removes some data the game might be interested in and adds some data the game may never have any use for. Rather than insisting on these differing processes relying on the same data, it is therefore preferable to keep them separate. This has the added benefit of allowing someone to update oldData without needing any knowledge of the game and vice versa.

As mentioned in section 3.1, oldData relies on sending HTTP requests to the NVDB API. addData instead uses a python library [11] created for NVDBv3. Hence the library will have to be installed using the following command for addData to function.

```
pip install nvdbapi-v3
```

Sections 4.3.3 through 4.3.6 will outline how addData works and how it can be expanded upon in the future.

4.3.3 Program Flow

The code within listing 13 is taken from addData and will be referred to throughout this section. Within it is a class called DatasStore. This class is responsible for gathering data from NVDB as well as storing it in the right location, meaning any code related to these tasks should be within a method of this class.

```
class DataStore():
    default_methods = ["Trafikkmengde", "Tunnel", "Skiltpunkt"]

    def __init__(self, lopId):
        self.lopId = lopId
        self.tunnellop = self.Tunnellop()
        self.Navn()
        self.tunnellop["egendefinert"] = {}

    def Tunnellop(self):
        return nvdbapiv3.finnid(self.lopId)[0]

    def Navn(self):
        for egenskap in self.tunnellop["egenskaper"]:
            if egenskap["navn"] == "Navn":
                self.tunnellop["navn"] = egenskap["verdi"]
                self.tunnellop["egenskaper"].remove(egenskap)

    def Main(self, methods=default_methods):
        for method in methods:
            getattr(self, method)()
        return self.tunnellop

    def Store(self):
        pathlib.Path(config.DATA_PATH).mkdir(parents=True, exist_ok=True)
        path = config.DATA_PATH + self.tunnellop["navn"] + ".txt"
```

```
        with open(path, "w") as fil:
            json.dump(self.tunnellop, fil)
            fil.close()
        print(f"Data saved as {path}")
```

Listing 13: The essential functions within addData

Python does unfortunately not support private and public methods. If it did, most of the methods of the class would be private. However, the __init__(), Main() and Store() methods shown in the listing are intended as public methods, and are the only methods that should be used by outside code. When interacting with the class, one is intended to:

1. First create an instance of the class. This requires supplying the ID of a tunnel tube. You may notice that when doing so, the constructor will call the Tunnellop() and Navn() methods. Tunnellop() retrieves all basic information about the tunnel tube. It therefore needs to run first so as to allow other methods to build upon the retrieved data. The Navn() method is a bit special in that it does not retrieve any new data. It only modifies the data already retrieved (and therefore needs to run after Tunnellop() has been called) to make the name of the tunnel more readily accessible, since it is often used by other code.

2. One would then call the Main() method, which simply calls all the "private" methods of the class, thereby supplying additional data to the class. (This process will be further discussed in section 4.3.4.) Note that Main() only calls the methods within the list "default_methods". This is to prevent it from running all methods within the class such as Store(). This means that any new methods created will need to be added to the list. One can also supply their own list of methods to call as parameters to Main() if only specific sets of data are desired.

3. Finally one will call the Store() method to store the data correctly wherever it is needed. The reason for separating this from Main() is to allow one to use the class for gathering data to be used in code, without needing to save it to their file system. It also has the added benefit of making it easier to change the saving process, such as where the data is saved.

In summary, this entire process can be run using the code shown in listing 14 below.

```
1  datastore = DataStore(489792809)
2  datastore.Main()
3  datastore.Store()
```

Listing 14: A demonstration of how to run addData. The code will gather all data (since Main() did not receive any inputs) belonging to the tunneltube with ID 489792809 and store it as JSON within the game directories

4.3.4 Building Upon Retrieved Data

As explained in section 4.3.3, the Tunnellop() method will always be called first, thereby gathering basic data about the tunnel. The problem from the game's perspective then is that it lacks data regarding its children, parents and other additional data the game may be interested in. Section 2.2.1 explained that each tunnel tube contains a field called "relasjoner" with the IDs of every child and parent object of the tunnel tube, but unfortunately lacks much data beyond said IDs. This section will explain how to go about retrieving additional information about these objects when desired.

Before doing so however, it is worth taking a deeper look at the "egenskaper" list than what was explored in section 2.2.1.


```
{
    "id":220812,
    "navn":"Assosierte Signalpunkt",
    "egenskapstype":"Liste",
    "datatype":"Liste",
    "innhold":[
        {
            "id":200812,
            "navn":"Assosiert Signalpunkt",
            "egenskapstype":"Assosiasjon",
            "datatype":"Assosiasjon",
            "verdi":489793251
        },
        {
            "id":200812,
            "navn":"Assosiert Signalpunkt",
            "egenskapstype":"Assosiasjon",
            "datatype":"Assosiasjon",
            "verdi":489793253
        }
    ]
},
```


Figure 31: Some more tunnel tube "egenskaper" containing an "Assosiasjon"

Certain "egenskaper" contain associations to other objects as shown in the figure. These associations have a field called "innhold" which in principle is the same as the "relasjoner" list shown before, resulting in some redundant storage. As an example, figure 31 shows an "egenskap" called "Associated Signal Point (Assosierte Signalpunkt)". Within its "innhold" is a list of all "signal point (signalpunkt)" associated with the tunnel. addData will be used to expand the elements of these "innhold" lists rather than the "relasjoner" for the following reasons:

- "egenskaper" is overall a nicer structure to work with, as the object type can be found directly under "navn" rather than having to navigate to "navn" inside of "type".
- Navigating the data structure will be easier as most, if not all, desired data can be found in the same place. Hence it is not necessary to navigate between "egenskaper" and "relasjoner" to find the desired data.

In conclusion, the goal of this section is to take an "egenskap" containing an association and reconstruct it with more data beyond simply the ID of the object. This is demonstrated in figures 32 and 33 below.


```
"innhold":[
  {
    "id":200004,
    "navn":"Assosiert Skiltplate",
    "egenskapstype":"Assosiasjon",
    "datatype":"Assosiasjon",
    "verdi":530886402
  }
]
```


Figure 32: Original contents of "Assosierte Skiltplate"


```
"innhold":[
  {
    "id":530886402,
    "href":"https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/96/530886402/4",
    "metadata":{ },
    "egenskaper":[
      {
        "id":100096,
        "navn":"PunktTilknytning",
        "egenskapstype":"Stedfesting",
        "datatype":"GeomPunkt",
        "stedfestingstype":"Punkt",
        "veglenkesekvensid":2303031,
        "relativPosisjon":0.02884607,
        "retning":"MED",
        "sideposisjon":"H",
        "kj\u00f8refelt":[ ]
      },
      {
        "id":4795,
        "navn":"Geometri, punkt",
        "egenskapstype":"Geometri",
        "datatype":"GeomPunkt",
        "verdi":"POINT (-36443.3146276983 6566001.13707952)",
```


Figure 33: Expanded contents of "Assosierte Skiltplate". Notice that "id" has the same value as "verdi" in figure 32

Before the rewrite, each item in the "innhold" list is essentially nothing more than the ID of a "sign plate (skiltplate)". It is then rewritten by first retrieving the data of said ID from NVDB and replacing the original list item with the returned data. The result now has its own list of "egenskaper" in addition to the ID. Listing 15 below demonstrates how this is done within the code using "sign point (skiltpunkt)" as an example.

```
def Skiltpunkt(self):
    for egenskap in self.tunnellop["egenskaper"]:
        if egenskap["navn"] == "Assosierte Skiltpunkt":
            nyttInnhold = []
            for skiltpunkt in egenskap["innhold"]:
                skiltpunkt = nvdbapiv3.finnid(skiltpunkt["verdi"])[0]
                self.Skiltplate(skiltpunkt)
```

```
8                 nyttInnhold.append(skiltpunkt)
9             egenskap["innhold"] = nyttInnhold
10
11 def Skiltplate(self, skiltpunkt):
12     for egenskap in skiltpunkt["egenskaper"]:
13         if egenskap["navn"] == "Assosierte Skiltplate":
14             nyttInnhold = []
15             for skiltplate in egenskap["innhold"]:
16                 skiltplate = nvdbapiv3.finnid(
17                     skiltplate["verdi"])[0]
18                 nyttInnhold.append(skiltplate)
19             egenskap["innhold"] = nyttInnhold
```

Listing 15: Code within addData used to gather data about a "Skiltpunkt"

Recall that the goal is to find an "egenskap" (not a "relation (relasjon)") containing the ID of a child object of the tunnel tube. Lines 2-3 of the code finds the "egenskap" "Associated Signpost (Assosierte Skiltpunkt)". This "egenskap" will have a field "innhold" containing the desired IDs. Hence the code loops through the list and uses the nvdbapiv3 library to retrieve data about the "skiltpunkt" before finally replacing the original contents of "innhold" with the retrieved data. Note the "[0]" at the end of line 6. Data from NVDB is always returned in a list, even if there is only a single item within the list. Keeping this list would only add unneeded complexity to the data structure, and there will only ever be one item in the list since the code is searching for an ID. Hence, only the first item is kept and not the list itself.

Moving on to the Skiltplate() method, it should be noted that it follows the same basic template. Any method used for this purpose of replacing IDs with their data should follow the same structure. It should be noted however that Skiltpunkt() makes a call to Skiltplate() (on line 7). This is because Skiltplate() depends on Skiltpunkt() running before it. If they were to run in opposite order, the data structure would not have any "skiltplate" IDs as they are child objects of a "skiltpunkt". Skiltpunkt() will therefore make the call itself to make the dependency more explicit. Furthermore, this means that "skiltplate" should *not* be called from Main() and therefore not added to default_methods. The methods are nevertheless kept separate for the purposes of readability and allowing changes to one without affecting the other, thus improving scalability.

4.3.5 Adding New Values

The previous section explained how the code will go about retrieving data related to children IDs. This section will explain the process of adding brand new pieces of data that were never referenced by the original tunnel tube. The Trafikkmengde() method will be used to demonstrate and is shown in listing 16.

```
1 def Trafikkmengde(self):
2     navn = self.tunnellop["navn"]
3     sokeObjekt = nvdbapiv3.nvdbFagdata(540)
4     sokeObjekt.filter({"overlapp": "67(1081='" + navn + "')"})
5     self.tunnellop["egendefinert"]["trafikkmengde"] = sokeObjekt.
      nesteForekomst()
```

Listing 16: Method used to retrieve data about the traffic within a tunnel tube

Within the code there are quite a few numbers and functions without a clear meaning. The following will attempt to explain what is needed to know to be able to understand this particular method. For more exhaustive documentation, please refer to the citations throughout.

The first step of the process is to create a search object, hence referred to as "sokeObjekt". This is a class providing an interface to interact with the NVDB API. An instance of the class is constructed on line 3 by passing in 540 as a parameter. 540 in this case is the ID, not of a specific object, but rather of an object type ("traffic volume (trafikkmengde)" to be precise). A full list of object IDs can be found within the Data Catalogue [36]. The URL equivalent of the "sokeObjekt" would be the following:

https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/540

Note however that by the third line of the code, no data has been retrieved yet.

Running the search using this "sokeObjekt" would return all objects in NVDB with object ID 540. The next step is therefore to add filters to the "sokeObjekt". Line 4 of the code adds an "overlap (overlapp)" filter to the "sokeObjekt", which filters the objects based on their physical positions overlapping. The number 67 is added as it is the object ID for tunnel tubes. Without the rest of the line, this filter would return all "trafikkmengde" objects which overlap with a tunnel tube object. As a URL this would look like the following:

https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/540?overlapp=67

The rest of line 4 is used to only retrieve the "trafikkmengde" within the tunnel tube in question. The number 1081 is the ID of an "egenskap" of the tunnel tube [36]; the "navn" "egenskap" in this case. So the filter will find the "navn" "egenskap" of the tunnel tube and compare it to the name that is supplied to the filter. The URL equivalent is shown below using "Blombakktunnelen" as an example:

https://nvdbapiles-v3.atlas.vegvesen.no/vegobjekter/540?overlapp=67(1081="Blombakktunnelen")

Additional examples of working with filters can be found in the citation [37].

After all the filters have been applied, data is retrieved using the nesteForekomst() method. This method only returns the next object that fulfills all the requirements. The game does not need more than one "trafikkmengde" object and therefore only calls nesteForekomst() once. If more objects are needed, they can be retrieved by iterating over the "sokeObjekt" rather than repeated calls to nesteForekomst() [11].

Recall from listing 13 that the constructor for addData will create an empty dictionary called "custom defined (egendefinert)". This is where any additional information is intended to be stored. The final step of Trafikkmengde() is therefore to create a field within "egendefinert" and store the retrieved data there.

4.3.6 Scaling the Code

Due to the volume of data within NVDB and the continuous updates to said data, it is not possible to create a version of addData that will not require some future updates of its own. There has therefore been a focus on writing the code in a scalable manner such that it can be easily updated and changed in the future. Thus this section will highlight some of the guidelines for expanding the code:

- Each method should only be responsible for one type of object. Meaning if there is a need for more data, then a new method should be created for it.
- Any method that is desired to run should be added to the default_methods list. However:
- Be careful when working with data for children of children as they *should* depend on their parent already being present in the data structure. This is to prevent the methods for the children from becoming too long. Children methods should therefore be called from their parents and not be added to the default_methods list.
- All parent methods are currently alphabetically sorted in the file, followed by their respective child methods. The main methods described in listing 13 are exempt from this sorting and instead appear at the beginning of the file.

4.3.7 Running Data Retrieval

Whenever a tunnel tube (not a tunnel) is modelled as described in appendix C, addData will automatically run its default methods. It can also be run separately from the modelling application, in case one only needs to retrieve new data about a tunnel without a need for a brand new model. This is achieved by running addData.py from the command line with a tunnel tube ID supplied. The following will run the program for the tunnel tube with ID 489792809:

```
py .\addData.py 489792809
```

Any additional arguments can be used to determine which methods to run. The following will only run the Skiltpunkt() method (in addition to the methods which are set to always run):

```
py .\addData.py 489792809 Skiltpunkt
```

If the program is run without any inputs, it will return a link to vegkart for finding a tunnel tube ID.

4.3.8 TNvdbData

addData as described in the preceding section 4.3.2, is separate from the actual game and could in theory be used for other purposes. The game therefore needs a way to access the stored data, which is accomplished through a class called TNvdbData. It can be found within the file of the same name, as shown in the tree below.

```
root
├── 3D-self-rescue-game-project
│   └── Assets
│       └── Resources
│           └── Scripts
│               └── Tunnel
│                   └── TNvdbData.cs
└── proceduralmodeling
    └── addData.py
```

Sections 4.3.9 through 4.3.12 will explain the details of how this has been implemented.

4.3.9 Representing JSON in C#

TNvdbData makes use of Unity's JSON utility [25]. This functions by creating a class in C# that matches the structure of the JSON formatted data. This will be demonstrated using figure 34 and listing 17 below.


```
{
   "id":1977,
   "navn":"Tunnelprofil",
   "egenskapstype":"Tekstenum",
   "datatype":"FlerverdiAttributt, Tekst",
   "verdi":"T10",
   "enum_id":4350
},
```


Figure 34: An "egenskap" stored as JSON

```
[Serializable]
public class Egenskap
{
    public string navn;
    public string verdi;
}
```

Listing 17: The egenskap class in C# (not final)

When the JSON utility is called, it will read the values of each JSON key-value pair and store them within their corresponding fields within the C# class. The names of the fields have to match exactly so that the JSON utility knows where to store what. As an example, using "Navn" or "name" rather than "navn" will not work. Also note that there are more keys in the JSON data than there are fields in the C# class. JSON keys without a corresponding field will be ignored and therefore not accessible for the game. Should there for instance be a need for the ID of an "egenskap", all one would have to do is add an "id" field within the Egenskap class. Finally, note that the class has to be marked as serializable for the JSON utility to work [25].

After the Egenskap class has been defined, it can be used within the NvdbData class as shown in listing 18 below.

```
[Serializable]
public class NvdbData
{
    public string navn;

    public List<Egenskap> egenskaper;

    public Relasjoner relasjoner;

    public Egendefinert egendefinert;
}
```

Listing 18: The NvdbData class. Note that this is an early implementation differing from the current implementation. It is only shown for demonstration purposes

In addition to Egenskap, the classes Relasjoner and Egendefinert have also been defined. These classes are needed because those fields store more complex data than a single string. The final product

will therefore be a hierarchy of child and parent classes closely resembling the original NVDB data structure. Unlike the fields, the class names are not required to match the data. It was nevertheless decided to have them match to clearly communicate what the classes represent.

NvdbData is the name of the class at the very top of the C# hierarchy. It will in other words represent the tunnel tube. TNvdbData on the other hand, is the class managing the entire process. It contains a field of type NvdbData and a method for loading data using the JSON utility as shown in listing 19.

```
public static class TNvdbData
{
    public static NvdbData Data { get; private set; }

    public void Awake()
    {
        string path = TConfig.TunnelDataPath + SETTINGS.TunnelName + ".txt";
        Data = JsonUtility.FromJson<NvdbData>(File.ReadAllText(path));
    }
```

Listing 19: The TNvdbData class. A static class that reads the JSON file for the current tunnel and stores the data within a field of type NvdbData

4.3.10 Early Implementations

With the NvdbData class ready, the game now needs some code to retrieve the data stored in the NvdbData class. This section will show some earlier implementations of the code and discuss the problems with them, in order to help explain why the current implementation discussed in the following section 4.3.11 looks the way it does and how it solves the aforementioned problems.

The simplest solution would be to let the game traverse the data structure and access values as needed. Since the code is usually interested in an "egenskap" however, this would often result in a foreach loop like the one in listing 20.

```
foreach (Egenskap egenskap in TNvdbData.Data.egenskaper)
{
    if (egenskap.navn == "Lengde")
    {
        Length = egenskap.verdi;
    }
}
```

Listing 20: Traversing TNvdbData in code. Not very scalable

Although this approach was functional, it led to many repetitive lines of code. It soon became apparent that this should be moved into its own method. Hence the GetEgenskap() method shown in listing 21 was created within TNvdbData.

```
public static Egenskap GetEgenskap(string navn)
{
    foreach(Egenskap egenskap in Data.egenskaper)
    {
        if (egenskap.navn == navn) return egenskap;
    }
}
```

Listing 21: The GetEgenskap() method. A better solution, yet still has some problems

This worked better as it allowed the rest of the code to simply call this one method rather than creating a new foreach loop every time. Problems arose from the fact that this method can only access "egenskaper" of the tunnel tube. Later on, some code needed access to "egenskaper" of the tunnel, which resulted in a GetTunnelEgenskap() method. Said method was nearly identical to listing 21, with the sole difference being that it traversed Data.relasjoner.tunnel.egenskaper instead. Some time later, other code needed access to "egenskaper" within the "trafikkmengde" field. It was apparent that the code would only keep snowballing with more methods created every time new fields were added to the data. In the end this led to the current implementation as described in the following section.

4.3.11 Current Implementation

The issues described in section 4.3.10 were ultimately solved by introducing one new class shown in listing 22 below.

```
[Serializable]
 public class EgenskapObjekt
{
    public List<Egenskap> egenskaper;

    public Egenskap GetEgenskap(string navn)
    {
        foreach (Egenskap egenskap in egenskaper)
        {
            if (egenskap.navn == navn) return egenskap;
        }
        return null;
    }
}
```

Listing 22: The EgenskapOjekt class. The solution to the problems with the code

In brief, the problem with the previous implementation was that many similar objects were stored in different places. This solution takes advantage of their similarities and creates the EgenskapObjekt class consisting of everything they have in common. Namely a list of "egenskaper" and a method to retrieve those "egenskaper". Any class containing "egenskaper" will now inherit from the EgenskapObjekt class. As an example, the final NvdbData class is shown in listing 23 below.

```
[Serializable]
public class NvdbData : EgenskapObjekt
{
    public string navn;

    public Relasjoner relasjoner;

    public Egendefinert egendefinert;
```

Listing 23: The current version of NvdbData. It now inherits from EgenskapObjekt

Notice that unlike listing 18, the class no longer contains a list of "egenskaper", as it is now inherited from the EgenskapObjekt class instead. Any class in need of such a list will now inherit from the EgenskapObjekt class. As a result, any "egenskap" can be accessed by navigating the NvdbData structure and calling GetEgenskap() on the appropriate field. Some examples include:

- TNvdbData.Data.GetEgenskap("Lengde");

- TNvdbData.Data.relasjoner.tunnel.GetEgenskap("Sum lengde alle løp");
- TNvdbData.Data.egendefinert.fartsgrense.GetEgenskap("Fartsgrense");

4.3.12 Scaling the Code

Just as with addData, TNvdbData is expected to grow in tandem with the project. What follows is therefore a summary of the guidelines for scaling this file to accommodate future needs:

- To access more data, all that is needed is to add another field of the same name to wherever is appropriate in the data structure. However:
- Remember to make use of the EgenskapObjekt class whenever new fields containing "egenskaper" are created.
- Many fields have no use beyond their "egenskaper" and can therefore be of type EgenskapObjekt rather than inheriting from the class. New classes should only be created when there is a distinct need to distinguish between the two classes. Listings 24 and 25 below demonstrate how this could be done. Unless this distinction is needed, a lot of space can be saved by adding the fields to the EgenskapObjekt class rather than creating a new class inheriting from it.

```
public class Egendefinert
    {
        public EgenskapObjekt trafikkmengde;

        public EgenskapObjekt fartsgrense;
```

Listing 24: The current implementation of the Egendefinert class. The fields trafikkmengde and fartsgrense are both of the same type and therefore contain the same types of data

```
public class Egendefinert
    {
        public Trafikkmengde trafikkmengde;

        public EgenskapObjekt fartsgrense;

        public class Trafikkmengde : EgenskapObjekt
        {
            public string id;
        }
```

Listing 25: An alternative implementation of Egendefinert

In the alternative implementation shown in listing 25, a new class has been created for "trafikkmengde" in order to access its ID, which is currently inaccessible within "speed limit (fartsgrense)". Note that this could be solved in an easier and arguably better way by adding the "id" field to the EgenskapObjekt class instead.

4.4 Placement of Objects

Previous implementations of the game would place a select few objects and space them evenly throughout the tunnel using some arbitrary rule for how to space them. Now that the game itself has

access to tunnel data (section 4.3), it is capable of placing objects according to their actual location within the real tunnel. It is also able to only place objects present in the real tunnel, rather than insisting on placing every object somewhere within every tunnel.

All code discussed in this chapter can be found within the files shown below:

```
Tunnel
├── Placement
│   ├── NvdbPlace.cs
│   └── NvdbSkiltPunkt.cs
└── TNvdbData.cs
```

4.4.1 Stedfestinger

Every object within NVDB has a field known as a "location fixing (stedfesting)" [41]. They are a collection of data used to determine the physical location of an object. To understand how they do so, one must understand that the road network within NVDB is made up of many small "road links (veglenker)". One can think of these as representations of all the roads across Norway. A "stedfesting" is then used to describe on which "road link (veglenke)" an object is placed, and where on said "veglenke".

Figure 35: Some "veglenker" highlighted in Vegkart, along with arrows indicating their direction

An example "stedfesting" could look like the following: 0.02884607@2303031. The interpretation of this example is that the object is placed on "veglenke" 2303031 at a relative position of 0.02884607. Relative position will always be a number between 0 and 1. Hence the object in this case is very close to the beginning of the "veglenke". A position of 0.9255135 on the other hand would be closer to the end.

A "stedfesting" may also be given in the following format: 0.14285714-0.27622061@2303015. Just as the previous example, 2303015 indicates the ID of the "veglenke" this object is placed on. The first two values separated by the hyphen indicate a range of relative positions occupied by the object. Tunnel tube "stedfestinger" will often be given in this format.

4.4.2 Approximating Object Position Within Tunnel Tube Model

With the "location fixings (stedfestinger)" retrieved, the code now needs a way to approximate where inside the tunnel tube a relative position corresponds to. This is accomplished using a list of all road segments within the tunnel. Said list is generated within TScene whenever the tunnel scene is loaded. Code within listing 26 is used to compute which road segment best corresponds to the given relative position.

```
public static int CalculateSegmentNumber(float relativPosisjon)
{
    float relativeLength = slutt - start;
    float segmentLength = relativeLength / TData.Get(Types.Road).Count;

    int res = 0;
    while (start + res * segmentLength <= relativPosisjon)
    {
        res++;
    }
    return res - 1;
}
```

Listing 26: The CalculateSegmentNumber() method of the NvdbPlace class (section 4.4.3)

The method mostly consists of various mathematics. It first calculates the length of the tunnel tube relative to the "veglenke" using the variables "end (slutt)" and "start (start)". Within those variables is the relative position of the beginning and the end of the tunnel tube, hence the result says how much of the "veglenke" is inside the tunnel.

The next step is to calculate how long a road segment of the tunnel tube model is relative to the "veglenke", which is done by dividing the length of the tunnel tube by the number of road segments. Finally, the while loop is used to determine how many such road segments are needed to reach the input location and returns the result (after subtracting 1 since list indices begin at 0).

Note that the result is ultimately an approximation. This should not pose a problem as the tunnel consists of many short road segments. Should more accurate results be needed, then the code would need to calculate precisely how far within the road segment the object should be placed. A possible expansion of listing 26 is demonstrated as pseudocode in listing 27

```
while (start + res * segmentLength <= relativPosisjon)
{
    res++;
}
float positionAdjustment = relativPosisjon - (start + res * segmentLength);
```

Listing 27: Calculating the exact position of the object

Furthermore, there are some objects which although associated with the tunnel tube, are not present inside of it. This is demonstrated in figure 36 below.

Figure 36: A tunnel tube (green) with two associated "skiltpunkt" (blue) positioned outside the tunnel

Seeing as the game solely takes place inside the tunnel tube, there is no need to place any objects outside of it. The method shown in listing 28 has therefore been implemented to check whether an object is inside the tunnel or not. If the code were to run without this check, then CalculateSegmentNumber() would return 0 or an out of bounds index for any object outside the tunnel.

```
public static bool IsInsideTunnel(Stedfesting stedfesting)
{
    float relativPosisjon = float.Parse(stedfesting.relativPosisjon);
    return start < relativPosisjon && relativPosisjon < slutt;
}
```

Listing 28: The IsInsideTunnel method() of the NvdbPlace class (section 4.4.3)

4.4.3 NVDB gameObject

As described in section 3.5, the file TScene.cs has undergone some restructuring. One of the main motivations for this was to lessen the excessive number of files dependent on TScene. This has been achieved by taking advantage of Unity's Awake() method. Since Awake() is called as soon as the gameObject is loaded into the scene, code will effectively run automatically as soon as the scene is loaded, rather than having to rely on TScene making explicit calls to have the desired code run. For the purposes of placement, the NVDB gameObject was created in the tunnel scene as shown in figure 37

Figure 37: The Nvdb gameObject in the hierarchy and its children

Another benefit of this approach is making dependencies more explicit, using a structure where children depend on their parents. The Nvdb object contains the script TNvdbData, meaning all data will be loaded using the Awake method (shown in listing 19) as soon as the object is loaded. The NvdbPlace and SkiltPunkt objects are both dependent on the data already being loaded. They are therefore positioned as children of Nvdb so as to guarantee the data will always be loaded when they run.

The NvdbPlace object contains the script of the same name. It contains the IsInsideTunnel() and CalculateRoadSegment() methods shown in listings 28 and 26 respectively. These are general methods that should prove useful for any class wishing to place an object.

The NvdbPlace class does require some setup for these methods to function properly. This takes place within the Awake() method. Hence all code depending on this setup is placed within children of the gameObject (such as SkiltPunkt). The relevant code is shown in listing 29.

```
public class NvdbPlace : MonoBehaviour
{
    private static float slutt;
    private static float start;

    void Awake()
    {
        Stedfesting stedfesting = Data.lokasjon.stedfestinger[0];
        start = float.Parse(stedfesting.startposisjon);
        slutt = float.Parse(stedfesting.sluttposisjon);
    }
```

Listing 29: The NvdbPlace class and its Awake() method

4.4.4 Placing Skiltpunkt

The actual placement of objects was a relatively late addition to the project. "skiltpunkt" is therefore currently the only object supported. It was decided to implement this as a proof of concept as well as an example upon which future additions to the code can be built. Listing 30 shows the NvdbSkiltPunkt class.

```
1 public class NvdbSkiltPunkt : MonoBehaviour
2 {
3     [SerializeField]
4     private GameObject skiltPunktPrefab;
5
6     void Awake()
```

```
7   {
8       Egenskap skiltPunktene = Data.GetEgenskap("Assosierte Skiltpunkt");
9       foreach (EgenskapObjekt skiltpunkt in skiltPunktene.innhold)
10      {
11          Stedfesting stedfesting = skiltpunkt.lokasjon.stedfestinger[0];
12          if (!NvdbPlace.IsInsideTunnel(stedfesting)) continue;
13
14          float relativPos = float.Parse(stedfesting.relativPosisjon);
15          int segmentNumber = NvdbPlace.CalculateSegmentNumber(relativPos);
16
17          Transform road = TData.Get(Types.Road, segmentNumber);
18          Vector3 position = road.GetComponent<Renderer>().bounds.center;
19
20          switch (stedfesting.sideposisjon)
21          {
22              case "H":
23                  position.x += TMath.roadWidth;
24                  Instantiate(skiltPunktPrefab, position, road.rotation,
    road);
25                  break;
26              case "V":
27                  position.x -= TMath.roadWidth;
28                  Instantiate(skiltPunktPrefab, position, road.rotation,
    road);
29                  break;
30          }
31      }
32  }
```

Listing 30: The NvdbSkiltPunkt class. Places "skiltpunkt" when Awake() is called

When the Awake() method is called, it has to go through three important steps:

1. Lines 8-9 will retrieve all "skiltpunkt" associated with the tunnel and begin looping through them. In doing so, it will run the following two steps for every "skiltpunkt" associated with the tunnel.
2. Lines 11-15 will determine whether the "skiltpunkt" is inside the tunnel or not. If it is, then the appropriate segment number will be calculated.
3. With the segment number ready, the final lines of code are used to instantiate the provided prefab. Lines 17-18 retrieve the road segment corresponding to the calculated segment number as well as its position. Before instantiating however, the switch case at line 20 is used to account for the object's "side position (sideposisjon)", in this case meaning whether it is positioned on the right or left side of the road.

The provided prefab is currently a simple placeholder as shown in figure 38.

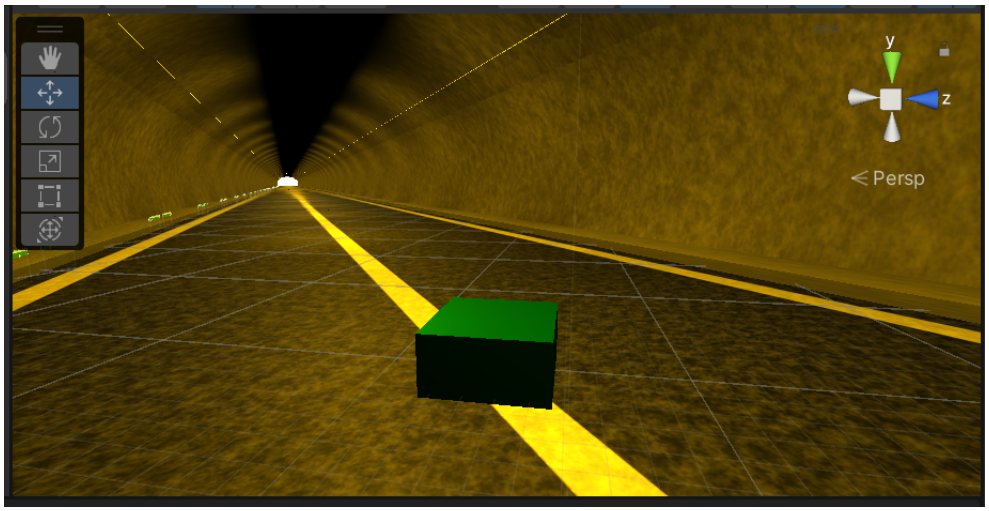


Figure 38: A "skiltpunkt" placeholder placed according to its relative position. (The code does not currently correct for its horizontal position along the road)

Note that the current code is merely a means of communicating the intended behaviour; it does not always work as intended. Furthermore, "sideposisjon" can contain far more values than "H" or "V" [41] which have to be handled accordingly.

4.4.5 A Note Regarding Maya Placeholders

During the modelling process, some placeholder boxes are placed throughout the tunnel. These are intended to represent objects within the tunnel. One of them is shown in figure 39 below.

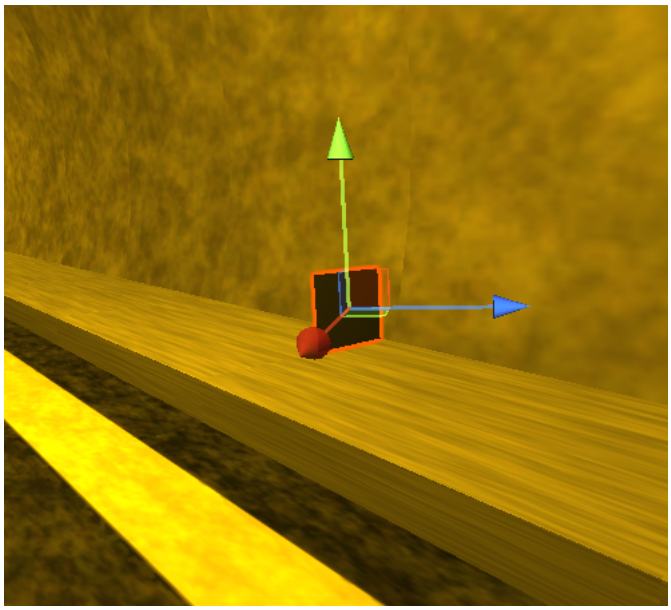


Figure 39: A placeholder placed by Maya

An alternative to the implementation discussed in this chapter could therefore be to simply replace these placeholders with their corresponding prefabs. In the end this solution was not chosen due to the following reasons:

- Most important is the fact that the placeholders are not placed correctly [21]. Meaning some problems with the modelling application would have to be fixed to be able to rely on them.
- Handling it all in Unity gives the game more control over what is in the tunnel, as well as where it is. Seeing as Unity will have to place prefabs either way, it is preferable to enable it to determine an object's position by itself, rather than trusting a separate application to do so.

4.5 Traffic Simulation

The implementation of traffic is a crucial aspect in creating a realistic simulation of tunnels. The main goal was to make the traffic simulation as realistic as possible. In order to achieve this, the NVDB API was utilized as detailed in section 4.3 to dynamically retrieve data about the traffic for selected tunnels. This created a realistic traffic flow that is tailored to each individual tunnel.

It should be noted that a traffic implementation had already been developed in a previous thesis. However, that traffic solution was deemed unsustainable for the finished game and lacked realism. As a result, this new traffic simulation is not an updated version, but rather a brand new implementation.

This section will detail the implementation of traffic and the steps taken to achieve a realistic simulation.

The relevant files are as follows:

```
Assets
└── Scripts
    └── Traffic
        ├── AutoDrive.cs
        ├── GenerateTraffic.cs
        └── Path.cs
```

4.5.1 Waypoints

To implement the traffic simulation, the first step was to create a path for the cars to drive on. Since each tunnel has a different layout, it was important to create a dynamic path that could be set up for each individual tunnel. To accomplish this, the road segments of the chosen tunnel were used to create a series of waypoints that formed a path for the cars. The waypoints were then added to a list in the correct order, starting from the center of the first road, and ending at the center of the last road segment, creating a clear path for the cars to follow.

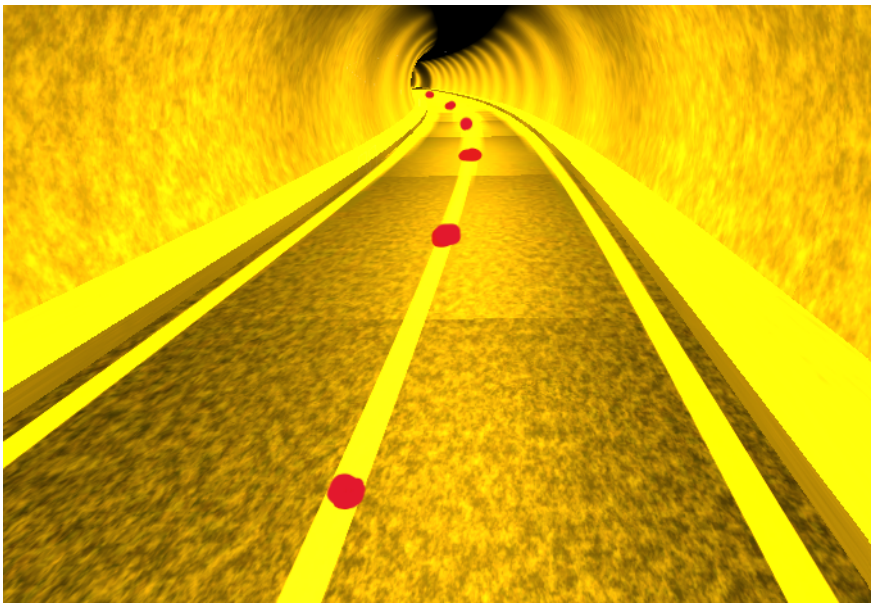


Figure 40: How the waypoints (red dots) were created according to the road segments

After the path was created, the waypoints were initially located at the center of the different road segments, as seen in figure 40. However, in reality, cars do not usually drive in the middle of the road, but would rather stay in their designated driving lanes. Therefore, the waypoints were then moved to the right by subtracting half the length of the road segment, resulting in the cars following the right-hand driving lane. How this was accomplished in code is shown in the listing 31 below.

```
public static void CreatePath(List<Transform> roads)
{
    foreach (Transform road in roads)
    {
        Vector3 node =  road.GetComponent<Renderer>().bounds.center;
        node.x -= road_width/2;
        node.z -= road_width/2;
        Rnodes.Add(node);
    }
}
```

Listing 31: CreatePath method in Path.cs

4.5.2 Automatically Driving Car

After equipping the cars with a dynamic list of waypoints to follow, the next step was to implement their driving behavior. This was achieved by adding Wheel Colliders to the car model, allowing Unity to handle the physics associated with the wheels. However, even though they had the path, they would at this point drive mindlessly without any regards to the path set out for them.

To enable the cars to follow the designated path, a steering behavior was implemented. This involved calculating the difference in position between the car and the next waypoint. However, to obtain the correct positions, it was necessary to work with the local space positions as opposed to the world space positions.

Understanding this distinction is important because the world space positions refer to the position of a gameObject in the overall space of Unity, whereas the local space positions are relative to the position of its parent [4]. The waypoints are all stored as global coordinates, meaning that if a car tried to follow one as is, it would move in parallel to a vector from the global origin point to the waypoint, such as the upper dotted line in figure 41. These global coordinates will therefore have to be converted to local coordinates, effectively moving the origin point to the car. This concept is demonstrated in the same figure 41 below.

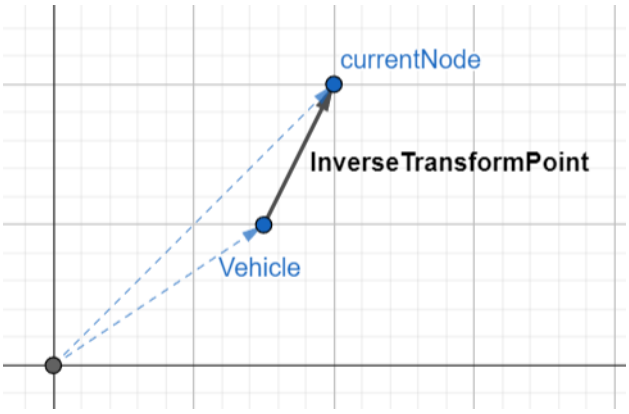


Figure 41: InverseTransformPoint() method used from the car's parent to the current node in the list

This is accomplished in code by retrieving the parent object and calling InverseTransformPoint() which converts the input vector from global to local space relative to the parent.

```
relativePos = transform.parent.InverseTransformPoint(nodes[currentNode]);
```

Listing 32: Relative position in local space

After the calculation of the position difference, the next step was the steering angle. The steering angle only needed to affect the two front wheels, since the two back wheels will automatically follow them.

```
private void Steer()
{
    float nextsteer = (relativePos.x / relativePos.magnitude) * maxSteerAngle;
    frontRightWheel.steerAngle = nextsteer;
    frontLeftWheel.steerAngle = nextsteer;
}
```

Listing 33: Steer method in AutoDrive.cs

To calculate the angle, the x-component of the position vector calculated above is normalised (by dividing by the length of the vector), representing the distance between the car and the waypoint along the x-axis. This gives a normalised value between -1 and 1. The normalized value is then multiplied by a set constant, which is the max steering angle set to 45 degrees. This means that if the next node is at a relative position of 0.8 to the vehicle, the front wheels would steer at an 36% angle to the right. As the name of the constant entails, the max steering angle set to 45 degrees, indicates the biggest angle allowed on the wheels.

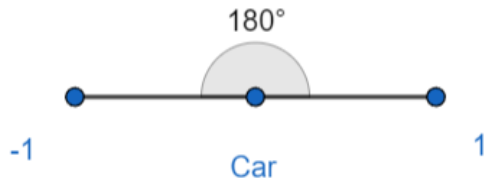


Figure 42: How the steering angle differentiated between left and right

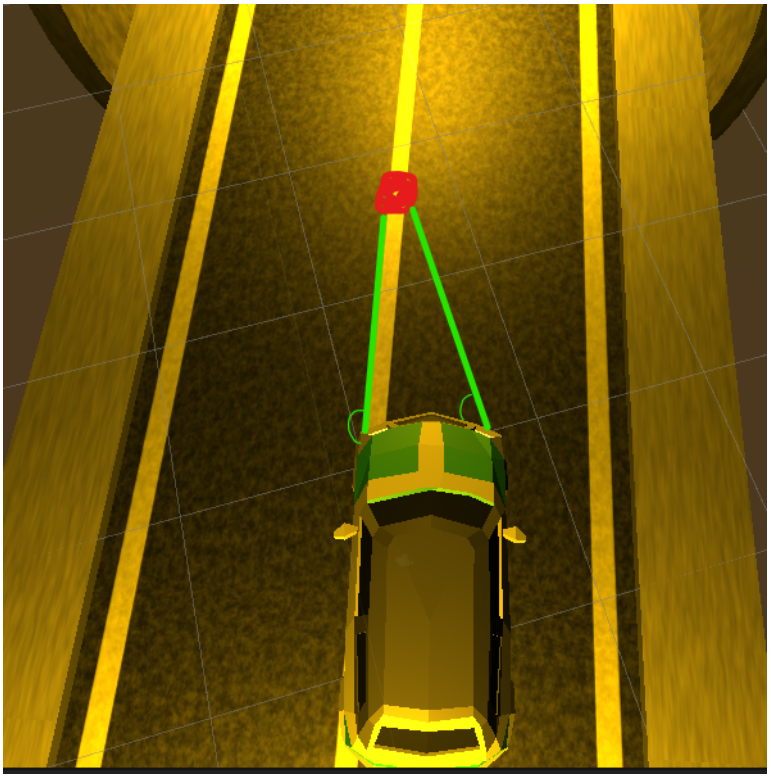


Figure 43: How the wheels would angle according to the next node

To enhance the realism of the traffic flow and ensure smoother driving overall, the cars advance to next waypoint in the list as soon as they come close enough to the current waypoint position, rather than having to hit the exact coordinates indicated by the waypoints.

```
private void NextWayPoint()
{
    if (Vector3.Distance(transform.position, _nodes[currentNode]) < 10f)
    {
        currentNode++;
    }
}
```

Listing 34: NextWayPoint() method in AutoDrive.cs

This dynamic adjustment of the waypoints accounts for the fact that a car for various reasons, may be further to the left or right than intended. Moreover, when the car reaches the last waypoint in the list, the gameObject of the car is automatically destroyed, since it has left the domain of the simulation. This design decision reflects that the game and this simulation are limited to the tunnel only, and provides an efficient way to manage the lifecycle of the cars, whilst also saving processing power.

This approach for driving also provides flexibility in changing the path of the cars whilst they are driving through the tunnels. For instance, if the road is blocked by a fire or heavy traffic causing congestion flooding up the driving lane, the path of the cars can be changed to a different path as needed by calling the SetPath() method.

```
public void SetPath(List<Vector3> nodes)
{
    _nodes = nodes;
}
```

Listing 35: SetPath() method in AutoDrive.cs

The SetPath() method is a crucial part of the traffic system, as it allows for dynamic modifications of the car's path whilst they are driving. To use the method, one would need to call on it with the desired list of nodes, which represents a path of waypoints. This list would then be the car's new path to follow, and the car's internal path would be updated with the new waypoints. This allows the traffic simulation to adapt to changing environments.

4.5.3 Speed Limit

Now that the cars could drive and follow a path, it was decided to make the tunnel have a "legal system". The driving behavior of the cars was extended to be able to follow a speed limit for the tunnel that they are in. To accomplish this, the current speed was calculated by reading real time values from the Wheel Colliders of the car, which allowed for dynamic updates to a variable representing the current speed.

```
currentSpeed = 2 * Mathf.PI * wheel.radius * wheel.RPM * 60/1000;
```

Listing 36: How the current speed is calculated

To calculate the current speed of a car's Wheel Colliders, the formula (36) computes the distance covered by the Wheel Colliders in one minute by multiplying the circumference of the wheels (2*pi*radius) by the Rotations per minute (RPM), before converting the result to kilometers per hour. After doing so, the cars need a target speed so that they can adjust their current speed accordingly. This is done by retrieving the NVDB "egenskap" for the speed limit, and setting it as the target speed for the cars.

```
targetSpeed = float.Parse(TNvdbData.Data.egendefinert.fartsgrense.GetEgenskap(
    "Fartsgrense").verdi);
```

Listing 37: Retrieving NVDB "egenskap" for speed limit

The speed limit is measured in kilometers per hour, so that is why the calculated current speed is also converted to the same format. The speed limit is dynamically retrieved, so if two different tunnels have different speed limits, the cars will adjust accordingly.

```
private void Drive()
{
    if (currentSpeed < targetSpeed)
    {
        float torque = (targetSpeed - currentSpeed) * maxTorque;
        ApplyMotorTorque(torque);
    }
    else
    {
        float brake = (currentSpeed - targetSpeed) * maxBrakeTorque;
        ApplyBrakeTorque(brake);
    }
}
```

Listing 38: Controlling the speed limit

The speed limit for each tunnel is defined as a constant value, as shown in listing 37. It is then used to determine if the car should continue to accelerate. If the car's current speed is under the speed limit, a torque is applied to the Wheel Colliders, causing the car to accelerate faster. The torque value is proportional to how fast the car's speed is, and is calculated such that the faster the car is moving, the less torque is applied to the wheels.

Similarly, if the current speed of the car exceeds the speed limit, the car will decelerate to match the speed limit. If the car needs to slow down, the brake is calculated in a similar fashion as the torque, but with a faster braking rate when the car is going faster until the car's current speed reaches the speed limit again. These approaches ensure that the driving behavior is compliant with the tunnel's legal system and improves the simulation, as cars now drive according to the actual speed limit of the tunnel.

In order to maintain a clean and organized structure for the driving behavior script (AutoDrive.cs), it was designed to separate the different functionalities into their own methods. This improves readability and ease of maintenance, as each method was only responsible for one specific task. By grouping the different functionalities in this way, it also makes it easier to modify and add new features without impacting other areas of the script. As a result, the script became flexible and structured, allowing for easier development going forward.

```
private void FixedUpdate()
{
    Steer();
    Drive();
    NextWayPoint();
}
```

Listing 39: FixedUpdate() method in AutoDrive.cs

4.5.4 Spawning the Traffic

The next step after implementing the driving behavior was to spawn traffic in the tunnels. To create a more realistic effect, the spawn point is at the first road segment, meaning inside the exit collider (see section 4.2.5). This creates the illusion that the cars are driving into the tunnel from the outside, rather than appearing inside the tunnel. This adds to the immersiveness of the traffic and the gaming experience as a whole.

Figure 44: A car spawning inside one of the exit colliders in the tunnel

To ensure that the cars spawn correctly and avoid any potential issues with the tunnel architecture, the spawn point of the vehicles is set to dynamically be above the road (as seen in figure 44). This is necessary as the tunnel segments might not always be leveled, meaning a spawn location that works within one tunnel could cause the car to spawn inside or even under the road inside of another tunnel. By setting the spawn point dynamically to be above the road instead, the cars are guaranteed to spawn in a safe location.

Despite its initial success, this method of spawning the cars would cause a recurring problem where the cars would fall straight through the road with no apparent collision between the two objects. This happened due to Unity only checking for collisions at specific intervals, and due to the high speed at which the cars were falling, there was a chance that the collision between the two objects would not be registered.

To prevent the aforementioned issue, the solution was setting the collision detection mode of the car's Rigidbody component to continuous rather than discrete. With continuous collision detection the physics engine in Unity will check for collisions with the object more frequently, preventing the cars from falling through the road. However, using continuous collision detection comes at a cost of increased computations for the physics engine of the game. Nevertheless, it is recommended for the best results to use continuous collision detection for fast moving objects to ensure reliable collision detection [31].

Recall from figure 44 above that the cars are tilted downwards upon spawning. This is due to the initial rotation that is set when they first spawn. To ensure the right orientation on spawn, the cars are spawning "looking" at the next waypoint in their path, as shown in listing 40. This approach resolved an important issue with the spawning since not all road segments were aligned with the same angle in the world space. With a static rotation, the cars would spawn with suboptimal rotations, and in the worst case, the cars would have to turn around 180 degrees to reach the next waypoint. This caused congestion and often resulted in the cars being stuck when the tunnel's road segment was too small. By dynamically aligning the car's rotation with the road segment, the problem was solved.

```
rot = Quaternion.LookRotation(nodes[1] - placement);
```

Listing 40: The local rotation set to "look" to the next waypoint

To ensure that the cars spawning are scaled correctly to the fit the driving lanes, they are instantiated as a child of the spawn point road segment. This parenting relationship allows the cars to inherit the scale of the road segment, as explained in section 4.2.4. By instantiating the cars this way, it is ensured that the cars are scaled correctly to fit within the driving lanes.

4.5.5 Bi-Directional vs Uni-Directional Tunnels

At this point in the implementation, the cars were able to spawn correctly inside any tunnel, and follow the right lane to the end of the tunnel. The left lane on the other hand, was as of yet completely unused. It was time to further enhance the realism of the traffic simulation. It is important to recognize that a tunnel can be both bi-directional and uni-directional. The former meaning that traffic flows in both directions simultaneously, and the latter meaning both lanes are driving in the same direction. Therefore, it was important to consider both types of traffic in order to more accurately reflect real traffic.

To accommodate for the two types of traffic occupying two driving lanes, a new list of waypoints needed to be created for the left lane. To accomplish this, the CreatePath() method in Path.cs was modified to include the creation of "Lnodes", in addition to the right lane waypoints (Rnodes) that were previously defined in section 4.5.1.

```
public static void CreatePath(List<Transform> roads)
{
    foreach (Transform road in roads)
    {
        var road_width = TMath.roadWidth;
        Vector3 node = road.GetComponent<Renderer>().bounds.center;
        node.x += road_width / 2;
        node.z += road_width / 2;

        Lnodes.Add(node);
    }
}
```

Listing 41: CreatePath() method in Path.cs

The main difference from Rnodes, is that Lnodes's waypoints are shifted to the left lane instead. Specifically, the x and z positions are moved to the left by half the road width. Resulting in the cars following the left lane.

To implement the different types of traffic, information would have to be retrieved from NVDB about the tunnel's direction of travel. This was originally thought to be a simple task, but looking through NVDB proved otherwise. The team was not able to find any information about the direction of travel within the tunnel tube, the tunnel itself nor the road object. An object called "Permitted driving direction (Tillat kjøreretning)" was found, but did not prove very helpful as there currently is not a single instance of this object within all of NVDB [40]. An alternative solution was needed.

It was decided to look at the amount of parallel tubes the tunnel consisted off, which was readily available within NVDB. By knowing the amount, one could assume the traffic directions. In most cases, a two-tube tunnel would mean uni-directional traffic within each tube, as the point of a tunnel consisting of two tubes is to have different uni-directional traffic in two separate tubes to optimize traffic flow. The same assumption works as well for most one-tube tunnels.

Seeing as NVDB did not give information about the traffic direction directly, the implementation of bi-directional vs uni-directional traffic is working around this "99%" accurate solution.

```
tubeAmount = TNvdbData.Data.relasjoner.tunnel.GetEgenskap("Antall parallelle
    hovedløp").verdi
```

Listing 42: Retrieving the number of tubes from NVDB

Now that it is clear which direction the cars are driving in, it would also be interesting to know how much traffic is within each lane relative to the other. Once again, searching through NVDB proved fruitless. An object called "Traffic amount, driving lane (Trafikkmengde, kjørefelt)" was found [39], but there is currently only a single instance of this object (located somewhere in Lillestrøm).

The solution is therefore to create an approximation based on logic and common knowledge rather than NVDB. When a car is instantiated, a random value ranging from 0 to 100 is assigned to each car. This value is then used to regulate the traffic in the tunnel, achieving a more dynamic and unpredictable traffic flow.

```
if (tubeAmount == 1)
{
    if(rnd>=50)
    {
        nodes = Lnodes.ToList();
        nodes.Reverse();
    }
}
```

Listing 43: Direction logic for one-tube tunnels

In the case of a single tunnel tube, the traffic is bi-directional. However, since NVDB lacks reliable information about the amount of traffic in each lane, a simple 50/50 split is used for traffic spawning at each end. This is based on the assumption that the average traffic flow in each direction should be about the same. If time-of-day patterns were incorporated into the simulation, the traffic in each lane might display more variations.

When the random value is greater than or equal to 50, the cars will spawn in the opposite end of the tunnel. This means Lnodes will be reversed, causing the cars to drive in the opposite lane and direction as compared to the usual cars. For the reversed cars, it was also necessary to adjust the spawn point to be on the last road segment, and rotated towards the next-to-last waypoint.

```
if (tubeAmount == 2)
{
    if (rnd >= 70)
    {
        nodes = Lnodes;
    }
}
```

Listing 44: Direction logic for two-tube tunnels

In the case of two-tube tunnels, the traffic is uni-directional, but since NVDB does not provide information on the amount of traffic in each driving lane, a solution was needed to regulate the traffic flow. It was recognized that the right lane tends to be more congested than the left one, so the implementation has 70% of the spawned cars use the right lane, and the remaining 30% use the left lane.

To accommodate the traffic in two-tube tunnels, the Lnodes's path is used for the 30% of cars whilst Rnodes's path is used for the other 70%. This adjustment ensures uni-directional traffic flow in the tunnel.

4.5.6 Traffic Frequency

To create a more realistic traffic simulation, the traffic frequency is determined based on the "Årsdøgntrafikk (ÅDT)" value obtained from NVDB. ÅDT represents the average amount of traffic passing through a certain point each day within a year [20]. By using the ÅDT value specific to the chosen tunnel, the simulation approximates the real traffic frequency in that tunnel. This approach results in a more accurate representation of the traffic, enchaining the overall realism of the simulation.

```
float trafficPerSecond = float.Parse(TNvdbData.Data.egendefinert.trafikkmengde
    .GetEgenskap("ÅDT, total").verdi) / (24 * 60 * 60);
```

Listing 45: Gathered ÅDT value

The traffic frequency is then further fine-tuned by converting the ÅDT to seconds from days, now representing the amount of cars driving in the tunnel per second as seen in listing 45. After doing so the per-second traffic count is multiplied with the elapsed time since the last frame update, which is then used to calculate the number of cars to spawn.

```
void Update()
{
    frameTime = Time.deltaTime;
    spawnVehicle += frameTime * trafficPerSecond;
    if (spawnVehicle >= 1f)
    {
        Traffic();
        spawnVehicle = 0;
    }
}
```

Listing 46: Implementing the ÅDT traffic frequency

These values are added together with every frame update, until they reach a value of 1, at which point a car will spawn in the tunnel, and the process value will be reset to 0 (note that always resetting to 0 intentionally results in some variation in the spawning intervals between each vehicle). This simple yet effective implementation can be seen in listing 46.

By implementing the traffic frequency as described, the amount of traffic that spawns over the course of a day, should be roughly equivalent to the original ÅDT "egenskap" for the selected tunnel. Thus, tunnels with high density traffic throughout the day, i.e. a high ÅDT, will have a lot of vehicles driving through. On the contrary, tunnels with low density traffic, i.e. a low ÅDT, will have less frequently spawned cars, and it will take some time before the player sees a car in the game. This allows for a realistic representation of the traffic in the simulation.

4.5.7 New Vehicles in the Tunnels

Cars are not the only vehicles driving inside of tunnels. As one of the main goals of this thesis was to contribute to making a more realistic tunnel simulation, it seemed natural to include other vehicles

as well. New prefabs were made in a similar fashion as the cars (see 4.1.7). To achieve this goal, an "egenskap" is once again retrieved from NVDB:

```
avgLargeVehicles = (TNvdbData.Data.egendefinert.trafikkmengde.GetEgenskap("ÅDT
    , andel lange kjøretøy").verdi) / 100;
```

Listing 47: Share of ÅDT data consisting of large vehicles in %

"ÅDT, share of long vehicles (ÅDT, andel lange kjøretøy)" indicates the amount (in percentage) of the ÅDT traffic that is defined as large vehicles. Large vehicles as defined within NVDB, are vehicles longer than 5.5 meters [39]. The vehicles implemented in the traffic simulation are a tank truck and a bus.

```
private GameObject VehicleSelection()
{
    float rnd = UnityEngine.Random.Range(0f, 1f);
    if (rnd < avgLargeVehicles)
    {
        int vehicleType = UnityEngine.Random.Range(0, 2);
        switch (vehicleType)
        {
            case 0:
                return drivableBusPrefab;
            default:
                return drivableTankTruckPrefab;
        }
    }
    else
    {
        return drivableCarPrefab;
    }
```

Listing 48: Vehilce selection method

The VehicleSelection() method shown above 48, is in charge of selecting which vehicle to spawn based on the average percentage of big vehicles in the tunnel. This percentage is determined by the "egenskap" value of the tunnel's "ÅDT, andel lange kjøretøy", and could vary depending on the tunnel. For example, a tunnel in an urban area may have a lower percentage of large vehicles compared to a tunnel in a rural area. This adds more variety to the traffic flow.

If the random value is lower than the avgBigVehicle percentage, it will again select a random number that will decide if the vehicle that will spawn is a bus or a tank truck. If the first random number is not lower than the avgLargeVehicle percentage, then a car will spawn. Note that when using the "Random.Range(min,max)" method, the resulting number is generated within the range of min(inclusive) and max(exclusive). Therefore, when using "Random.Range(0,2)" as in listing 48, the returned number can possibly only be an integer of 0 or 1.

By adding new prefabs into the switch case, it can easily be expanded to allow for more large vehicles in the future. Although the switch case may seem redundant with only two vehicles to choose from, it provides a framework for handling multiple prefabs in a scalable way.

4.5.8 Data Assumptions

Since the traffic implementation makes use of some data from NVDB, this section will detail some of the assumptions made in gathering said data and their potential shortcomings.

- **ÅDT:** Any ÅDT data used within the code is part of a larger object called "trafikkmengde". A "trafikkmengde" will occupy a certain range of a "veglenke" in the same way as a tunnel. (See section 4.4.1 for more details.) There is however no guarantee that tunnel and "trafikkmengde" occupy the same coordinates. There may therefore be more than one "trafikkmengde" within the same tunnel. The data collected is only from the first "trafikkmengde" overlapping with the tunnel. Doing so is not expected to cause many issues seeing as the amount of traffic will be the same throughout most tunnels. More accurate numbers could however be achieved through an average of all "traffic volumes (trafikkmengder)" overlapping with the tunnel.

  More noteworthy is the case where a "trafikkmengde" stretches outside the tunnel, as demonstrated in figure 45.

Figure 45: A "trafikkmengde" (blue) that stretches outside the tunnel in both directions. The green line indicates the length of the tunnel

  In such cases, the "trafikkmengde" could theoretically stretch beyond an intersection. This is not known to happen, but if it does happen, then it could skew the ÅDT values depending on how the "trafikkmengde" in the other road compares to the tunnel. If this turns out to be a problem, one could instead retrieve a "trafikkmengde" strictly inside the tunnel, if such an object is available.

- **Speed Limit:** The speed limit is retrieved from a "fartsgrense" object. Just as the "trafikkmengde", this object occupies a range of a "veglenke" and the code will retrieve the first one that overlaps with the tunnel. Meaning the code is operating on the assumption that the speed limit is constant throughout the tunnel. If the speed limit *does* change, then more "fartsgrense" objects will need to be retrieved from the tunnel.

5 Evaluation

5.1 Drivable Vehicles

One of the existing vehicle models was chosen and modified to help satisfy the goal of a drivable vehicle. Driving the vehicle works mostly as intended, allowing the player to turn, accelerate, and break as a normal vehicle would. The player is able to enter the vehicle and exit it as one would expect.

5.1.1 Handling

The vehicles' handling can be unstable especially at higher speeds. Turning is also a challenge, as they have a tendency to want to flip over, rendering them useless. It is recommended that this is looked at fairly early as this can be a headache later. For instance, if roundabouts are implemented, it is not guaranteed that the cars will be able to handle the turns, instead unintentionally throwing themselves off course. An attempt to lower the center of mass has been made in order to make them more stable (as per Unity's documentation [30]) however it does not seem to have solved the issue.

If the vehicle manages to get on top of the road shoulder, they have a difficult time backing out and back on the road properly. The problem here seems to be more on the modelling rather than the car as the road shoulders are likely scaled to be taller than in reality, making the car unable to back up on the road again.

5.1.2 Collisions

Collisions is also something that is not handled well, both regarding other vehicles and gameObjects. The vehicles have a tendency to act weightless and fly around when they are hit hard enough. This is despite the vehicle mass (measured in kilograms [32]) in the Rigidbody set to over 1000, which one would think would make it more difficult to get the vehicle flying. This breaks the realism of the game.

5.1.3 Player Placement

The placement of the player in the car generally works well, although this can be inconsistent. At times, the Player gameObject is rotated incorrectly, which can cause the player capsule body to be visible from outside of the vehicle. Though it is not an issue for single player, it will be an issue to address if multiplayer is ever introduced.

The player as is should not really be inside the car to begin with. The only reason for shoving it in there rather than despawning it is to have the exit colliders register the player driven car, thus keeping the functionality of the player going inside the exit collider, stopping their movement and ending the game. In the future however, it may be preferable to have a car model with an actual inside that the player can enter and interact with.

5.2 Rescaling the Tunnel

There were two primary objectives for this section:

- The primary objective was to scale the tunnels in such a way that they would look more convincing.
- The second was to enable every object within the tunnel to scale dynamically, so that rescaling the tunnel again in the future would be much easier.

The second objective has been achieved by making use of two major changes:

- A constant has been introduced to the code, allowing not only the tunnel itself to scale, but also enabling other objects to adapt to differing tunnel sizes through the code.
- Many objects have been repositioned within the hierarchy, thus enabling them to scale according to their parent object without the need for any additional code.

It should be noted however that some of the numbers could still use some tuning. The exit colliders for instance, are not able to cover the entire opening of the tunnel at massive tunnel sizes. The lights are also poorly adjusted for small or large tunnel sizes. These issues however, are not considered as severe, as there should be little need for tunnels of a scale drastically differing from the current implementation.

As for the first objective; although the resulting tunnels do look marginally different from their original form, they still have some of the same problems as they did before. The exact issue has been difficult to pin down, but it appears to lie within the modelling of the tunnels as will be discussed in the following section.

5.2.1 Shortcomings of the Tunnel Models

During the rescaling, it was naïvely assumed that the main issue was the tunnels being too big compared to the player. It was in other words assumed that all parts of the tunnel (such as roads and walls) were correctly scaled relative to each other. This has now been determined to be false, thus meaning simply scaling the tunnel and player relative to each other will not fix the underlying issue.

The biggest issue is the fact that all tunnels use the same texture with two driving lanes for the road, further assuming there is nothing separating those two lanes. Countless tunnels will in reality deviate from this approach, thus meaning the road as seen in the game is highly misleading. Figures 46 and 47 below demonstrate this using Åsnuttunnelen as an example.

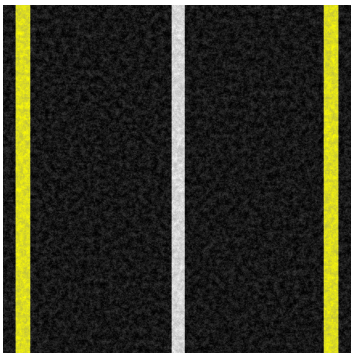


Figure 46: The texture currently used for all roads

Figure 47: The real Åsnuttunnelen [7]

Note that the real tunnel has a fence separating the two driving lanes. Since the modelling of tunnels does not take such elements into accounts, the resulting tunnel model will have to stretch the road texture in order to fill the space that should be occupied by the fence, thus resulting in the driving lanes being wider than they should be. This becomes a problem when scaling the cars, as the cars are scaled based on how well they fit within the lanes. The player has by extension also been scaled incorrectly, as it was scaled based on the already incorrectly scaled car.

To conclude this section, the roads within the tunnels are *not* a reliable frame of reference, thus meaning either a different method is needed to scale its objects or the tunnel models need to account for what the real tunnel roads look like.

5.2.2 Old Placement Code

As discussed in section 4.4, this thesis has created a system for placing objects within the tunnel according to their real location. There is however some old code left over from previous projects that will attempt to place some objects of its own. Said code does not use any data from NVDB nor does it take the scale of the tunnel into account. This results in some objects being misplaced, as shown in the figure below.

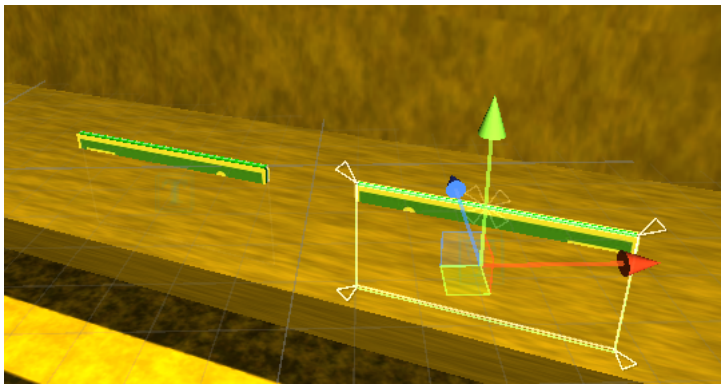


Figure 48: Exit signs placed inside the ground

The exit signs in the figure are simply placed throughout the tunnel at fixed intervals and a certain distance above the ground. Since the tunnels now are larger than they used to be, this fixed height is now much too short, resulting in the exit signs being buried below ground. With both the NVDB data as well as the tunnel size available, this should be relatively easy to fix.

5.3 Integration of NVDB

One of the goals of this thesis was to allow the game to adapt according to data retrieved from NVDB. This has successfully been implemented in two parts. The first part was to expand the procedural modelling with a solution for retrieving and storing the data. The second part was then to create an API within the game that grants access to all of the stored data.

Hence the groundwork has been laid for future expansions to the project to be able to use any relevant data found within NVDB. Doing so will inevitably require further expanding the code however. The team remains optimistic that this should be relatively easy to do, but only time will tell how well it holds up in practise.

5.3.1 Infinite Loops

When running the game, Unity will raise some concerns regarding the potential of infinite loops within the data structure as seen in figure 49. This is due to the class EgenskapObjekt containing a list of Egenskaper which themselves contain a list of EgenskapObjekt. This results in a recursive loop where all of these fields will be filled with null values as the game runs out of data to store in them. The loop does luckily eventually end after 10 iterations, as indicated by the aforementioned warnings.

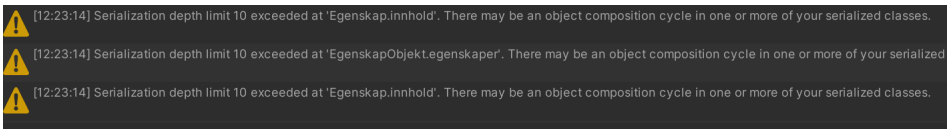


Figure 49: Unity warnings regarding the data structure

In practise, this should never result in any data loss seeing as the data from NVDB will likely never have a depth of 10. It is however an optimisation issue as a lot of time is wasted running this entire process 10 times. Should it turn out to be too much of a problem, then some light reworking of the C# data structure will be needed, such as adding more classes to avoid the loop.

5.4 Object Placement

Ultimately this became a late addition to the project as it necessitated other features being implemented first. As a result, the team was not able to place many differing objects as originally desired. There has nevertheless been successfully implemented a method of placing "skiltpunkt" within the tunnel which can be generalised for other objects.

There is still room to fine-tune the placement of "skiltpunkt" as well as creating a proper model rather than the current placeholder. Note that all signs have their own corresponding code. The official handbook may prove useful if the objects are desired to make use of the correct sign plates [38].

5.4.1 Order of Execution

Section 4.4.3 discussed the use of parents and children in the hierarchy to enforce a certain order of calls to their respective Awake() methods. They were placed under the assumption that a parents Awake() call happens before its children have their Awake() methods executed. This assumption however does not appear to hold.

The official documentation [29] guarantees that all Awake() methods will be executed before any Start() methods. It does not however guarantee any order when executing the Awake() and Start() methods. Recall the following structure within the hierarchy:

```
Nvdb
└── NvdbPlace
    └── SkiltPunkt
```

Nvdb uses the Awake() method to instantiate itself. NvdbPlace cannot instantiate itself without Nvdb and would therefore use the Start() method to do so. This creates a problem for SkiltPunkt since it needs NvdbPlace to be instantiated before it can place anything. SkiltPunkt currently does so within the Start() method whilst Nvdb and NvdbPlace use their Awake() methods. This could in theory lead to NvdbPlace being executed before Nvdb, preventing all placement from working correctly.

It should be noted that although Unity does not guarantee any order, that does not mean there is no order. The code has been tested multiple times and Nvdb has never failed to be called first. It seems whatever ordering the current version of Unity is using favours the intended execution order. This has in other words not been a problem as of yet.

Nevertheless, this structure should receive a light rework so as to ensure they always execute in the correct order. Although it defeats some of the purpose of this structure, the easiest fix would be to create new methods for a different class to call. As an example, all children of NvdbPlace could implement an interface with a Place() method, enabling NvdbPlace to loop through its children and call their respective Place() methods.

5.5 Traffic Simulation

The main objective of the traffic implementation was to create a sustainable and realistic simulation of traffic, that could dynamically adapt to the selected tunnel's attributes and values.

5.5.1 Waypoints

The created path for the cars works fine for most tunnels, but a problem occurs with tunnels that curve a lot from the world center position (as a tunnel like figure 50).

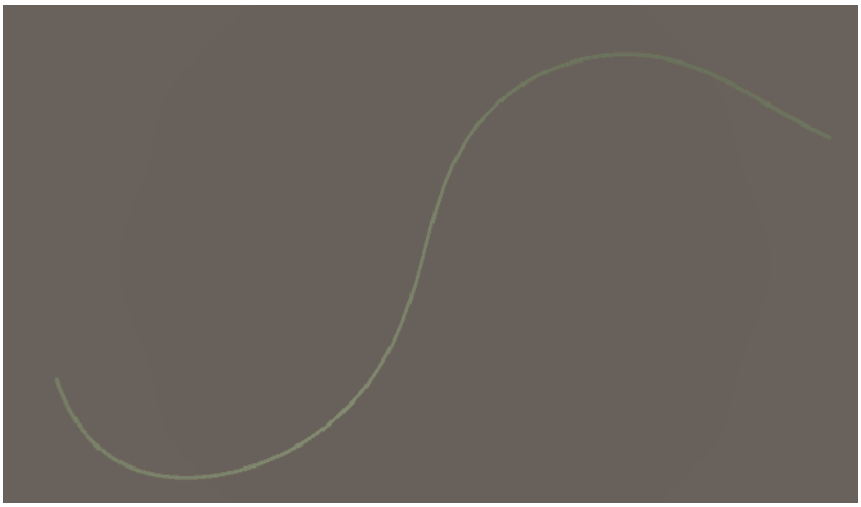


Figure 50: Skatestraumtunnelen in Vestland county as seen from above in Unity

The problem lies with how the waypoints are calculated. As mentioned in section 4.5.1, the waypoints are calculated based on the bounds of the road segments. The problem with calculating them like this, is that the bounds of a road does not rotate with the road, but is instead always aligned with the unit vectors of the world (see figure 51).

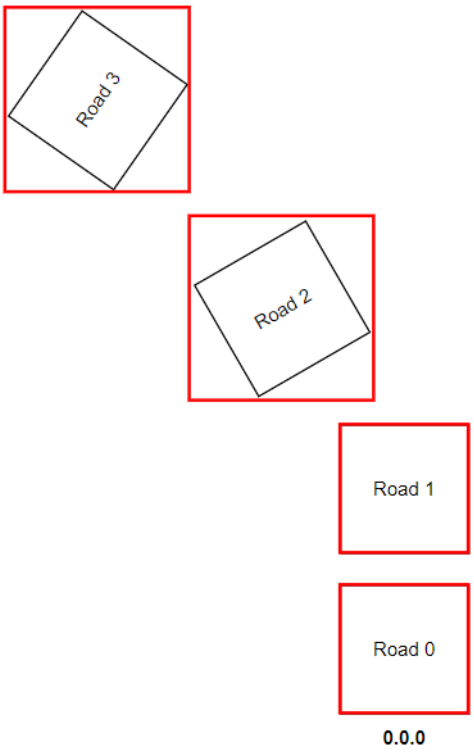


Figure 51: The bounds boxes(red boxes) around differently rotated road segments

As shown in the figure, not only are the bounds not rotating with the road segments, but they are also expanding to fit the whole segment. This inconvenience results in the waypoints being calculated unfavourably for the traffic implementation. Figure 52 shows what the road waypoints approximately look like in Skatstraumtunnelen (50).

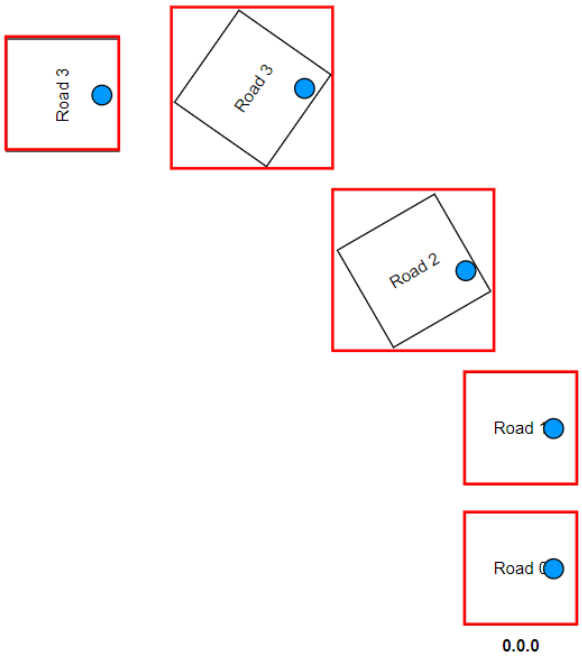


Figure 52: The bounds boxes with the waypoints (blue) after calculation

As can be seen in the figure, in drastic cases where the tunnel road segments turn at a 90 degree angle or more from the world origin coordinates, the waypoints are significantly misplaced with regards to the driving lane. If there is a tunnel that turns 180 degree, the cars could even switch lanes from right to left, which is evidently not the preferred behaviour.

5.5.2 Bi-Directional vs Uni-Directional

One-tube and two-tubes tunnels, do not always indicate bi-directional and uni-directional traffic respectively, as mentioned in section 4.5.5. The most common issue with this method of assuming traffic directions, is when there is only one driving lane in the tunnel, as the tunnel in figure 53.

Figure 53: Dørefjelltunnelen with only one driving lane [8]

In this case since it is a one-tube tunnel, the traffic implementation will assume this tunnel has bi-directional traffic. Resulting in some cars breaking traffic laws and colliding into each other.

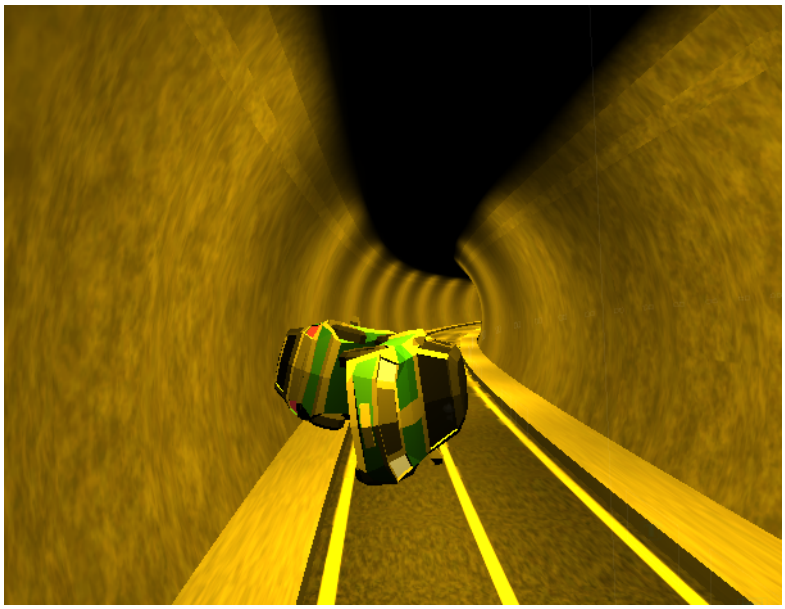


Figure 54: Moments after collision disaster in Dørefjelltunnelen

5.5.3 Traffic Frequency Inconsistencies

The traffic spawning pattern is fixed and does not change randomly over time as real-time traffic would. The traffic spawns in a deterministic fashion according to the average traffic (ÅDT) per second. While this gives the right amount of cars through out the day, it does not accurately reflect the aspect of randomness of real life traffic.

In reality, traffic is influenced by a variety of different factors, such as time of day, weather conditions, even holidays or events. Therefore, a more realistic traffic simulation should be less deterministic and should incorporate a certain randomness into the spawning to more closely resemblance real life traffic patterns.

5.5.4 Spawning the Traffic

In highly congested tunnels, the vehicles have a high chance to spawn inside of or on top of each other. This is especially an issue with the large vehicles that take up a lot of spawn space, and in uni-directional tunnels, where the vehicles spawn on the same spawn point, resulting in a higher change of spawning on top each other.

A solution to this might be to check if the spawning point road segment is clear before spawning. However, this would make the traffic frequency less realistic than it is now, since in highly congested tunnels, the vehicles have to wait to spawn, making the vehicles spawn less frequently overall. Another potential solution could be to make a queue system, where vehicles are put into a list acting as a virtual queue instead of waiting to spawn. So in theory, it's as if the vehicles are standing in a queue outside of the tunnel, but in reality they haven't spawned in yet. However, this implementation would require additional consideration, such as the vehicles would need to be in a queue after spawning.

Figure 55: Efficient, though unrealistic traffic

5.5.5 Longer Vehicles

The trucks and the buses have longer wheelbases than the cars do, which causes them to be less maneuverable at high speeds. Additionally, their weight distribution is also more uneven than that of a car. This as well can cause instability during sudden changes in direction and at high speeds.

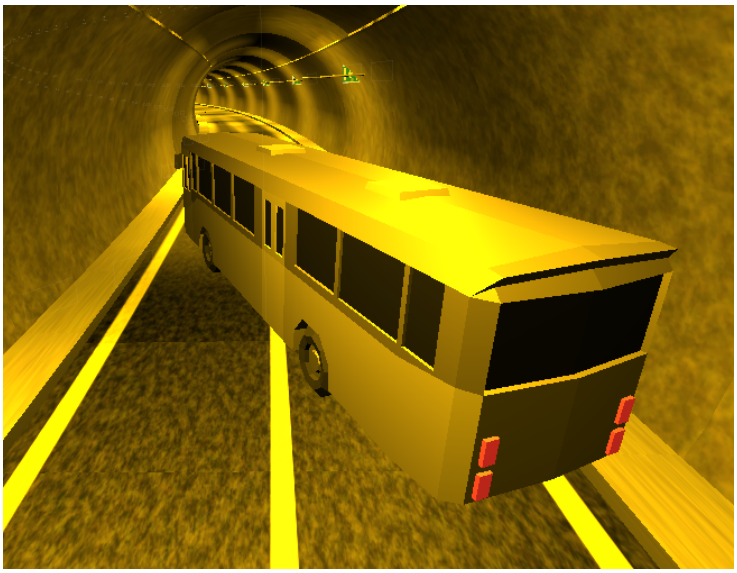


Figure 56: A bus going into a drift trying to turn at high speeds

Overall, this makes the large vehicles struggle with turning and navigating curves, as they often end up in a drifting state. This results in them taking up the whole road, as seen in figure 56, which can easily lead to collision with other vehicles and the large vehicles tipping over.

5.6 Miscellaneous Issues

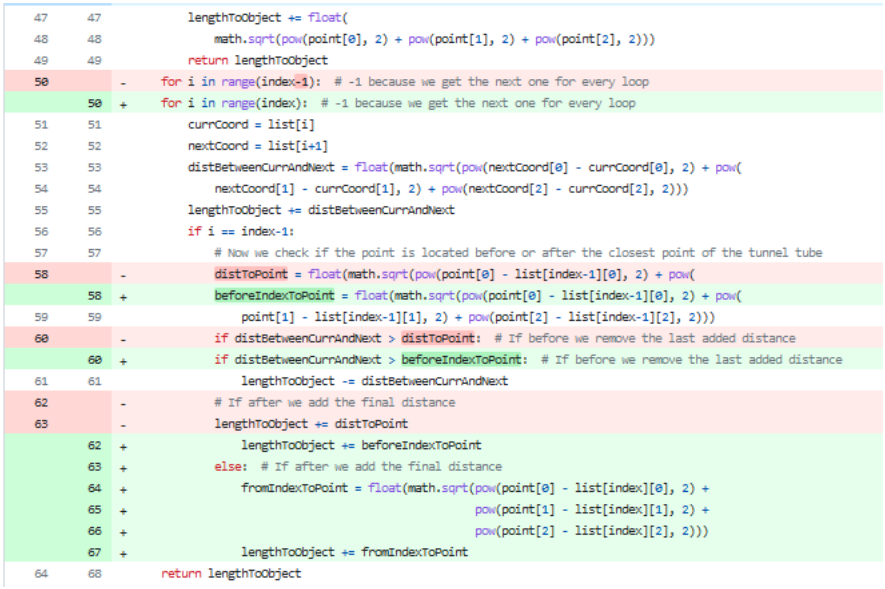
5.6.1 findLengthToObject

findLengthToObject is a function found within the procedural modelling side of the project. The file in question would be nvdb.py. The reason this function gathered attention was due to some unnerving results when modelling a tunnel. During the modelling process, a function known as checkIfLengthIsCorrect will be called. The purpose of said function is to compare the actual length of the tunnel (provided from NVDB) against the computed length between the coordinates retrieved from NVDB using the findLengthToObject function. This check however would often claim that the difference was "not ok", and in fact several hundred meters in length. Some example output is shown in the figure below.

```
Checking if total length between coordinates are correct...
    Result: Not Ok
    Difference: 347.99451792071113m
```

Figure 57: The modelling process overestimating the difference between the calculated and actual length of the tunnel tube. (Output generated using tunnel tube 79608890)

The cause of the problem was found and rectified, using the code changes shown in the following figure.



```
47  47          lengthToObject += float(
48  48              math.sqrt(pow(point[0], 2) + pow(point[1], 2) + pow(point[2], 2)))
49  49          return lengthToObject
50      -   for i in range(index-1):  # -1 because we get the next one for every loop
        50  +   for i in range(index):  # -1 because we get the next one for every loop
51  51          currCoord = list[i]
52  52          nextCoord = list[i+1]
53  53          distBetweenCurrAndNext = float(math.sqrt(pow(nextCoord[0] - currCoord[0], 2) + pow(
54  54              nextCoord[1] - currCoord[1], 2) + pow(nextCoord[2] - currCoord[2], 2)))
55  55          lengthToObject += distBetweenCurrAndNext
56  56          if i == index-1:
57  57              # Now we check if the point is located before or after the closest point of the tunnel tube
58      -           distToPoint = float(math.sqrt(pow(point[0] - list[index-1][0], 2) + pow(
        58  +       beforeIndexToPoint = float(math.sqrt(pow(point[0] - list[index-1][0], 2) + pow(
59  59                  point[1] - list[index-1][1], 2) + pow(point[2] - list[index-1][2], 2)))
60      -           if distBetweenCurrAndNext > distToPoint:  # If before we remove the last added distance
        60  +       if distBetweenCurrAndNext > beforeIndexToPoint:  # If before we remove the last added distance
61  61                  lengthToObject -= distBetweenCurrAndNext
62      -           # If after we add the final distance
63      -           lengthToObject += distToPoint
        62  +           lengthToObject += beforeIndexToPoint
        63  +       else:  # If after we add the final distance
        64  +           fromIndexToPoint = float(math.sqrt(pow(point[0] - list[index][0], 2) +
        65  +                                             pow(point[1] - list[index][1], 2) +
        66  +                                             pow(point[2] - list[index][2], 2)))
        67  +           lengthToObject += fromIndexToPoint
64  68      return lengthToObject
```

Figure 58: The changes made to findLengthToObject(). Old code is shown in red, whilst the fix is shown in green. (Code can be found within commit dccdf1e on GitHub)

The main problem was on line 50. The code is intended to loop through the list of points and calculate the length between the first and second point, then the second and third point, and so on. Since the range function in python is non-inclusive, the final comparison would be between index-2 and index-1, rather than the intended index-1 and index. In other words, the final point within the tunnel would not be considered in the calculation. This is especially severe for straight tunnels where the final point may be placed far away from the preceding point seeing as very few points are needed to accurately represent the tunnel shape.

After changing the code, the reported difference was smaller and more tunnels reported an "ok" difference. (It should be noted that whether a difference is "ok" or not is determined by a constant within config.py known as MAX_ACCEPTABLE_LENGTH_DEFICIENCY, which appears to be a magic number.)

The problem appeared to be fixed, but it was later discovered that the changes had broken the modelling process. It seemed the broken code would not stretch each segment of the tunnel far enough, resulting in a very short tunnel as seen in figure 59.

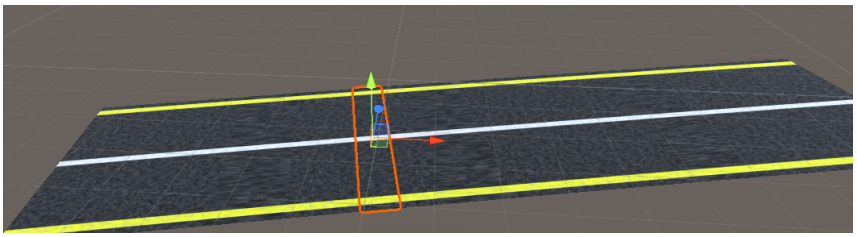


Figure 59: Bergeland tunnel in compact form with a much too short road segment highlighted. (Note that walls and such have been modeled in the same incorrect way. They have simply been left out of the figure to better illustrate the problem.)

The code has therefore been reverted back to its original state. Thus meaning the modelling is once again operational despite findLengthToObject() not functioning correctly. It is speculated that this may be due to the rest of the code somehow correcting for the error. By the time this problem was discovered, there was not enough time for much work beyond reverting the change. Although the function is not known to cause any issues beyond faulty logs, it may nevertheless warrant further investigation. There *are* tunnels which crash the application upon modelling, and this *could* be part of the reason why.

5.6.2 GetWidthAndLength

In the script TMath.cs, there is a method called GetWidthAndLength(). It is intended to be used to retrieve the width and length of the input gameObject. When the method is done calculating, it sets the variables roadWidth, roadLength, widthVec and lengthVec (widthVec and lengthVec being of type Vector3). Listing 49 below shows the method in question.

```
public static void GetWidthAndLength(Transform trans)
{
    Mesh mesh = trans.GetComponent<MeshFilter>().mesh;
    Vector3[] vertices = mesh.vertices;
    Vector3 minX = vertices[0];
    Vector3 maxX = vertices[0];
    Vector3 minZ = vertices[0];
    foreach (Vector3 vect in vertices)
    {
        minX = (vect.x < minX.x ? vect : minX);
        maxX = (vect.x > maxX.x ? vect : maxX);
        minZ = (vect.z < minZ.z ? vect : minZ);
    }
    float width = Vector3.Distance(minX, minZ);
    float length = Vector3.Distance(maxX, minZ);
    Vector3 widthVect = minZ - minX;
    Vector3 lengthVect = maxX - minZ;
    roadWidth = width;
    roadLength = length;
    widthVec = widthVect;
    lengthVec = lengthVect;
}
```

Listing 49: Code for GetWidthAndLength() in TMath.cs

Previously this method simply returned these values. Seeing as the is only called in one line within TScene.cs it did its job. When implementing scaling however, the authors wanted to have these four variables (see lines 18-21 in the listing above) available to be fetched at all times so it was changed from a return to setting the variables like in the listing above making them available to any method that wanted these values.

This admittedly was not a well planned solution. The problem now is that if for example a wall is used as input, then its height and width will be stored in the variables roadWidth and roadHeight, which is obviously quite misleading and results in data loss. One could instead have the road segment width and length be retrieved some other place or have a specific method for the retrieval of these values. For now, the current solution does the job, but should ideally be changed for a better solution.

6 Conclusion

The main contributions of this thesis to the project have been granting the game access to NVDB and using it to create a brand new physics based implementation of traffic within the tunnel that dynamically adapts to the traffic data from the real tunnel. As for the primary goals of the thesis:

- One of the primary objectives of this thesis was to have the tunnel within the game more closely match its physical counterpart. This has been achieved by implementing a system that enables the game to access data from NVDB. Certain parts of the game already make use of this data, such as functionality for placing child objects of the tunnel. This functionality is currently limited, but serves as a proof of concept to be built upon in the future. Finally, the tunnel and everything within it underwent some rescaling. The results are not final, but the new system allows one to easily scale the tunnel in the future in order to test different scales.
- An early objective was to enable the player to drive vehicles. Using Raycasts for starting the interaction process of letting the payer use the vehicle, and Wheel Colliders to handle the driving. A small selection of vehicles have been create are available for the player to drive them around in the tunnel. The same vehicles are also able to drive by themselves when spawned by the Unity. These have been achieved albeit with some issues. They are rough around the edges and likely need some refining to make the driving experience better.
- The traffic simulation's main objective was to create a realistic traffic picture in the tunnel. This goal has been achieved through the use of the NVDB API to retrieve and utilizing selected attributes of the selected tunnel. The traffic simulation demonstrates the potential of realistic traffic, whilst being a sustainable solution and easy to build upon, even if does fall short in certain areas.

All of the objectives were achieved with the goal of being future oriented in mind. The groundwork has been made for all of them to be easily changed or updated, rather than having to be replaced with new implementations in the future.

6.1 Future Works

The game is far from finished. There are certainly numerous improvements and features which can be built on top of the current version of the game. This section however, will focus suggestion for some bigger tasks and projects that could be worked on in the future to further improve the game. The topics will be presented in a somewhat prioritised order starting from the most important.

6.1.1 Rewriting and Optimisation of Code

As hinted at earlier there's a fair amount of problems in the code itself. It is strongly encouraged to clean up the code whenever possible and it is likely that some small optimisations of the code are needed here and there.

Splitting up code when appropriate and documenting the code well will make the process of finding the relevant code to change, whatever change that may be, easier to find. If more files are split

up there's a smaller chance that more than one person has written code inside the same file making pushing and pulling to GitHub less painful by decreasing the frequency of merge conflicts.

Large pieces of the code are not well documented and commented, making it difficult to understand the reason why a certain method was implemented in the first place.

6.1.2 Merge with Digital Twin

This thesis uses the coolengine thesis from 2022 as its foundation. As has been noted, that thesis used the self-rescue game from 2021 as its foundation, rather than the digital twin thesis written in the same year. Thus some features of the digital twin thesis are not found within coolengine but rather live on a separate branch on GitHub. The team did not have much time to look into it, as the project cloned from GitHub contains compilation errors. Based on the report [18] however, the project has some unique features such as a different traffic implementation. Hence it could be beneficial to fix the errors and port over any useful features.

6.1.3 Implement a Fire Scenario

The game does not currently have a good implementation of an actual fire in the tunnel. There are two implementations from the previous theses which are currently disabled by default. Reimplementing and improving one of these will have to be done at some point, unless a brand new implementation is created.

- The bachelor thesis from 2021 (2.1.2) wrote some code to instantiate a fire within the tunnel, as well as code for killing the player when being exposed to smoke. This implementation seems to make use of a fire asset imported from the Unity Asset Store. Some of the code however does not function correctly as demonstrated in figure 60.
- The thesis from 2022 (2.1.4) made a different implementation using an FDS simulation. The main problem with this implementation is that it is too resource intensive, resulting in long loading times and an unresponsive game. If it were to be reimplemented, it would therefore need some optimisations. As of now, it functions better as a fire simulation than an asset to be used in a game.

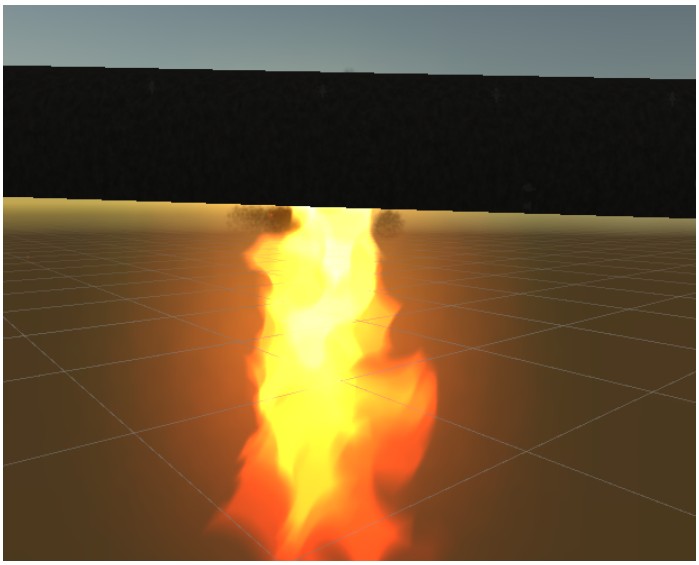


Figure 60: A tunnel tube as seen from outside whilst using the Unity rendered fire. Functions well until it attempts to spread the fire to other vehicles, which causes the flame to drastically intensify in a downward direction

Regardless of which implementation is used, the game lacks any rules for when and where this fire should appear. Hence some scenarios will need to be implemented later.

These could include having another car catch fire whilst driving or instantiating a burning vehicle reminiscent of the ones from 2021 somewhere in the tunnel. The important part is to keep in mind all of the ripple effects this will have for the rest of the game and how other objects within the tunnel should respond to the fire.

6.1.4 Implement NPCs

In real life there is usually more than one person inside the tunnel. Therefore the idea of a Non-Playable Character (NPC) naturally came to be, as this will not only make the game more realistic but also give the player more interactions to consider when playing the game.

By default they would all likely be driving a car through the tunnel. Once they notice a fire however, they may respond in different ways. Some may try to turn their vehicle around in order to escape whilst others may leave their vehicle and try to escape on foot.

Making this work would require expanding the automatically driving cars with a driver inside of them. Said driver could in early prototypes be implemented relatively easily with another player object that is instantiated next to the car when needed. Once outside, there are many possibilities for how an NPC may evaluate and respond to its surroundings.

6.1.5 User Defined Scenarios

This thesis has explored how to use the data within NVDB to have the game mimic a real tunnel. Although it does provide the most realistic experience, it does have a few disadvantages. Namely

that the game is dependent on having access to such data, as well as being restricted to scenarios as specified by that data. Future implementations should make the data from NVDB optional, whilst providing another option of allowing the user to define their own scenarios. Such scenarios could let the user set values such as speed limit, traffic rate, what objects are present in the tunnel, and so on.

It should be noted that some of the groundwork for this has already been laid by a previous thesis (2.1.4). The options so far however are quite limited as they are mostly focused on the FDS.

A suggestion for how this could be implemented in code would be to rename the current TScene into TNvdbScene. One could then make a separate TScene which will be loaded instead when the game is not expected to make use of NVDB. Such an approach provides the benefit of keeping the two solutions in two separate domains, although it would also likely lead to much reused code. Another solution would be to introduce more configuration parameters, which would either be set by NVDB or the user depending on what is desired.

6.1.6 Improve Modelling Application

Although the modelling process is capable of handling simple single-tube tunnels quite well, exploring the more complex tunnels will inevitably have one bear witness to some curious results. Some examples of problems found will be shown below. In addition to these problems, please keep in mind the problems already discussed in section 5.2.1.

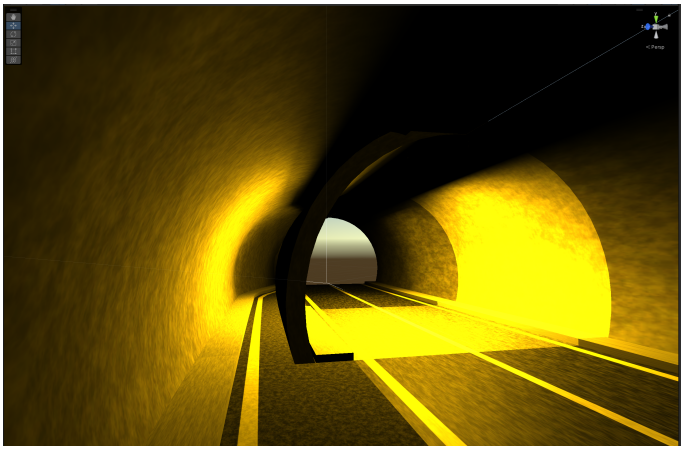


Figure 61: Entrance of Kleivene nordgående

Some tunnels seem to contain other tunnel segments. The figure shows an additional segment being placed inside the beginning of the tunnel, although more extreme cases have a separate tube intersecting with the tunnel.

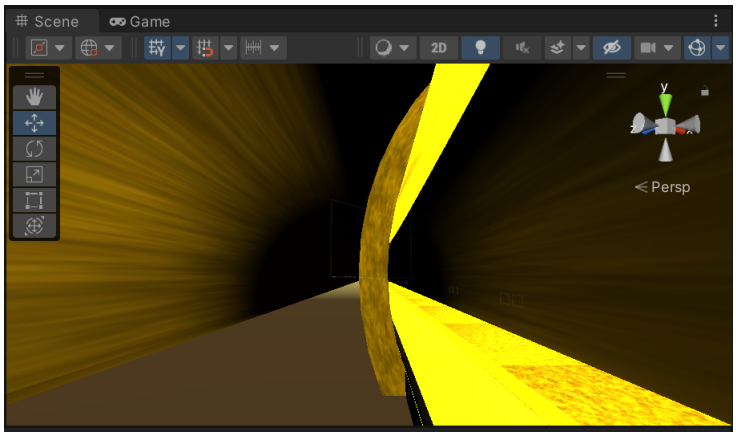


Figure 62: Helltunnelen. The camera is currently inside a wall of the main tunnel tube. The black walls on the side seem to form a second tunnel tube intersecting with the original tube

Results like the above seem to occur when a single tube of a twin-tube tunnel is modeled. It is theorised that the process for some reason tries to model both tubes even though it has been set to only model the one tube.

As discussed by the original thesis [21], the emergency parkings are not generated very well. They will often lead to problems like the ones seen below.

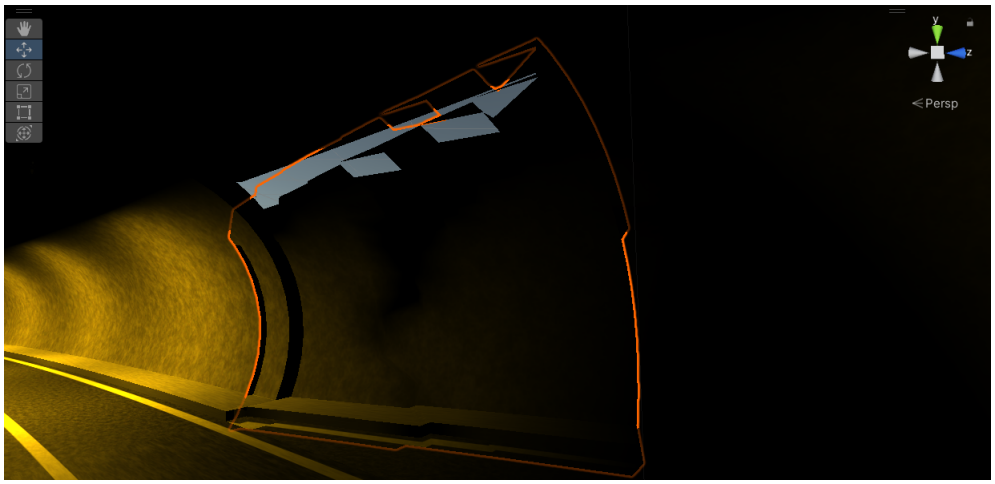


Figure 63: Storhaug emergency parking. The wall does not properly connect with the roof and the road appears jagged

Finally, the process does not seem to be able to handle roundabouts properly. Tunnel objects containing a roundabout seem to crash the modelling. The individual tubes can be modeled separately leading to results like the one below.

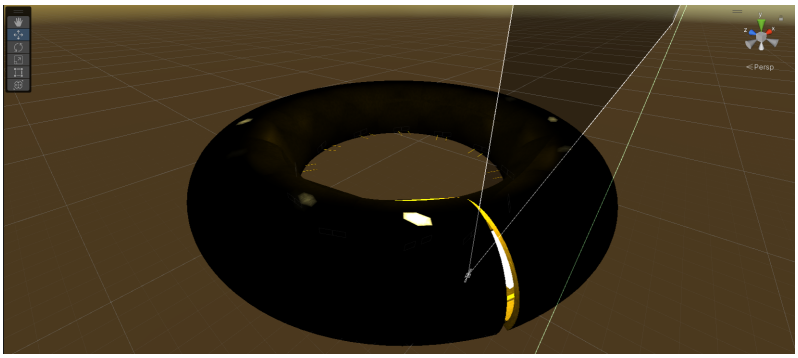


Figure 64: Outside the roundabout of Marienborgtunnelen

The above roundabout is nothing more than a normal tunnel tube that happens to follow a circular path. In order to accurately represent the tunnel, the model will need to create openings for other tubes to connect to.

The aforementioned problems are general problems with the modelling itself that may affect other applications besides the game. From the perspective of the game however, the emergency parkings are perhaps the most important to fix as they should serve as safe zones within the game.

Finally, segments of the tunnel are all named polysurface1, polysurface2 and so on as shown in figure 65 below.

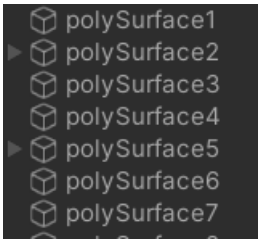


Figure 65: Segments of the tunnel model within the hierarchy. All of them called polysurface

Changing these names is not necessary, but would be helpful for working on the code as the names provide no distinction between objects such as walls and roads.

6.1.7 Support for Complete Tunnels

As has been noted, the team has been developing the game within tunnel tube objects generated by Maya rather than the tunnel objects. To clarify the difference, a tunnel object may consist of multiple parallel tubes, whilst modelling a tunnel tube will only model the one tube in question. From the perspective of the game, modelling a tunnel object will usually only result in overhead seeing as the game takes place within one tube and ends once the player exits said tube. There will in other words never be a need for both tubes to be present in the game.

There are however some exceptions; namely tunnels with multiple interconnecting tunnel tubes. Tunnels with a roundabout for instance are considered as separate tubes converging in another circular tube. This is demonstrated in figure 66.

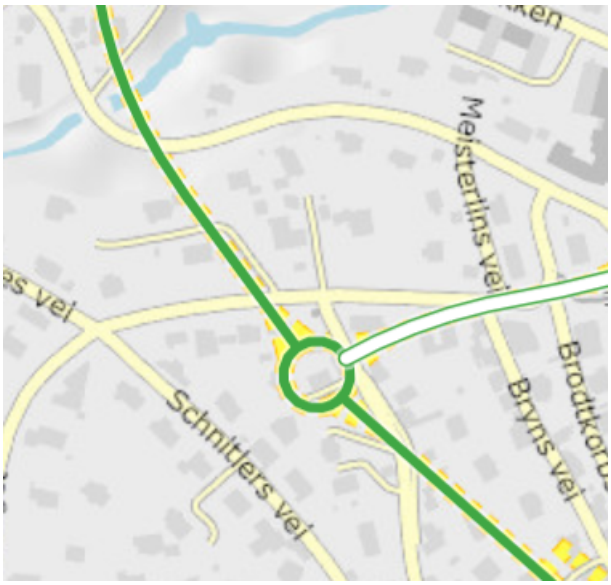


Figure 66: Marienborgtunnelen consisting of 4 tunnel tubes. 3 of them in green and the fourth tube highlighted in white

Modelling only one of these tubes would have the game incorrectly place an exit in both ends of the tunnel tube, despite one end connecting to another tube rather than being an exit.

One could solve this by modelling the tubes separately and have Unity stitch them together. The proposed solution however, would be to have Maya create a model of the entire tunnel ready for use by Unity. Maya is created specifically for the purpose of 3D-modelling after all. Doing this within Unity would also add to the already heavy workload needed when loading a tunnel.

The major obstacle in the way of this goal would be the fact that the modelling process seems to crash quite often when modelling a tunnel object (rather than tunnel tube). This is true for more complicated tunnels like the one in figure 66 as well as for many simpler tunnels. The first step would therefore be to update the modelling application to better support tunnel objects.

6.1.8 Implement Wind Direction

Several tunnels have wind turbines in order to have air flow through the tunnel, thus pushing vehicle exhaust out of the tunnel and fresh air inside. This would of course affect the air inside the tunnel as well as a potential fire that could occur inside it.

NVDB is not known to contain any data about the wind direction inside the tunnel, but it would nevertheless be of interest to implement an arbitrary wind direction in the game. Smoke would follow the direction of the wind, meaning the player may have to take the wind direction into account when evacuating the tunnel.

6.1.9 New Rescue Game Mode

Figure 67: What a game mode selection menu could look like

The new game mode introduces the player as a fireman with the task of rescuing people in the tunnel. The player objective might vary, such as to save a number of civilians in various scenarios and situations. The gameplay might involve navigating different realistic hazards and obstacles in the tunnel fire.

Various challenges can be incorporated, including obstacles as having to traverse highly dense smoke impairing the eye sight of the player, and a time limit indicating by the fire spreading through the tunnel.

By creating this new game mode, it would give the game a new value, as the finished game would be a great way for the fire department to practise with, particularly in Norway were the tunnels are located. Leveraging data gathered from NPRA, ensures for a more realistic simulation environment for practicing with for anyone. This new game mode, and the coexisting game, will serve as an immersive platform for firefighters to train their skills and preparation for emergency situations.

6.1.10 VR

Since the ultimate goal of this project is a game for self-rescue training, the game would benefit from being as realistic as possible.

Implementing VR would enhance the immersion of the player and thereby help simulate how one should act in a real scenario.

Since this project improve the realism of the game. and have a big impact on how the user interacts with the objects in the game.

However due to the number of issues and additional features that are more important for the finished project, Virtual Reality (VR) is a lower priority feature to implement that would likely better serve as one of the final implementations.

Bibliography

[1] URL: https://github.com/github/gitignore/blob/main/Unity.gitignore.
(last accessed 27.04.2023).

[2] Autodesk.
Maya Software: Get Prices and Buy Maya 2023.
URL: https://www.autodesk.eu/products/maya/overview?term=1-YEAR&tab=subscription.
(last accessed: 14.02.2023).

[3] Creative Bloq.
The best 3D modelling software in March 2023.
URL: https://www.creativebloq.com/features/best-3d-modelling-software.
(last accessed: 09.03.2023).

[4] Matthew Clark.
Local Space vs World Space in Unity.
URL: https://medium.com/nerd-for-tech/local-space-vs-world-space-in-unity-6a9944470478.
(last accessed 24.04.2023).

[5] John French.
Raycasts in Unity, made easy.
URL: https://gamedevbeginner.com/raycasts-in-unity-made-easy/.
(last accessed: 06.04.2023).

[6] GitHub.
Git Large File Storage.
URL: https://git-lfs.com/.
(last accessed 27.04.2023).

[7] Google.
Google maps streetview, Åsnutt.
URL: https://www.google.com/maps/@58.8961946,5.6714425,3a,75y,257.01h,74.13t/data=!3m6!1e1!3m4!1stL-s8wP80RHMEiWgMfY1HA!2e0!7i16384!8i8192.
(last accessed: 11.05.2023).

[8] Google.
Google maps streetview, Dørefjell.
URL: https://www.google.com/maps/@61.0387387,5.8148502,3a,75y,148.97h,90.75t/data=!3m6!1e1!3m4!1sX1xxeZDOnW87hjPTN6oqag!2e0!7i13312!8i6656.

(last accessed: 27.04.2023).

[9] Emil Haavardtun, Audun Stjernelund Lien, and Dag Hermann Valvik.
"3D Self-Rescue Game for Tunnel Fire".
Bachelors thesis. University of Stavanger, 2021.
URL: https://hdl.handle.net/11250/2774390.

[10] IBM.
What is a digital twin?
URL: https://www.ibm.com/topics/what-is-a-digital-twin.
(last accessed: 29.04.2023).

[11] Jan Krisitan Jensen and Francesco Frassinelli.
Jobb interaktivt mot NVDB api V3.
URL: https://github.com/LtGlahn/nvdbapi-V3.
(last accessed: 30.03.2023).

[12] Unity Learn.
Prefabs.
URL: https://learn.unity.com/tutorial/prefabs-e#.
(last accessed: 14.02.2023).

[13] Microsoft.
Access Modifiers (C# Programming Guide).
URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers.
(last accessed 28.04.2023).

[14] Microsoft.
override (C# Reference).
URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override.
(last accessed 28.04.2023).

[15] Microsoft.
static (C# Reference).
URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static.
(last accessed 07.04.2023).

[16] Microsoft.
virtual (C# Reference).
URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual.
(last accessed 28.04.2023).

[17] Benjamin Mydland and Ove Oftedal.
"CoolEngine: Simulating Realistic Fire and Smoke in a Unity3D-based Tunnel Fire Rescue Game".
Bachelors thesis. University of Stavanger, 2022.
URL: https://hdl.handle.net/11250/3005462.

[18] Berke Kağan Nohut.

"Digital Tunnel Twin Using Procedurally Made 3D Models".
MA thesis. University of Stavanger, 2021.
URL: https://hdl.handle.net/11250/2786176.

[19] Dented Pixel.
LeenTwean.
URL: https://assetstore.unity.com/packages/tools/animation/leantween-3595.
(last accessed 07.04.2023).

[20] snl.
Årsdøgntrafikk.
URL: https://snl.no/%C3%A5rsd%C3%B8gntrafikk.
(last accessed 28.04.2023).

[21] Georg Stava, Henrik Simonsen Knutsen, and Stian Henriksen.
"Procedural 3D Modeling of Road Tunnels: a Norwegian Use-case".
Bachelors thesis. University of Stavanger, 2020.
URL: [UNPUBLISHED].

[22] Unity.
Character Controller component reference.
URL: https://docs.unity3d.com/Manual/class-CharacterController.html.
(last accessed 20.04.2023).

[23] Unity.
CI/CD Cloud Build Automation & Deployment Tools.
URL: https://unity.com/solutions/ci-cd.
(last accessed: 09.03.2023).

[24] Unity.
Important Classes - Debug.
URL: https://docs.unity3d.com/Manual/class-Debug.html.
(last accessed: 09.03.2023).

[25] Unity.
JSON Serialization.
URL: https://docs.unity3d.com/2020.1/Documentation/Manual/JSONSerialization.html.
(last accessed: 15.03.2023).

[26] Unity.
LayerMask.
URL: https://docs.unity3d.com/ScriptReference/LayerMask.html.
(last accessed 29.04.2024).

[27] Unity.
MonoBehaviour.
URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.
(last accessed 27.04.2023).

[28] Unity.
Order of execution for event funcitons.

URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.
(last accessed 27.04.2023).

[29] Unity.
Order of execution for event functions.
URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.
(last accessed 27.04.2023).

[30] Unity.
Rigidbody.centerOfMass.
URL: https://docs.unity3d.com/ScriptReference/Rigidbody-centerOfMass.html.
(last accessed 11.05.2023).

[31] Unity.
Rigidbody.collisionDetectionMode.
URL: https://docs.unity3d.com/ScriptReference/Rigidbody-collisionDetectionMode.html.
(last accessed: 10.04.2023).

[32] Unity.
RigidbodyComponentReference.
URL: https://docs.unity3d.com/Manual/class-Rigidbody.html.
(last accessed 28.04.2023).

[33] Unity.
Tags.
URL: https://docs.unity3d.com/Manual/Tags.html.
(last accessed 29.04.2023).

[34] Unity.
Version control systems.
URL: https://unity.com/solutions/what-is-version-control.
(last accessed: 09.03.2023).

[35] Broken Vector.
Low Poly Cars.
URL: https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-cars-101798.
(last accessed 07.04.2023).

[36] Staten Vegvesen.
Nasjonal vegdatabank Datakatalog.
URL: https://datakatalogen.atlas.vegvesen.no/#/.
(last accessed: 20.04.2023).

[37] Statens Vegvesen.
Angivelse av filter.
URL: https://api.vegdata.no/parameter/egenskapsfilter.html.
(last accessed: 07.04.2023).

[38] Statens Vegvesen.
N300 Trafikkskilt.

URL: https://www.vegvesen.no/fag/publikasjoner/handboker/vegnormalene/n300/.
(last accessed: 29.03.2023).

[39] Statens Vegvesen.
Nasjonal vegdatabank Datakatalog.
URL: https://datakatalogen.atlas.vegvesen.no/#/798/Trafikkmengde,%20kj%C3%B8refelt.
(last accessed: 29.04.2023).

[40] Statens Vegvesen.
Tillat kjøreretning.
URL: https://datakatalogen.atlas.vegvesen.no/#/977/Tillatt%20kj%C3%B8reretning.
(last accessed: 07.05.2023).

[41] Statens Vegvesen.
Veglenke.
URL: https://api.vegdata.no/verdi/veglenke.html.
(last accessed: 28.03.2023).

[42] Statens vegvesen.
Hva er Nasjonal vegdatabank (NVDB).
URL: https://www.vegvesen.no/fag/teknologi/nasjonal-vegdatabank/hva-er-nasjonal-vegdatabank/.
(last accessed: 09.03.2023).

Appendix A

Source Code

A GitHub Branch

The project was developed using the Unity game engine as well as other applications such as the NVDB API, to demonstrates the potential applications of game development in fields beyond traditional gaming, such as education and training.

The GitHub branch link for this project:
https://github.com/TunnelSafety/3D-tunnel/tree/v23-self-rescue-game
(Last updated 15.05.2023)

Contributors

- Simen Kase
- Magnus Tysdal
- Jesper Søberg

Maintainers

- Naeem Khademi
- Erlend Tøssebro

B Cloning the Branch

To clone the branch v23-self-rescue-game from GitHub, use the following command:

```
git clone -b v23-self-rescue-game --single-branch git@github.com:
TunnelSafety/3D-tunnel.git [folder_name]
```

After cloning the project, a local copy of the branch should be available within a new folder called [folder_name].

Note: When cloning the repository, it is essential to run the command "*git lfs pull*" after cloning or pulling the repository. This is because, some operating systems and Git settings may prevent the Unity project from opening. This is because the LFS references for large files may not be automatically downloaded, causing the project to be incomplete. By running "*git lfs pull*", you ensure that the LFS files are properly downloaded and integrated into the project, allowing it to be opened without any issues.

Appendix B

Running the Unity Application

A Requirements

Supported Operating Systems

- Windows 10
- Windows 11

Application Requirments

- Unity 2021.3.16f1

B Running the Software

To start the game, simply run the executable included in the build folder:

```
3D-self-rescue-game-build
└─ 3D Self-rescue game.exe
```

C Unity Build

To build the project follow these steps:

1. Open the project in Unity Hub.
2. Navigate to "file" and select "Build Settings".
3. In the Build settings window, select the "Build" button and choose the location of where to place the build.

After completing the build process, it is necessary to manually move some additional assets. This involves moving the "Assets/Resources" directory to the "Assets" folder in the build location. (The same must be done for the "Assets/CFD" folder, if desired in the game.)

If there are new assets added in the future that are necessary for the game to function properly, they must also be moved manually to the "Assets" directory in the build location.

Appendix C

Modelling Tunnels

A Requirements

Firstly, Autodesk Maya will need to be downloaded from the follwoing link:

https://www.autodesk.com/products/maya/free-trial

Furthermore, the application has a number of python dependencies. Most of them should be installed by running the following command inside the proceduralmodeling directory:

```
pip install -r requirements.txt
```

The dependencies are listed below in case the command should fail.

- Requests
- Colorama
- Maya
- Pymel
- SciPy

Finally, the nvdbapi-v3 library [11] is also required and can be installed using the following command:

```
pip install nvdbapi-v3
```

B Running the Modelling Application

The easiest way of modelling a tunnel is to navigate inside the proceduralmodeling directory and running gui.py. Doing so will open the GUI shown in figure C.1.

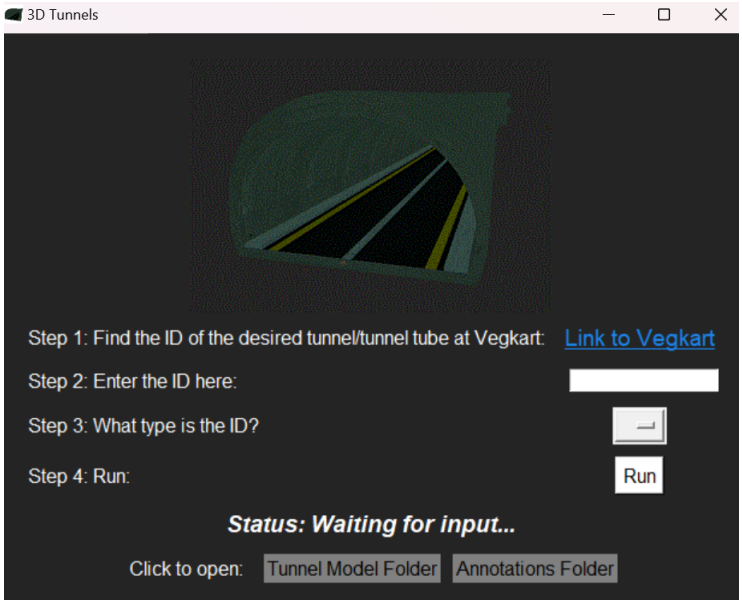


Figure C.1: The GUI used for modelling tunnels

Follow the steps provided by the program, although keep in mind that the game is currently built with tunnel tube objects in mind (not tunnels). When "Run" is pressed, the application will begin logging various messages in the terminal. The process is finished when the message "Successfully closed Maya subprocess!" appears.

At that point, the tunnel should be ready for the game. The files will automatically be placed where they are needed and the data retrieval process will automatically be run as well.

Appendix D

Player Controls

On Foot:

- Player Movement: WASD
- Jump: Space Bar
- Crouch: C
- Interact: F
  - When holding object (e.g. a fire extinguisher): Drop: G
- Spawn vehicle: T

In Vehicle:

- Car Movement: WASD
- Break: Space Bar
- Exit car: G