University of Stavanger

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

3D Self-Rescue Game for Tunnel Fire

Bachelors's Thesis in Computer Science

by

Emil Haavardtun

Audun Stjernelund Lien

Dag Hermann Valvik

Internal Supervisor

Erlend Tøssebro

External Supervisor

Naeem Khademi

May 15, 2021

Abstract

This thesis aims to create a 3D game in Unity to practice self-rescue training during emergencies involving fire, using tunnels that are modeled procedurally. Closing down tunnels to practice tunnel safety is cost-inefficient and an impediment to the tunnels' operation and infrastructure. This motivation and previous work on tunnel safety inspired and laid the basis for this thesis.

The research concluded that it is possible to create a self-rescue game in Unity, using tunnel models created procedurally. The game created for Unity can populate a tunnel model with objects and create a playable self-rescue scenario and opens for the possibility to use any single tube tunnel model.

The hope is to inspire researchers to continue this research to create more accurate models and more realistic self-rescue scenarios for the public.

Acknowledgements

Contents

Chapter 1

Introduction

This chapter will:

- Explain the motivation for the thesis.

- Introduce the objectives of the thesis.

1.1 Motivation

Fire and other accident constitute a significant security concern when designing tunnels in Norway. When accidents occur, the principle of self-rescue applies as a person cannot assume that rescue personnel will arrive in time. Many resources are being put towards research on systems that can help people survive. Systems like lighting, special alarms, and color choice are all carefully researched, tested, and finally selected to help anyone intuitively rescue themselves in case of emergency. More extensive, more prominent, and more extreme tunnels are being built as time passes. One example of this is the newly opened Ryfast tunnel which is the longest undersea tunnel in the world. Closing down tunnels for the purpose of practicing tunnel safety is both cost-inefficient and an impediment to the operation and infrastructure in the tunnels. Therefore, UiS has established a research group, ITC-based Tunnel Safety Research Group (ITSRG). ITSRG has the goal to make tunnel safety training available for the public by making digital versions of the tunnels and preparing the users of road tunnels for emergency scenarios. Former theses made from the research group are a mobile app for self-rescuing, and a system to procedural generate 3D models of tunnels in Norway.

1.2   Objectives

The objective of this thesis is to create a 3D game in Unity, using tunnels that are modeled procedurally. The game can be used to practice self-rescuing training during emergencies involving fire.

Secondary objectives of the thesis are:

- Place relevant objects in the tunnel.

- Implementing interactive objects, that can help the user rescue themselves.

- Creating different levels of difficulties.

- Use virtual reality to increase the realism.

Chapter 2

Technology Choices

This chapter will:

- Give an overview of the terms and concepts needed to understand the project.

- Describe the technology used to build the game in this project.

2.1 API

An Application Programming Interface (API) is an interface that lets one program access features from another program. One example of this is how a program can get information from NVDB by sending a GET-request to the website. Another example can be an application that responds to user input which can control something in the application, for instance moving an object.

2.2 C#

Unity uses C#(C-Sharp) as its scripting API. Scripting in Unity tells GameObjects how to behave and interact with other GameObjects. C# is a type-safe object-oriented language developed by Microsoft [1]. The primary parts of the language are variables, methods, and classes. The variables hold information, such as a number, a string, or a reference. To easily distinguish the variables they should be written with a lowercase first letter. Methods contain code to manipulate the variables, and contrary to the variables should be capitalized. Code is written in methods to efficiently be reused in different parts of a program. Most methods and variables are declared in classes, where depending on

the security level, other methods outside the class can access them. Classes like methods should be capitalized [2].

2.3   Visual Studio

To write C# code, an environment to develop in is needed. Microsoft Visual Studio is an integrated development environment (IDE) [3]. Visual Studio is compatible with Unity and is available for both macOS and Windows. Visual Studio is integrated well with Unity which is the reason for choosing Visual Studio as the IDE. Visual Studio has good support for C#, making it easy to write and run code [3].

2.4   NVDB

Nasjonal vegdatabank, NVDB for short, is a national database with information about roads in Norway, created by the Norwegian Public Roads Administration. The database contains, among other things, the geometry and topology of the roads, an overview of equipment along the road, an overview of traffic and accidents [4]. The NVDB API lets the developer interact with the database to retrieve data.

By getting data sets through the NVDB API, the procedural modeling algorithm can generate 3D models of road tunnels. When the researchers of the algorithm developed it, the NVDB API was in version 2. However as they pointed out, version 3 was under development and expected to be released soon [5]. When this project started, version 3 was already released [6]. However, as it did not include any updates that would impact the procedural modeling and time was limited, it was chosen to stick with version 2.

2.5   Autodesk Maya

Autodesk Maya, often just called Maya is a powerful and professional 3D modeling and animation tool [7]. The thesis on procedurally generating 3D models of road tunnels used Maya [5]. Thus, the researchers are continuing the use of Maya in this thesis. The functionalities in Maya can be exposed through the scripting language MEL (Maya Embedded Language) [8]. Maya also has a Python library with an API (maya.cmds) that allows most of the functionalities in MEL to be used in Python [9]. Thus Python can be used to create 3D models in Maya.

2.6   Python 2

In the thesis on procedurally modeling tunnels which this thesis builds on, the researchers chose Python. Therefore, the usage of Python was continued. In their thesis, they note that Maya uses Python 2. However, they also note that Maya had plans to switch to Python 3 [5]. Starting this project, the researchers investigated and found that the Maya API had not yet released support for Python 3. During the project development, Maya released an update for their API supporting Python 3 [10]. At this point, work with Maya and Python was finished, and the researchers decided against updating the project to use Python 3.

2.7   GitHub

GitHub is a code host platform for collaboration on projects [11]. GitHub allows for version control, meaning storing modifications and old versions of a project is possible. A developer can create something new or make changes to a project, commit the changes, and the new and old work will be stored. On GitHub, repositories are used to organize projects. A repository can contain folders, documents, images, code, and all project documentation [11]. A repository can either be private for only the project developers to see or public. In a repository, multiple branches can be made to work separately on the project. Issues in GitHub are a way of organizing a project into tasks and bugs. Issues is a separate section in every repository and is an excellent tool for the team to discuss challenges and enchantments for the project.

2.8   Procedural 3D Modeling

A 3D model is a mathematical representation of a 3-dimensional object. Procedural 3D modeling is a technique that uses an algorithm to convert input sets into an output model. The input values include all parameters and constraints to make the procedural content.

An example of this is the procedural 3D modeling of road tunnels [5]. Here the input set is data describing properties of the road tunnel, among others the width, length, and position. The input set is taken from NVDB and a 3D modeling software runs the algorithm. In this case, it is Autodesk Maya and the outputted model is a 3D model of the tunnel in the *FBX* file format. This way it does not matter which tunnel is described, if the input set follows the requirements the algorithm produces a model that resembles the tunnel specified based on the input. With this technique, a 2D model of a tunnel on a map, as in Figure 2.1 can be converted to a 3D model in Unity as in Figure 2.2.

2.9 Unity

Unity is a cross-platform game engine and is developed by Unity Technologies. It is a professional platform that offers a paid subscription for its products. Fortunately, Unity offers free products to students. Unity has a repertoire and can create 2D, 3D, VR, and AR games and other experiences such as simulations and films [12]. The objective of the thesis is to use Unity as the platform to develop the game as described in section 1.2.
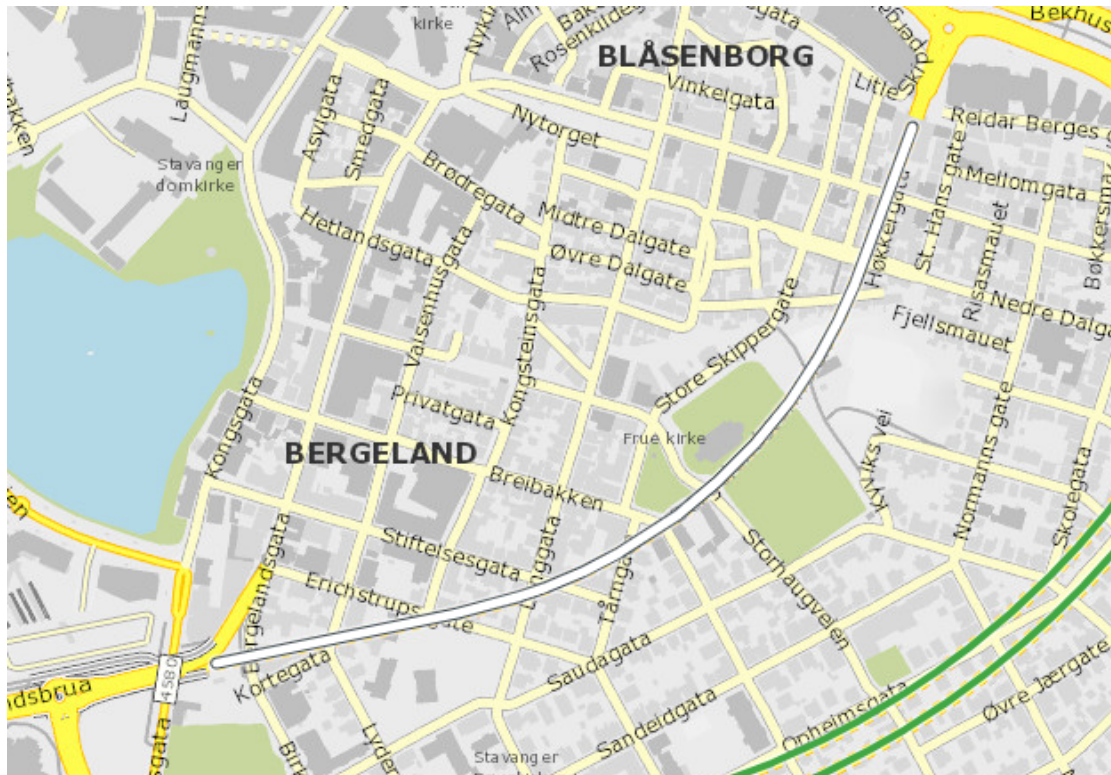
Figure 2.1: Visual of a 2D tunnel from a map
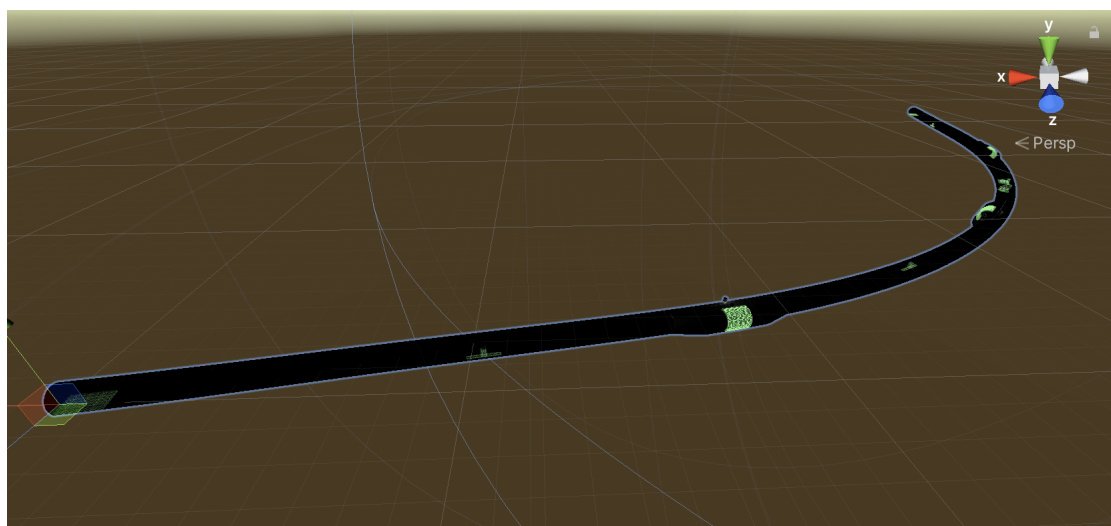
Figure 2.2: 3D model of a tunnel in Unity

This is in accordance with the theses that this thesis builds on, as it had chosen Unity [5].

2.9.1 Unity developer UI
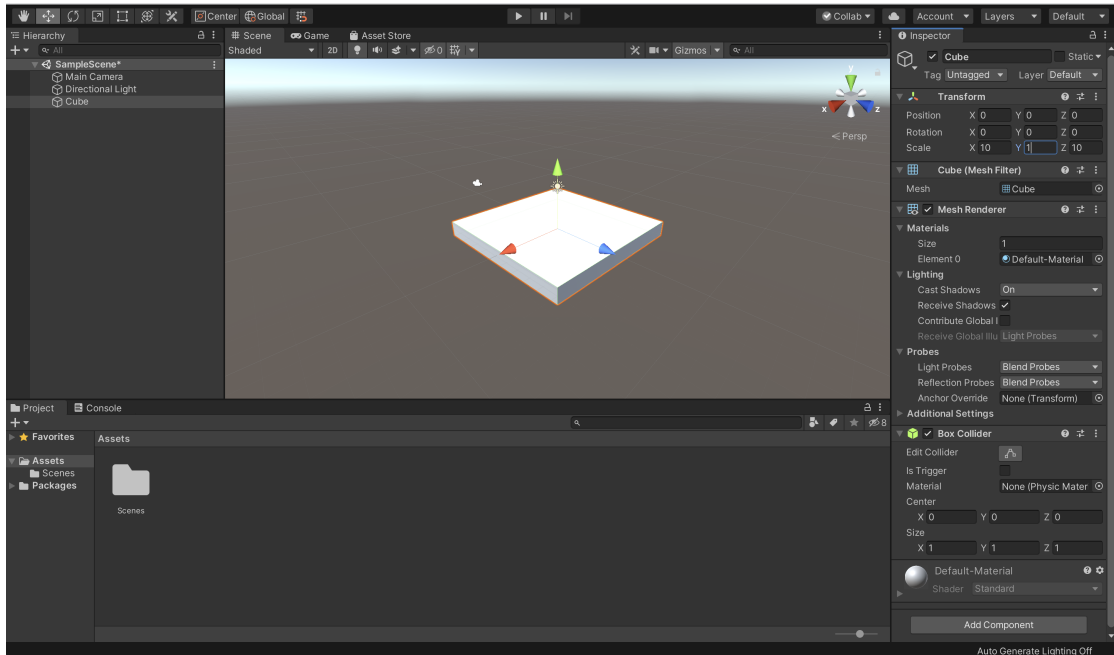
Figure 2.3: Default layout of the user interface in Unity

The default UI (User Interface) is shown in Figure 2.3 and consists of four windows. Each window has its functionality, allowing the developer to interact with the application's various features. Some of the windows are:

- Scene - Shows the current scene and allows the developer to interact with it. The developer can position the view as desired and move GameObjects around in a scene and save it later on.

- Hierarchy - Shows all GameObjects in the current scene. In Figure 2.3 the developer is in the scene "SampleScene" and all GameObjects in the scene are children of "SampleScene" like the GameObject "Cube".

- *Project* - All project files are available from the project window. Assets files like scripts, prefabs and materials the developer want to use can be found here.

- *Inspector* - Shows components and properties of a selected GameObject. In Figure 2.3 the GameObject "Cube" is selected and shows the components with its properties, for instance "Transform" and the properties "Position", "Rotation" and "Scale". These components can be altered and new ones can be added.

- *Console* - When running the game, the console window display error messages, other notifications and debugging messages from the developer from such as the print function.

- *Game* - Runs the application in the way the user would see it. The game window allows the developer to use the functionalities of the user in a simulation of the application.

2.9.2 GameObject and Components

Every object in a Unity application is a GameObject. The cube in Figure 2.3 is a GameObject. A GameObject in itself does not have any functionality, and it simply serves as a container. For example, in the inspector window, the developer can create an object by adding and modifying components to the GameObject container. These components will make the characteristics and functionality of the object. Examples of the most important components are:

- *Transform* - By default, all GameObjects in Unity have a Transform Component attached [13]. Transform is used to handle an object's position, rotation, and scale.

- *Script* - A script can be added to a GameObject to give it functionality. Scripts are used to offer user interaction, trigger events, and alter all the other object components over time.

- *Rigidbody* - The Rigidbody component will give the GameObject physical characteristics like mass and gravity.

- Mesh - The Mesh of a GameObject determines its boundaries and how to apply components like materials. The Mesh is split into two Components, a Mesh Filter containing the boundaries of the GameObject, that is where the GameObject should be rendered, and a Mesh Renderer telling Unity how the GameObject should be rendered.

- *Material* - Materials decide how an object will look like on the surface. Materials have different shaders, and the chosen shaders decide what properties can be changed. Some of the properties of the materials are color, texture, and sliders.

- *Light* - A Light object can be created by adding the Light component.

- *Particle System* - A particle system is used to produce particle effects dynamically. The system renders small images, which are called particles. All particles are simulated collectively and can be manipulated with the ParticleSystem Component.

ParticleSystem is a comprehensive tool with many parameters. The parameters can be manipulated in the Inspector, and scripts [14].

2.9.3 Built-in methods in Unity

All classes in Unity inherit from the most basic class, MonoBehaviour [15]. When making a script that derives from MonoBehaviour, the `Start` and `Update` methods are made automatically. The class contains some other important methods as well:

- `Awake` - The method is called when a script is being loaded when a GameObject containing the script is initialized [15].

- `Update` - The method is called for every frame. The method can be used when handling things that need continuous updating, like controlling a player's movement [15]. `FixedUpdate` - and `LateUpdate` can be used alternatively. `FixedUpdate` uses `Time.fixedDeltaTime` as the time between calls [16]. This means the updates are called independently of the frame rate, which can vary by many factors, making it ideal for applying force to an object, as this requires constant updates. `LateUpdate` is always called after all `Update` functions. If something needs to be handled exactly after the `Update` is called, `LateUpdate` can be used [17].

- `Start` - This method, like `Awake`, is only called once for a given script. It is called on the first frame the script is enabled just before `Update`, or any other method, is called [18]. The result is that `Awake` is always called before `Start`, which means that the order of execution can be controlled with `Start` and `Awake`.

- `OnCollision` - is used to handle collision between GameObjects. There are several methods using collision, with `OnCollisionEnter`, `OnCollisionExit` and `OnCollisionStay` as the most important ones [15]. When the Collider or Rigidbody of a GameObject is in contact with another Collider or Rigidbody, `OnCollisionEnter` is called. `OnCollisionExit` is called the frame when the contact has stopped, and `OnCollisionStay` is called every frame there is contact.

- `StartCoroutine` - starts a Coroutine, an excellent tool for managing behavior over several frames [19]. Yield is used to pause the coroutine's execution and resume to the next frame. `StartCoroutine` can be used if a method should be called after a certain amount of seconds, then the method and amount of seconds to wait should be passed in as arguments.

2.9.4 Unity UI toolkit

Unity UI is a UI toolkit for developing user interfaces for games and applications. It is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces. The Canvas is the area in which all UI elements should be inside. The Canvas is a GameObject with a Canvas Component attached to it. The UI toolkit makes creating user interfaces reasonably simple, and it even simplifies the process in many scenarios. For example, when creating a new UI element Image, the toolkit will automatically create a canvas and make the image element a child of the canvas if there is no canvas in the scene.

2.9.5 Assets

Assets in Unity are items that can be brought into the application. These items can be imported as external files from the Unity asset store or any other website. Assets can also be made in Unity and other programs. Examples of assets are 3D models, images, prefabs, scenes, and scripts. Prefabs are important assets when developing a game in Unity. The prefab system allows the developer to store complete GameObjects with its components to be a reusable component. If the developer wants a GameObject to be used in many different scene locations or different scenes, the GameObject should be converted to a prefab. The developer can then load the Prefab as a GameObject when needed and optionally change any of the components of each specific GameObject. Any changes made to the prefab will then sync with all the GameObjects using the prefab, making it easy to maintain.

2.9.6 Plugins

Plugins are the use of code created outside Unity [20]. In Unity, two types of plugins can be used. Managed plugins are exclusively made in the .NET library, while native plugins are made by third-party libraries [20].

- *TextMeshPro* is used as a replacement for the components UI Text and Text Mesh [21], and is a managed plugin. TextMeshPro provides better control over text formatting and text rendering.

- *LeanTween* is a tweening plugin used for Unity3D [22]. Tweening is the animation when objects are going between frames. LeanTween can be used to manipulate UI Components from a script.

2.9.7 Unity Collaborate

Collaborate is a built-in feature in Unity and makes working as a team easier. In collaborate a team member can save, share and sync Unity projects with other team members over the cloud. In Figure 2.4 the collaborate feature is shown.
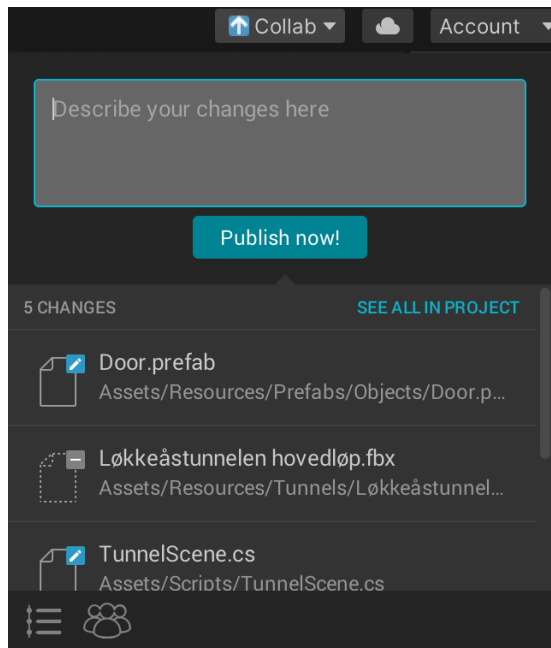
Figure 2.4: Collaborate feature of Unity. Local changes can be pushed to the cloud and the publishing history can be displayed by pressing the bottom left button.

Chapter 3

Related literature

This chapter will describe:

- Previous works done by ITSRG.

- Projects on tunnel safety done by others.

3.1   3D models for tunneling developed at UIS

The University of Stavanger has a research group on tunnel security called ITC-based Tunnel Safety Research Group (ITSRG). This section will address some of the projects related to the research group. These projects will be a basis for this thesis.

3.1.1   Procedural 3D Modeling of Road Tunnels: a Norwegian Use-case, bachelor thesis

A bachelor thesis in 2020 developed a system to generate 3D models of road tunnels in Norway procedurally. This model was developed to perform tunnel safety training in any given tunnel in Norway. The thesis explored the possibility of using a public database of Norwegian tunnels developed by the Norwegian Public Roads Administration (NVDB) to generate 3D models of the tunnels procedurally. The models generated include a single tunnel tube with the correct profile and shape. A graphical user interface was implemented in the application where the user can choose a specific tunnel from the NVDB to be generated. Autodesk Maya was used as the software to develop the 3D models. A simple demo using virtual reality was used to give the player a genuine experience of moving around in a tunnel [5].

In this thesis, an aim is that a user can choose a preferred 3D model of a tunnel generated using the algorithm described above. The tunnel can then be used in a scene in the game, making it possible to practice self-rescuing that is relevant for each tunnel user. Therefore, the thesis/algorithm can be considered the basis that this thesis builds upon.

3.1.2 Procedural and Parametric 3D Modeling of Road Tunnels, a DAT620 project report

The Procedural and Parametric 3D Modeling of Road Tunnels project aimed to take procedurally generated tunnel models into interactive use. The tunnels generated from the NVDB gets information about equipment used in the tunnels [5]. Such information was presented as placeholders. This project combined scripts and manual work to replace the placeholders with 3D models such as lights and emergency stations. A fire and traffic simulation was developed, where the user could extinguish the fire that appeared with an interactive fire extinguisher. Other objectives were to support both VR devices and mobile devices. The project highlighted some of the problems with the models that were generated, and also improved part of the algorithm to fix some of them [23].

3.2 3D modeling Ryfast tunnel, by Bouvet

In 2019, the Ryfast tunnel, the world's longest underwater tunnel, between Stavanger and Ryfylke opened. The consulting company Bouvet, was challenged to develop a digital twin of the tunnel to practice safety in the tunnel without interfering with traffic and infrastructure inside the tunnel [24]. The challenge consisted of developing a full-scale 3D model of the Ryfast tunnel, with replications of every object in the tunnel, and implement support for virtual reality.

To implement a replica of the tunnel, Bouvet used Unity. Things like pictures, a real-life 3D model of the tunnel from the designers, and manual measurements were used to place out all objects in the tunnel. The user could then interact with items in the tunnel, as well as use the fire extinguisher to extinguish a simulated car fire. To simulate a real-life tunnel, there were cars inside the tunnel driving at 90 km/h [24].

The project turned out to be a great success. Today, the VR tunnel is used by Vital Infrastruktur Arena to have courses and to educate the public in tunnel safety.

3.3 Research on safety rooms in tunnels, by SINTEF

In 2013, Statens vegvesen (The Norwegian Public Roads Administration) got harsh criticism from Statens havarikommisjon (Accident Investigation Board Norway) after the dramatic fire in Gudvangatunnelen. Gudvangatunnelen is one of fifty single-track tunnels with a length of more than 3 kilometers in Norway. These tunnels are considered to be especially exposed when accidents occur. Over the last ten years, there had been multiple other fire accidents as well, therefore a research project was started to study tunnel safety. The research aimed to find out if rescue rooms could be one solution for optimizing safety in exposed tunnels. SINTEF (Stiftelsen for industriell og teknisk forskning), a Norwegian research institute that had experience with virtual reality as a tool for research was handed the job. The researchers had to find out if people would find the safety rooms on their own when there is a tunnel fire and if the rooms are perceived safe enough [25].

For the study, 44 persons were placed inside an occupational physiology laboratory where temperature and humidity were regulated to simulate the climate in a tunnel. The gear used in the attempt was a walking simulator and VR headset. From the study, the researchers documented the following observations:

- Independent of age, sex, and nationality, road users find a way to evacuation rooms when they are surrounded by thick black smoke.

- People think it is safe to stay in the evacuation rooms over an extended time.

- Visual and acoustic guidance systems help people to find the evacuation rooms.

- Distance between evacuation rooms should not extend 250 meters.

- No one should leave the room by themselves.

- Communication with road traffic control center is important for people to feel safe and taken care of when staying in an evacuation room.

- Placement of a communication system in the rooms is important.

See [25] for further reading of the study.

Chapter 4

Construction

This chapter will describe:

- Difficulties when using procedurally modeled tunnels in Unity.

- Construction and solutions of creating a 3D game in Unity.

4.1 Tunnel structure

The most important part of the game is the tunnel models that the games are placed in. These are 3D models of tunnels created using the algorithm for procedurally generating models in Maya Autodesk. The algorithm stores the models in an FBX file. FBX, short for Filmbox, is a file format for storing models [26]. These models can then be imported into Unity, which also supports the FBX format. Each such FBX file consists of a single parent object, a container, and many child objects that together form a tunnel. An example of the hierarchy of a tunnel model imported into Unity is shown in Figure 4.2 These child objects can be split into two categories, segments, and placeholders. Together the segments make up the tunnel, and they can be further divided into road, wall, roof, ground, and shoulder. An example of a tunnel with a shoulder segment highlighted can be seen in Figure 4.1. There are also respective versions for the tunnel lay-bys described in section 4.5.

On the other hand, the placeholders are placeholders for objects that are not properly implemented in the tunnel model. Each of the placeholders has different colors, where each
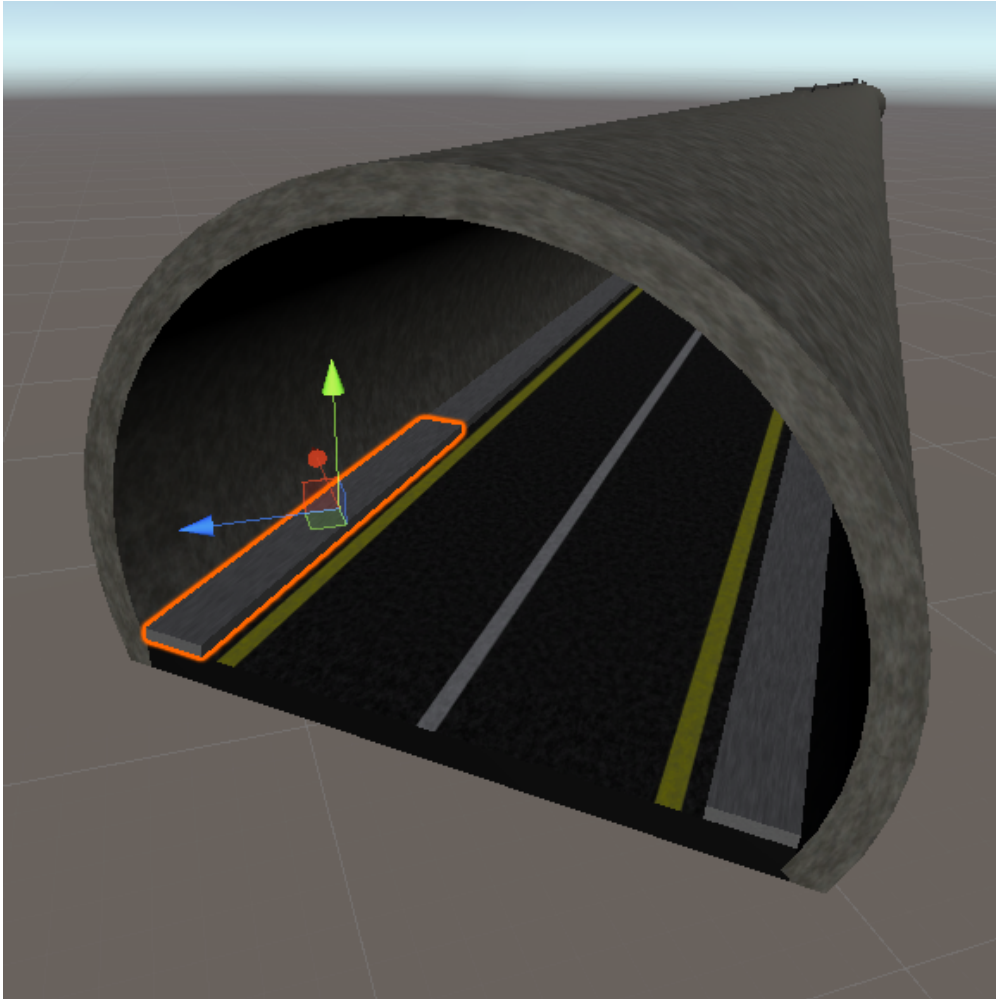
Figure 4.1: A tunnel with a shoulder segment highlighted.

color indicates different categories of objects. All the placeholders with their respective color are listed in Table 4.1.

4.1.1 Related problems

When the mentioned FBX-file is imported into Unity, only the parent object has a Transform component with a meaningful position, scale, and orientation. None of the child objects do neither segments nor placeholders. The reason this data is lost seems to be the way FBX files work. In an FBX file, a child object's transform properties are relative to the parent object. When an FBX file is imported into Unity any property containing this information has been lost. It might be a different way of exporting and or importing the files might keep this information, however, the researchers were unable to find any way of doing this. A consequence of the transformation being lost is that placing GameObjects in a model is not entirely straightforward. Furthermore, the placeholder objects in each tunnel model are not placed at their correct place. They are all stacked at either the start or the end of the tunnel. This is due to placeholders not being fully

Figure 4.2: Hierarchy of a tunnel with some segments listed.

Object Category	Box mesh color
Lighting point	Yellow
Lighting distance	Yellow
Led light distance	Yellow
Fire extinguisher	Red
Fire alarm system	Red
Hydrant	Red
Emergency station	Red
Emergency exit	Green
Sign	Green
Technical cabinet	Blue

Table 4.1: Placeholders for objects in a tunnel and their corresponding color.

implemented into the algorithm generating the models, as described in the Procedural 3D Modeling of Road Tunnels thesis [5].

4.2 Scenes

Scenes are an important part when creating a game in Unity as they give control over which objects are loaded. This game utilizes two scenes, a Menu scene and a Tunnel scene. When the program starts the user is placed in the Menu scene, which is described in subsection 4.11.1. The Menu scene has a method `PlayGame` that is called when the user presses the Start button. The method loads the Tunnel scene asynchronously in

single-mode. By loading the scene in single-mode the Menu scene is destroyed, which results in Unity having to keep track of less and thus increasing performance while in a scene. On the other hand, it might also lead to slightly longer loading times when the game changes scene as it has to recreate the scene. By loading the scenes asynchronously, increased control of the loading process is achieved, such as the possibility to implement a loading scene should the loading time become too long. The Tunnel scene is now loaded in, containing a tunnel model with, among others, vehicles, emergency doors, and lights, all of which are described elsewhere in this chapter. While in the scene there are multiple ways the user can return to the Menu. However, all of these ways call either the `ExitGame.Exit` method or the `ExitGame.InstantExit` method. These methods ensure that everything is handled correctly upon loading the Menu scene again. It loads the Menu scene in single-mode, destroying the current scene and therefore, also the tunnel model. This way, the next time the Tunnel scene is loaded it is empty.

4.3 Distinguishing segments

When a tunnel model is procedurally generated and then imported into Unity, there is no easy way to determine what kind of segment any specific tunnel segment is. As Figure 4.2 shows, as each polySurface is a segment, segments are named numerically and with no other identifier. The amount and order of segments vary from tunnel model to tunnel model, which means there is no pattern in the order of the segments. As such, simply selecting one or more segments by name and number will not work as a way to determine the type of segment(s). For example, there is no pattern such as every fifth segment being a shoulder segment. However, one way of distinguishing a specific segment is by looking at the segment's material. Since a road segment looks different than a wall segment, they are distinguishable by comparing the materials. However, when Unity imports an FBX model, it creates a new name for all of its materials if no equal material already exists. As with the segments, it once again names them numerically. For example, a road segment could have the material named "lambert2", while a shoulder segment might have the material named "lambert3". Once again, this is a problem, as it is difficult knowing which number equals which type of segment, especially as the values might be changed if changes are made to the procedural modeling. Therefore, a solution that both made the materials have more logical names and managed to identify the material and map this to the type of segment.

Firstly, a script, `TunnelImporterScript.cs`, that extends the class `AssetPostprocessor` was created in the folder where new tunnel models are imported. An asset post-processor in Unity allows for control over how assets are imported into Unity. The processor is only
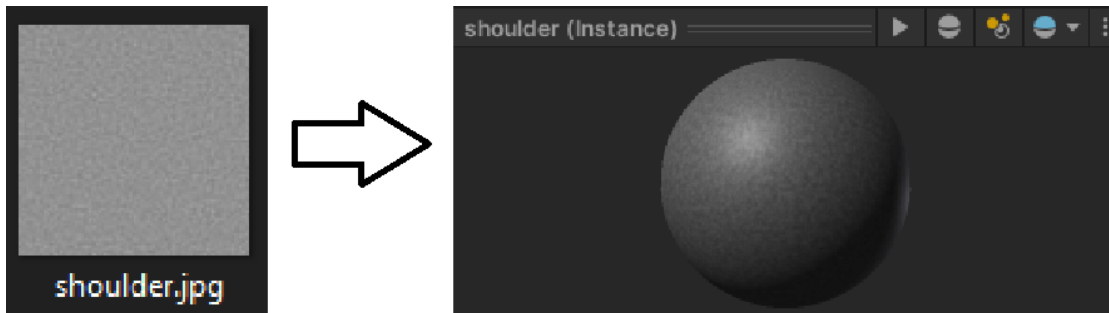
Figure 4.3: Shoulder material when used in the procedural modeling compared to the shoulder material in Unity.

affecting the assets imported into the folder it is located in, giving flexibility and control. With control over the import, it is possible to set the import settings of the model to use external material location. This way the materials are extracted separately from the tunnel model. By extracting the materials separately, they keep the same name as in the procedural modeling when they were originally applied to each segment. This means a shoulder segment now gets the name "shoulder", as that is what the material is named in the procedural modeling. An example of the "shoulder" material being converted is shown in Figure 4.3. This means each type of material, and thus segment, is now easily distinguishable from others. Getting the material name is simply done by reading the name from the Mesh Renderer component. When a tunnel loads into a scene, it loops through segments getting their material name and then saves a reference in a list for each segment type. Later on, an enumerator was used for easy and quick access when they are needed. For example, if a method needs to loop through all the wall segments, it can simply get a reference to the list.

This solution works great as it is both fast and scalable, only looping through a tunnel model once. As described later, this was especially useful when there was a need to use new materials in tunnel models.

4.4 Position and placement

When placing an object in the tunnel both the position and the orientation of an object is essential. One example is instantiating traffic in the tunnel where vehicles must be placed spread out, and orientated in the direction of the tunnel. When placing an object in the scene, it is always done relative to a tunnel segment. As mentioned earlier the Transform Component of the segments does not have a global position nor orientation. However, the segments have a Mesh, and the Mesh (both the has a global position to render it at the correct place. Therefore, the object's position could be found from the Mesh, allowing the placement of the object relative to the segment. Figure 4.4 is highlighting the Mesh

of a road segment. Also visible is the Transform Component and its properties. The orientation of the segment is not as simple and therefore two methods of calculating the rotation were developed. These methods are both handled in the `TMath.cs` script and described in the following sections.
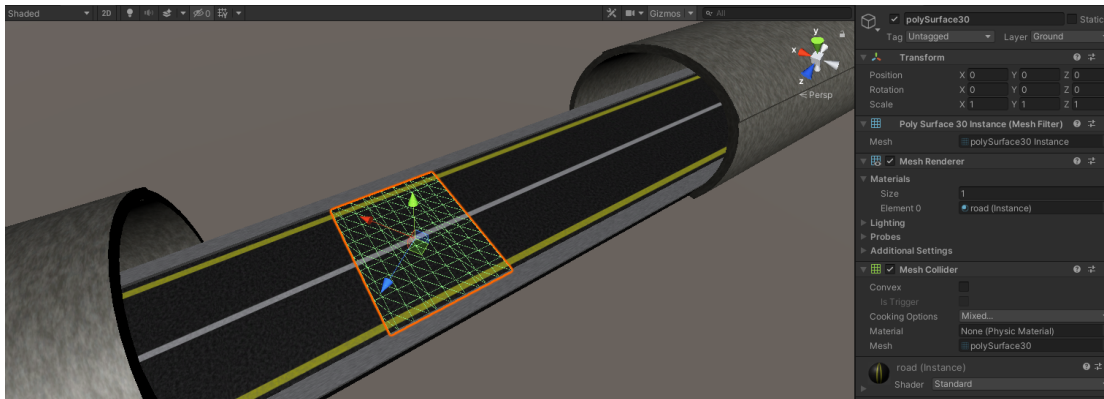

Figure 4.4: A road segment, with the Mesh Filter highlighted.

4.4.1   DirectionVectorBetweenRoads

The first method `DirectionVectorBetweenRoads` gets the center global position of the two specified objects and returns a vector between the position of these two points. An object can then align along the direction of the vector. However, the method does have an inaccuracy. An example of this is when the method calculates the rotation of the road. When the road is turning, the vector between the two segments will be oriented slightly more in the direction of the turn than the specific road segment. An example of the difference is seen in Figure 4.5, where the red line is equal to the vector between the position of the two center points and the blue line shows a closer representation of the orientation of the object. As can be seen the two lines converges, meaning that they are not equal, with the red line pointing more in the direction of the turn than the blue line. This inaccuracy is perhaps most notable when placing vehicles in the tunnel as even just a few degrees off makes it look like the vehicles are driving into the tunnel wall.

4.4.2   DirectionVector

The other method is `DirectionVector` which aims to find a vector that has the same direction as the blue line in Figure 4.5, thus avoiding the problem described in the previous section. The method does this through a few steps. First, it calls on the `GetWidthAndLength` method that returns the width and length of an object, both as scalars and as vectors. This is done by looping through all the vertices in the Mesh Filter Component, which equals all the corners of the visual object, the Mesh. These vertices have global positions. While looping through, the method finds the vertex with the lowest x-coordinate, the highest x-coordinate, and the lowest z-coordinate. A representation can be seen in Figure 4.6, where the yellow sphere is equal to the minimum z-coordinate
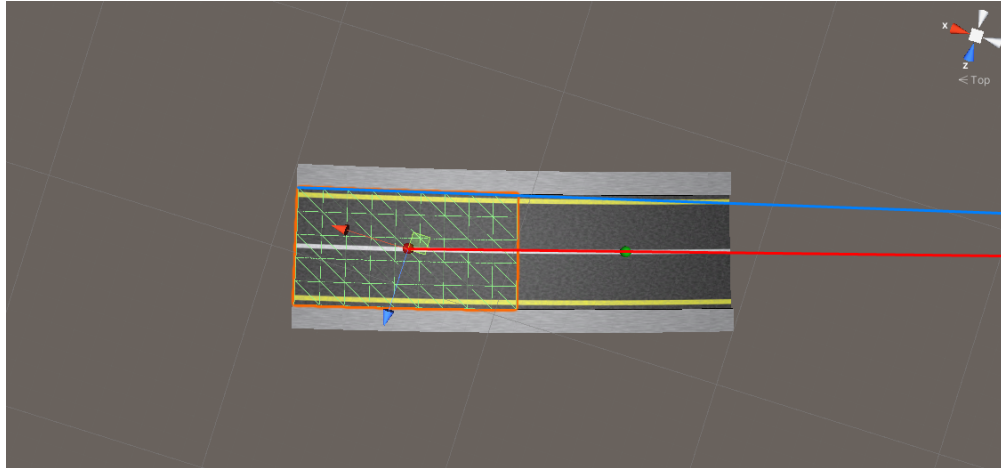
Figure 4.5: Direction vector between two segments in red compared to the correct direction vector in blue.

vertex. The method then creates a vector from the minimum z vertex to each of the other two points. These vectors are approximately equal to the length and width of the segment, as shown in green and orange in Figure 4.6. As there is no guarantee as to which of these two vectors are equal to the length of the segment, the blue line in Figure 4.5, the method calls upon the `DirectionVectorBetweenRoads` to get an approximate direction vector. It then normalizes all the three vectors and compares the absolute value of the dot product between the direction vector and each of the two other vectors. The dot product gives a measurement of the correlation between the direction of the vectors, and thus the vector with the highest measured correlation is the length vector. This method is probably best used for objects that resembles a parallelepiped, like road and shoulder segments, because of how it uses the edges of the object.
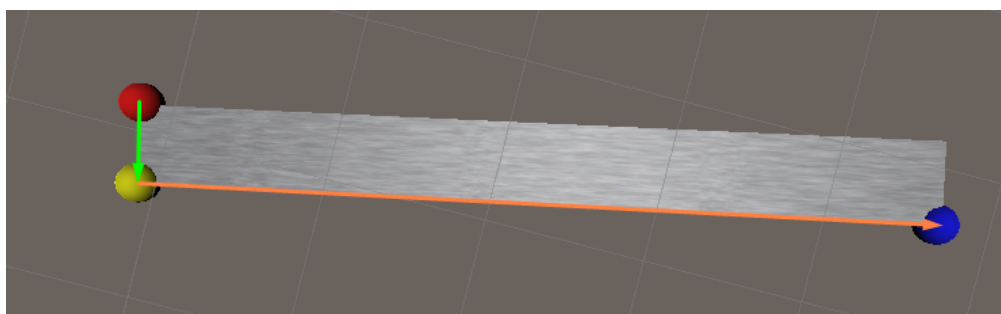
Figure 4.6: Three corners of a shoulder segment and the vector between them.

4.5   Separating Tunnel Lay-Bys From the Tunnel

One of the problems encountered was how to separate the segments in the tunnel lay-bys from the other tunnel segments. A tunnel lay-by is defined as "a space outside the carriageway of a tunnel reserved for temporary parking, for example in emergencies or for service vehicles" [27, p. 70]. In the procedural modeling of the tunnels, the lay-bys

are generated after the tunnel is created and then merged into the tunnel. As a consequence, it is possible to control how the segments in the lay-bys are generated separately from the other segments. Since the different tunnel segments can be distinguished by material, a simple solution to the problem was to rename the materials of the segments. This was done by duplicating each of the materials and renaming the duplicate, for example copying *road.mat* and renaming it *em_road.mat*. Then it was just a matter of changing the code generating the tunnel lay-by to use the new materials instead of the old materials. The `emParks.py` script is responsible for generating the tunnel lay-bys. The scripts calls upon the `profiles.createProfileModel` function. A Boolean was added to this function call that when true was used in the `profiles.py` script to tell functions they were generating a tunnel lay-by. The Boolean was checked in all the *_create* functions in `profiles.py`: `_createRoad`, `_createRoof`, `_createSphericalWalls`, `_createStragihtWalls`, or `_createFlatRoof`, and if true the new emergency material was applied instead of the normal material. The `addRoofCube` function in the `emParks.py` file was also changed to use the new material instead. Thus, distinguishing the emergency segments from the other tunnel segments by comparing the material name is straightforward in Unity.

4.6   Placement of objects

When the 3D model of a tunnel is procedurally generated and imported into Unity, there are placeholders in the tunnel which are supposed to indicate where various different objects should be placed. As mentioned in subsection 4.1.1, placeholders do not have global coordinates and are stacked.

Figure 4.7 shows an example where a yellow *Belysningspunkt* placeholder is marked in the hierarchy and stacked with all the other similar GameObjects at the end of the tunnel. The placeholder is the orange box inside the tunnel and its position is at (0, 0, 0) marked by the triangles at the other side of the tunnel. The same problem of missing global coordinates applies to most of the other placeholders as well. Since the placeholders are not placed correctly inside the tunnel, it was decided not to use them as a part of this project. Therefore they are destroyed by the program when a tunnel model is loaded. Instead of finding each placeholder inside the tunnel and changing it with the object's prefab, other methods were implemented to instantiate the objects inside the tunnel. Consequently, objects were not placed at their correct location but instead placed in a recurring pattern.
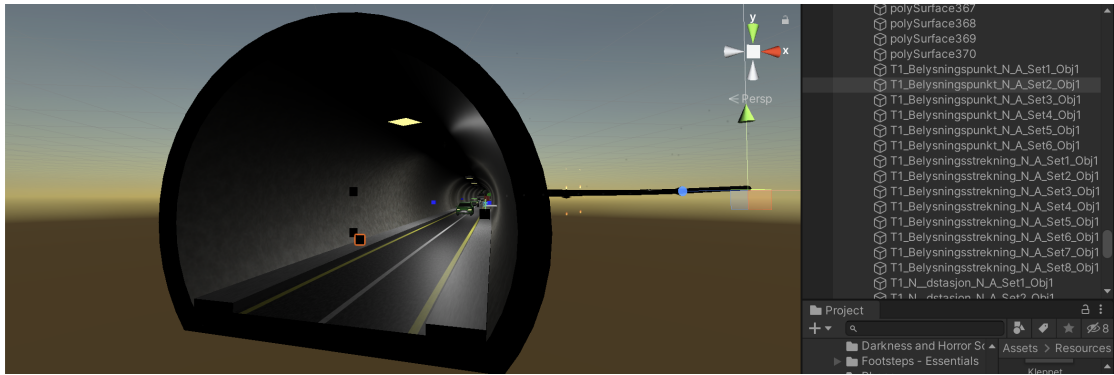
Figure 4.7: Example of a placeholder placed wrongly in a tunnel

4.6.1 Lights

In the script `TScene.cs` lights are generated in the method `GenerateLighting`. The method loops through each wall in the tunnel and instantiates a light prefab at the top of every other wall segment. Figure 4.8 shows where the light prefab is instantiated in the hierarchy and how it looks from a player perspective. In this example a clone of the Prefab *littleStairLight* is instantiated as a child of the *polySurface287* GameObject, which is the wall segment marked by the orange lines in the scene view.
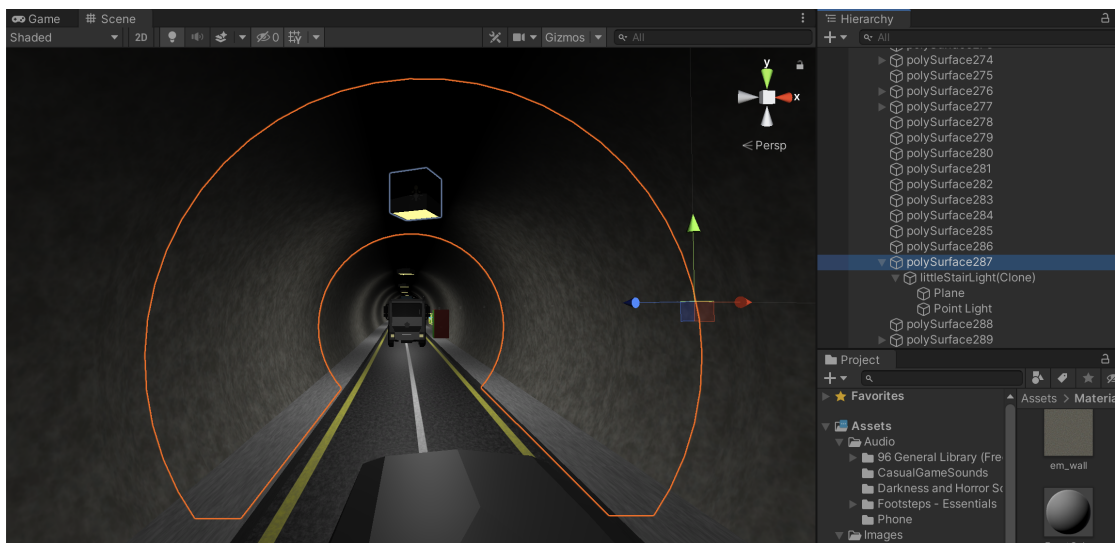
Figure 4.8: Visual of lights in the tunnel

4.6.2 Emergency stations

In `PlaceEmergencyStations.cs` emergency stations are placed in the tunnel. The *EmergencyStation* Prefab consists of one emergency phone and two fire extinguishers, which the player can interact with. The Prefab is placed on every fifteenth shoulder alternating which side of the road it is placed on. This placement is an attempt to make it roughly realistic for every tunnel and make it possible to get an emergency station from all

the places in the tunnel. Figure 4.9 shows how an emergency station looks like from a character perspective.

Figure 4.9: Emergency station from a character perspective.

4.6.3   Tunnel exits

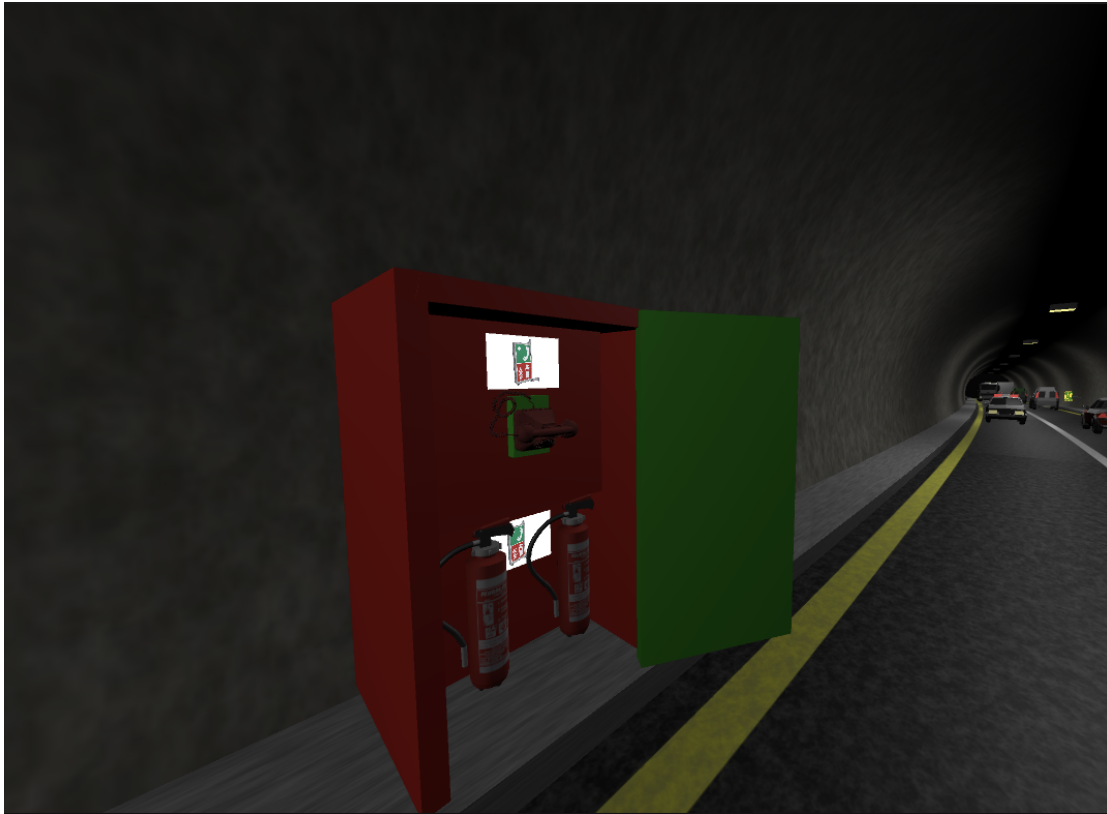All tunnel tubes have obviously at least an exit in both ends of the tunnel, the tunnel portal. Furthermore, many tunnels have one or more emergency exits along the tunnel. Some of these might lead to another tunnel tube, others leading outside. As soon as the player character leaves through an exit it is considered safe and the game ends. It does not matter where the door leads.

Tunnel portal exits

The tunnel portal exit is placed in the `ExitCollider.cs` script. It grabs the first and last road segment, then instantiates a huge white box at the center of the road, covering the tunnel. The box has a Collider Component that triggers when the player character collides with the box, calling the exit method from the `ExitGame.cs` script. Making the box white gives somewhat of a resemblance of being inside a dark tunnel looking out on the bright outside. Furthermore, the placement being at the center of the road means the exit is not actually at the tunnel opening but rather a few meters inside. This simple solution sacrifices a bit of accuracy for easier implementation, but it was deemed that this inaccuracy does not matter much. This simple solution entails a small inaccuracy.

The researchers deems that this acceptable as it makes implementation much easier and the inaccuracy is insignificant.
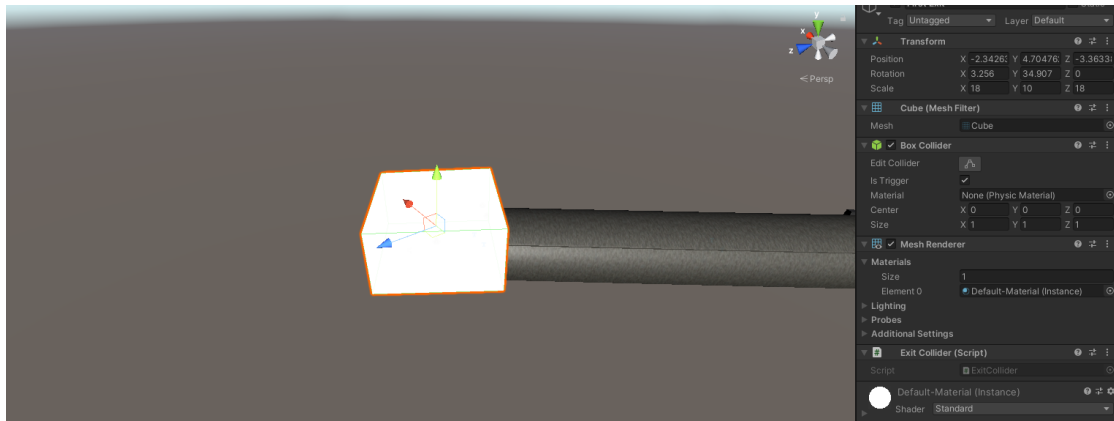

Figure 4.10: Exit collider in white from outside a tunnel.

Figure 4.11: Exit collider in white from inside a tunnel.

Emergency exits

The placing of the Emergency Door Prefab is done in the `PlaceEmergencyDoors.cs` script. Each tunnels emergency rooms always consist of either 5 or 10 segments. There are always five emergency wall segments in a row, but sometimes two rows are stacked on top of each other for a total of ten segments. Meaning the index of the middle segment in the room is always predictable when first checking how many segments the room contains. The `PlaceEmergencyDoors.cs` script checks how many emergency rooms and walls there are in the tunnel. It places the door on the correct segment by passing the index as an argument to the `PlaceDoor.Place` method.

4.6.4 Directions signs

The script `DirectionSigns.cs` has the tasks to calculate the distance to the closest exit and place the sign in the tunnel. Figure 4.13 shows how the direction signs are

viewed from a player perspective. The sign is a Gameobject made up by an Image Component and a Canvas with a TextMeshPro - Text (UI) Component. The idea is that there are signs placed out regularly on the walls of the tunnel, showing the distance to the closest exit in both directions. At every fifth road segment in the tunnel, the script finds the distance to both tunnel exits and checks if there is an emergency exit closer in each direction. The shortest distance calculated in each direction is written in the Text Component of the Prefab.

4.7 Generating an accident and traffic in the tunnel

When generating traffic, the main idea is choosing a random spawn point for the character not too close to the tunnel portal. From here an accident is generated around the character and traffic is generated for the rest of the tunnel. The accident around the spawn point always has the same elements, one or more blocking vehicles and one or more cars on fire, both depending on the difficulty level. After placing the character on its road segment, placing the vehicles on the adjacent road segments completes the spawn scene.

Then the regular traffic is generated around these road segments, which is done by instantiating Prefabs from the pool of vehicles. The vehicles are low poly car assets downloaded for free from the Unity Asset Store. The regular traffic had some problems with collisions that needed improving, and the significant improvement came from calculating the width of the tunnel. As some tunnels are very narrow when procedurally modeled, especially as the shoulder segments take up considerable space, it was decided to make an arbitrary limit of 6 meters for the width of a tunnel. If a tunnel is narrower
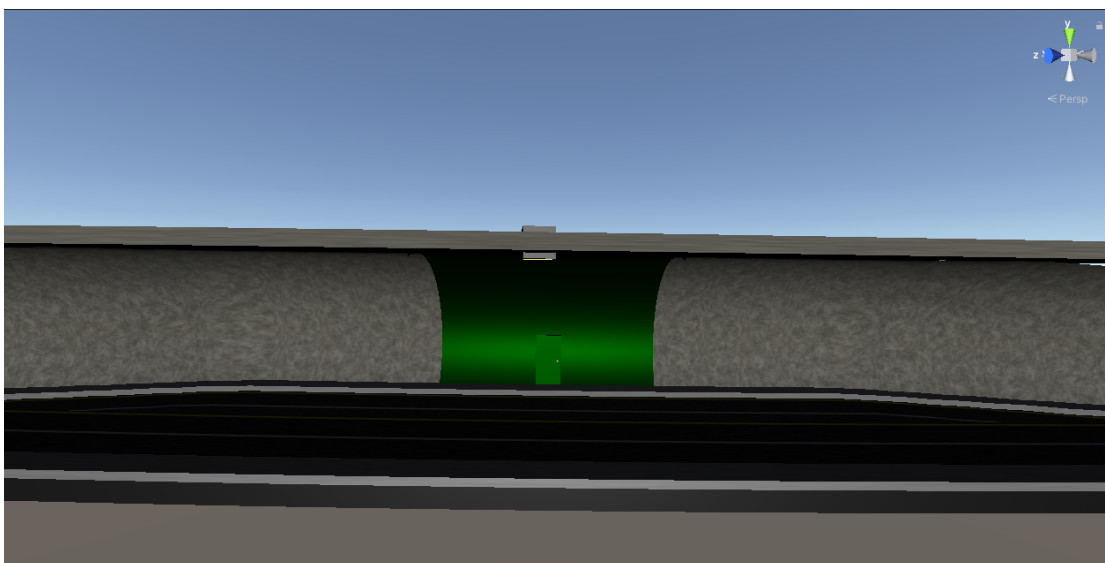

Figure 4.12: A tunnel lay-by with an emergency door on a wall colored in green.

Figure 4.13: Visual of direction signs from player perspective

than the limit, it is marked as a one-way tunnel. In one-way tunnels, the vehicles are just placed in one direction and rotated accordingly, which allowed the cars to instantiate without intersecting. If the tunnel is two-way driven, the vehicles are placed in each lane and rotated accordingly.

Furthermore, the difficulty of the level had to be considered. Again, the idea was to create a spawn scene, but this time construct it differently based on the difficulty level. It was decided to use adjacent road segments to the spawn scene and place either more blocking vehicles or additional cars on fire at increasing difficulty levels, thus creating more dangers. The baseline traffic on *easy* difficulty was decreased to one vehicle on every third road segment, then increased for every difficulty level. Figure 4.14 show an example of how the traffic can look like from a character perspective.

4.8 Interactivity

A great way of learning is learning by doing. Therefore, the player should have the opportunity to use the assistive technologies available in the tunnel. Inside a tunnel, objects that can assist a person can be: the emergency exits leading out of the tunnel, the fire extinguisher that can put out the fire, and emergency phones that can contact the emergency services and alert of the accident. The objects that had some sort of interactivity implemented, and are described in the following sections, where:

Figure 4.14: Traffic from a character perspective

- Fire extinguisher

- Emergency phone

- Emergency door

- Emergency stations

In reality, a person interacts with objects most of the time using their hands, which is the case with all the objects stated above. Similarly, the game lets the user interact with all the mentioned objects equally, that is, looking at any object and pressing the same button for all objects. This was done by creating a parent class, `Interactable`, that all interactive objects can inherit. This base class defines a few methods that are called when the user interacts in different ways with the objects. Among other things, these methods take care of position, gravity, the possibility of dropping the item interacted with, and the placement of objects in the hierarchy. The interaction is called from the `CameraController` script, which calls upon the correct method. For example when pressing the key mapped to `Interact`, `Interaction` might be called. Further details about how the camera controller does this is described in subsection 4.10.3. As said, any object that is made interactable can extend the Interactable class. The child class can then override any of the methods to implement distinct functionality. For example in `ExitDoor.cs`, which extends Interactable, the class overrides the *Interact* method of the

parent class and instead ends the game when the Interact method is called.

4.8.1 Fire extinguisher

If the character stands close to a fire extinguisher, looks at it, and presses interact, the character picks up the extinguisher. The extinguisher now has the potential to put out the fire. The `Extinguisher.cs` script is attached to all Fire Extinguisher Prefabs. When the character holds the extinguisher and presses interact, the script makes sure the extinguisher is not empty and then sprays a foam of particles. Figure 4.15 demonstrates what it looks like when the character interacts with a fire extinguisher and spray foam from it.

Figure 4.15: The character interacts with a fire extinguisher.

`Fire.cs` is attached to the Fire Prefab. In the script, there is an *OnParticleCollision* method to handle the extinguishing of fire in the script. When other particles collide with the fire particles, OnParticleCollision is triggered. If the particles colliding with the fire has the "foam" tag, the ParticleSystem of the corresponding Fire Prefab will stop playing after some time. As stated before, there can be several Fire Prefabs scattered on a vehicle. To extinguish the vehicle fire totally, all these Prefabs need to be hit by the foam.

4.8.2 Emergency phone

Interaction with the emergency phone works the same as with the fire extinguisher. Since there is no possibility of interacting directly with the Emergency Phone Prefab, the Phone GameObject was made using UI Components. To simulate that the character uses the emergency phone when it is interacted with, the Phone is displayed when interacting with the emergency phone. This is demonstrated in Figure 4.16. The player can dial in a number and press the green call button. If the number is one of the Norwegian emergency numbers, a voice mail is played, telling them that this is a self-rescue mission

and they cannot get any help. Otherwise, an error sound starts playing. An emergency phone would immediately connect to a call center in a real-life situation without the need to dial any number. As such, the main point of the emergency phone is to practice remembering the emergency numbers under stress and keep the player occupied for a little while in a stressful situation.
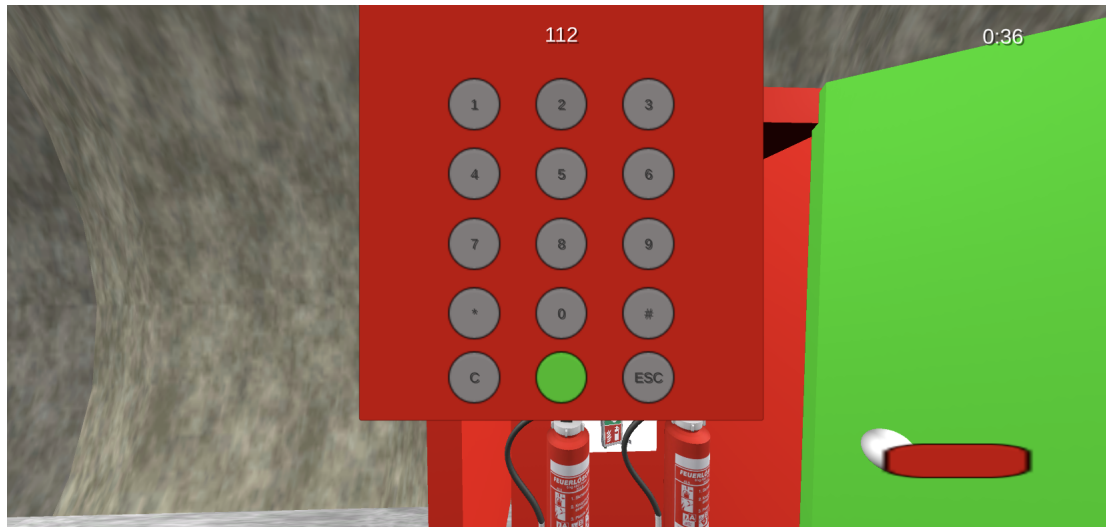

Figure 4.16: Interaction with an emergency phone.

4.8.3   Doors

In tunnels, the use of doors is most of the time, either with opening up the emergency stations or opening emergency exits to get to safety. There are different ways to interact with these doors. For the emergency station, the character only needs to be inside the Collider of the door for it to swing open, letting the user interact with the emergency station. As for the exit door it inherits the Interactable class, which means that the player can look at the door and interact with it. The `ExitGame.Exit` method is called when the player interacts with the door, ending the game and returning to the main menu. As mentioned earlier the game is then ended because the player is considered to be in safety, sending the user back to the main menu.

4.9   The simulation of a car accident

The simulation of a car fire is an important task in this thesis. The idea is that the player starts the game and spawns at a random place inside the tunnel next to a car that is on fire. The fire itself will be dangerous for the player and it will start to make smoke that will be hazardous for the player. The fire will always start at the engine of the car and will spread to the rest of the car as well as to cars in the vicinity. Figure 4.17 shows a case where the fire has spread to the whole car.
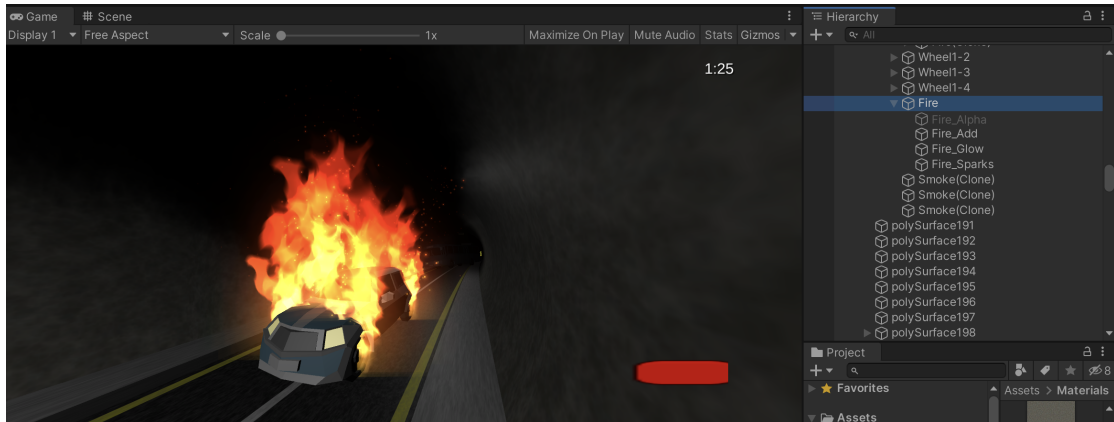
Figure 4.17: A car on fire from a player perspective

4.9.1 Making fire

The *Fire* Prefab is imported from Unity Assets. The Prefab itself is highlighted in the hierarchy of Figure 4.17 where the structure of the Prefab is shown. The Fire is made using Unity's *Particle System* Component where each layer of the Prefab emits different particles to make the fire look genuine.

Management of tunnel fire is done in the `TScene.cs` script with two methods. When a car is on fire, the method `SpreadFireOnCar` calls itself with a coroutine to make sure fire is spreading to the whole car in a certain amount of time. Cars that are located close to a car that is on fire should also burn after a certain amount of time. When a car starts to burn, it calls `SpreadFireToCar` with a coroutine. This method seeks any other car or cars close enough to be affected by the fire.

4.9.2 Making smoke

The *Smoke* Prefab is made using the Particle System Component. The Prefab was edited and shaped in the Inspector window. The Prefab emits a hundred particles of smoke every tenth millisecond. The Particle System Component uses gravity and force to make the smoke particles move in the directions desired and collision detection to make them stay inside the tunnel. Each time a smoke particle collides with the player character, the coroutine `PlayerHealth.ShowFog` is called. The method increases the fog density of the game, resulting in a visual effect of reduced visibility. An example of this is shown in Figure 4.18 as experienced by the user.

4.10   Character

4.10.1   Player Prefab

The user-controlled character is a Prefab called *First Person Player*. The Prefab has a body that can be controlled. It has a camera that controls what the user can see, an invisible GameObject *Ground Check* placed at the bottom of the body, and an invisible GameObject *Hand* placed in front of the body. Furthermore the Prefab has a few other components attached, like a Character Controller and a Box Collider.
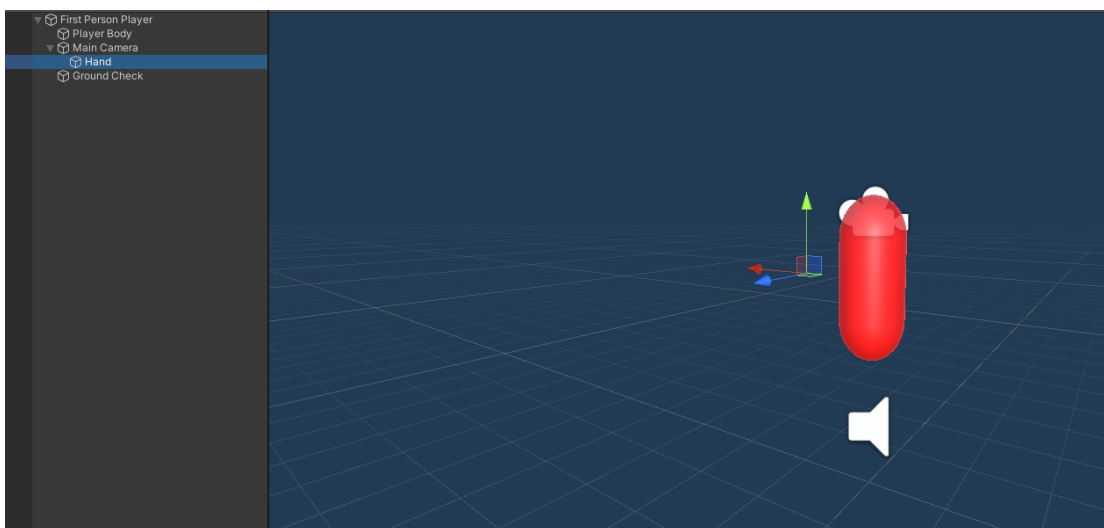

Figure 4.18: Visibility while in smoke.

Figure 4.19: The Player Prefab, with the hand object marked.

4.10.2 Character control

For a self-rescue game, it is vital to have the ability to move around and control the character. Everything concerning the character's movement is in the `PlayerMovement.cs` script. This script handles all the physics and input regarding movement, all in its `Update` method. The *Player* Prefab has a `CharacterController` component that the script uses to control the character. The CharacterController class is a built-in Unity class with all the functions needed to move and control a character. When the scripts get input to, for example jump, it calculates the necessary physics to jump and sends that to the CharacterController class and its `Move` method.

4.10.3 Camera control

The camera controller, implemented in the `CameraController.cs` script has a few functions. Similarly to how the character controller controls the player movement, the camera controller controls the camera on the *Player* Prefab. Furthermore, the camera controller also handles some of the logic concerning interactions and holding objects.

Camera movement

As mentioned, the camera is a child object on the *Player Prefab* that is placed at eye level. As it is a Unity Component, it has a Transform component with the rotation property. Therefore, the camera controller can receive any inputs the user does with a computer mouse and use it to change the camera's rotation accordingly, letting the user look around. The script also has a Boolean that can be used to freeze the camera.

Interaction

The camera controller also handles much of the logic concerning interactions. To help with interactions, the controller uses Unity's *Raycasting*. Raycasting is a technique that sends a ray from a given spatial coordinate in some given direction, returning True if the ray collides. When specified to do so, a reference to the object collided with is also returned. However, Raycasting can become a computationally demanding operation quickly and should be used with care. The camera controller uses Raycasting to check if the camera is looking directly at an object that is marked as interactable. It does this by casting a ray from the center of the camera in the direction the camera is facing, returning the first object it intersects. To only collide with the relevant objects, the ray casted uses a *LayerMask*, the LayerMask tells the ray to only check for intersections with objects that are in the specified layer.

In order to only use Raycast when it is necessary, the script casts in two circumstances. The first is constant from the time the character loads into a tunnel scene and until the player looks at an object that is interactable. At this point, a text pops up on the screen telling the user how to interact with objects, and the camera controller stops Raycasting. The second is whenever a user presses the interaction key, as defined in the Input Manager. When the specified key is pressed, a ray is cast and returns any interactable object that it hits.

When the user interacts with the object, what happens depends on the object, as described in section 4.8. One example is, however, what happens with a fire extinguisher. Upon interacting the fire extinguisher is placed as a child object of the earlier mentioned *Hand* GameObject that the *Player Prefab* has, effectively moving its position The camera controller now knows that the Player is holding an object, and instead of trying to pick up an object, it will now call the *Action* method on the fire extinguisher.

4.11   Intractable menus

In game development, having an intractable menu is important for the user experience. It can let the user customize settings to their liking, hopefully improving the overall user experience. Using *TextMeshPro - Button, Slider and Dropdown* Components instead of the Unity default gives more room for customizing the layout. Changing the alpha values of the buttons to emphasize when they are highlighted and pressed makes the menu more visually intuitive. It can be changed in the Button Component settings.

4.11.1   Main Menu

The idea for the main menu is to have multiple menus layered on top of each other inside the canvas, using UI buttons to toggle the menus on and off. The main menu has a sub-menu where the user can adjust sensitivity and volume called options/settings. It also has a level selection sub-menu where it is possible to start the game in addition to the main menu. To make the layering of the menus easy, in Unity, there is the option to duplicate the MainMenu GameObject to create the other menus. The next step is to have the buttons do something. The buttons should take the user to their respective menus using the layering technique. The *Button Component* in Unity has an `OnClick` method by default which can be used to set the *SetActive* property of the menus to *true/false*. Setting the SetActive property of a GameObject to false hides it in the Game View.

The StartMenu consists of a Dropdown element to select the tunnel model the user wants to play. The Dropdown is populated by the `Information.cs` script that fetches

the tunnel names from the *Resources* folder. The play button's OnClick method runs the
`PlayGame` method in the `MainMenu.cs` script as described in section 4.2.

Tunnel names

Originally the tunnel model files were named their ID from the NVDB. Thus, the
Dropdown consisted of files named "79610296" and "81507454" and so on. Using numbers
as names is not particularly user-friendly, and therefore it was decided to change the
procedural modeling algorithm to use the names of the tunnels instead. A tunnel model
is exported and named in the `mayaProc.py` file in the `exportTunnel` function. A new
function `getTunnelName` was implemented that opens the text file corresponding to the
ID of the tunnel model. The text file contains the data set from the NVDB on that
tunnel. `getTunnelName` then retrieves the tunnel name from the data set and returns it.
The retrieved tunnel name was then used instead of the tunnels ID, resulting in the file,
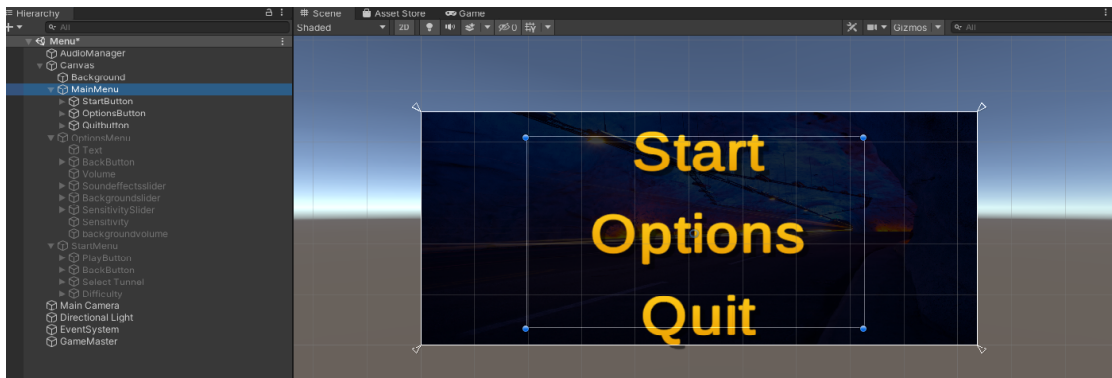and thus the Dropdown, having more understandable names.

Figure 4.20: Hierarchy in the main menu.

The options menu uses sliders to control volume and mouse sensitivity. *Slider* components
in Unity have an `OnValueChanged` method. Using OnValueChanged, the Slider calls
a specified method from a specified script when the value of a Slider changes. The
MainMenu.cs script handles all the changes to the Sliders in the options menu. Therefore,
the script is attached to the *OptionsMenu* GameObject, letting the OnValueChanged
calls be bound to their respective methods in the script. The Slider Component is shown
in Figure 4.21. This way, the MainMenu.cs script can handle all the values change to the
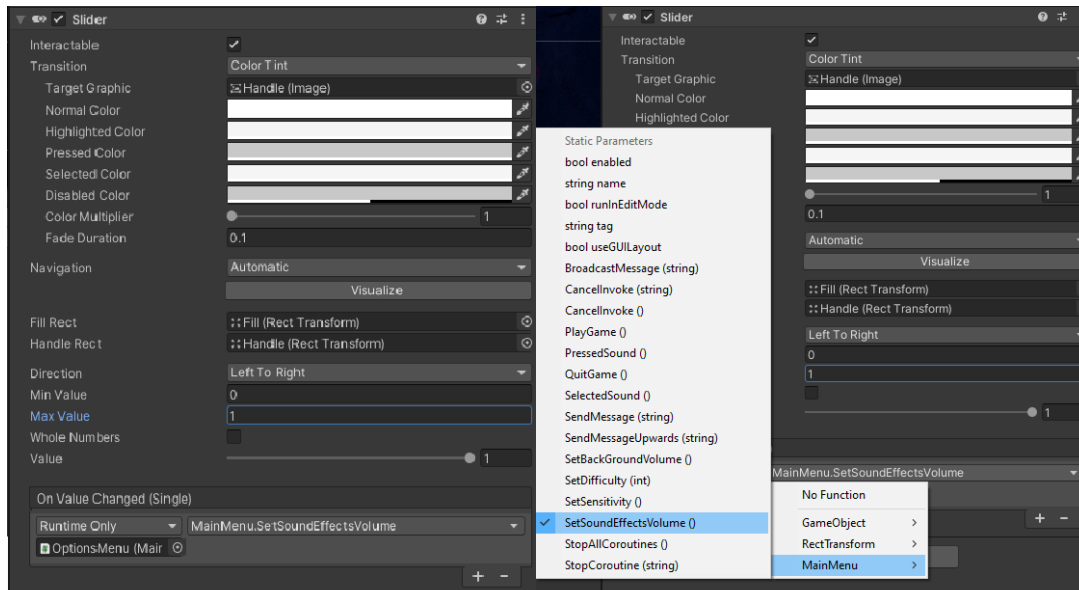game settings and save the settings in the *PlayerPrefs*.

Figure 4.21: Slider Component.

4.11.2 Pause Menu

The Pause Menu is an in-game menu, implemented to be accessible by pressing the tab key on the keyboard. By creating a GameObject on the Canvas in the Tunnel Scene called PauseMenu, the menu's look uses the same principles as the main menu UI. The `PauseMenu.cs` script handles pausing and resuming the game by changing the time scale of the program. The time scale is the time between each frame. Setting the time scale to zero stops the game from progressing, no new frame is rendered. Meaning none of the Unity methods, like `Update` is called. `FixedUpdate` and coroutines are also stopped [28].

4.12 Audio manager

Having audio in the game is very important. It makes the UI more intuitive and objects more realistic in the game. Being able to control when audio is playing is critical to achieving a realistic feeling, and the script `AudioManager.cs` does this. It is attached to a GameObject called AudioManager in the Menu Scene. The AudioManager.cs script uses a singleton pattern and is calling `DontDestoyOnLoad` in its Awake method to prevent it from getting destroyed when switching between scenes. Not destroying the instance of the AudioManager allows the script to have complete control over the audio in both scenes, even when switching back and forth.

The Sound class gave audio clips names and stores playoff settings using the AudioManager. The AudioManager has an array `Sounds`, of the `Sound` class, as seen in Figure 4.22. The
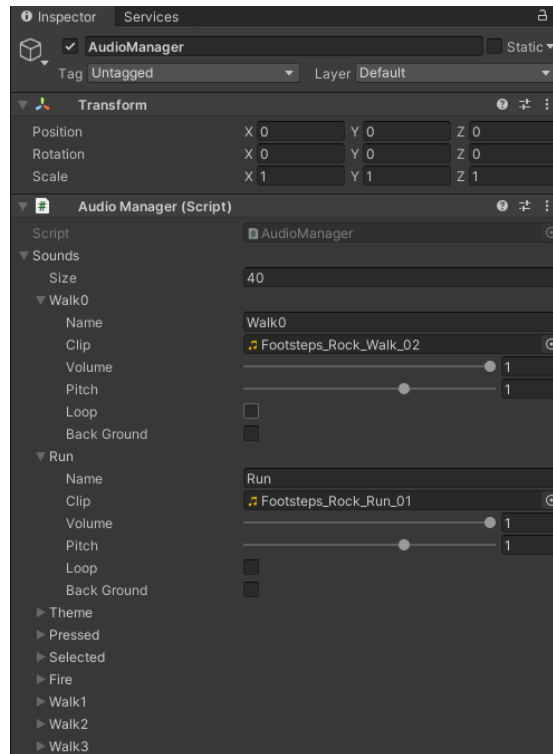
Figure 4.22: Audio manager.

`Play` method in the AudioManager plays a sound by its name. The Play method allows for playing specific sounds in other scripts. The AudioManager also has a `Stop` method that stops playing sounds. Both these methods have overload, which for the Play method means in addition to playing sounds by its name, it has optional input for a number of variants to play. The Stop method stops all sounds playing if there is no string input and stops a sound with that name if there is input.

4.12.1 Menu audio

In the `MainMenu.cs` script, the AudioManager is used to play music on startup and stop it when the next scene loads. The MainMenu script also has methods to play a sound when buttons are pressed or selected. For the latter to work, an Event Trigger Component has to be added to the buttons as the Button Component only has `Onclick)` as default.

4.12.2 Footsteps audio

To create a more realistic experience, the sound of each footstep the character takes varies slightly. Using multiple audio clips of footsteps, running and jumping, and randomizing which one is played each time, the result is realistic sounding footsteps. The `Footsteps.cs` script was created to control when a clip should be played. The script has all its Boolean checks in the `Update` method that is called every frame, and if true, it plays the appropriate

sound using the `Play` method in the AudioManager. All audio clips used for the footsteps were downloaded for free in the Unity Asset Store.

4.12.3   Fire audio

The sound of the car burning was attached to *Car_1_Blue_Variant_Fire.prefab* by adding an AudioSource Component to the Prefab. Checking off *loop* and *play on awake* ensures it starts playing when the Prefab is instantiated and continues to loop. Setting the spatial blend to 3D gives the sound the most realism for a 3D application. The audio clip was downloaded for free from Freesound.org [29].

4.13   In-game UI

While in-game, multiple UI elements are shown, two of these elements are a health bar showing the player's health and a timer showing how long the current game has lasted.

4.13.1   Player health

To give the player a feeling of how dangerous the fire and smoke are, the player is given a certain amount of health. When health reaches zero, the character dies and the game is over. The character health is handled in the `PlayerHealth.cs` script. Every time the character is in contact with smoke or fire particles, the method `OnParticleCollision` is called and the health is decreased. How much health the character has left can be seen on the health bar, named *Health*. The health bar can be seen at the bottom right at Figure 4.17 and is a UI element using the Image Component. The `HealthBar.cs` script is attached to the health bar. Every time the character loses health, the script updates the health bar to the amount of health that is left.

4.13.2   Timer

It was decided to implement a timer to the game telling the user how the current tunnel scene have been active. The timer is visible at the top right corner in Figure 4.17. It uses the Unity variable `Time.timeSinceLevelLoaded` which gets the time in seconds since the current scene was loaded, and converts it into a more readable format.

4.14   Difficulty

The game has a difficulty setting that the user can control in the main menu. The setting and helper methods are handled in `SETTINGS.cs` Using the difficulty setting, it is possible to control many parts of the scene. For example, as described in section 4.7, the number of vehicles blocking the road depends on the difficulty setting. Moreover, the difficulty

setting is implemented as an *enum*(enumeration), making it practical for controlling many parameters in the game. This is done by using the integer value of the enum as a factor in the parameters. Some examples are:

- Multiplying a base decrease in visibility by the difficulty setting, when the character is inside smoke.

- Adding the difficulty setting to the iterator when placing vehicles, meaning vehicles are placed more or less frequently as the setting is higher or lower.

- Dividing the time to wait before fire spreads on a car by the difficulty setting.

- Controlling the effectivity of fire extinguishers by dividing on the difficulty setting.

Chapter 5

Discussion and Future Directions

This chapter will:

- Explain how the project was organized.

- Discuss the research.

- Explore future directions.

5.1 Organizing the project

When planning and organizing how the game should be developed, user stories were collected to ensure functionalities desired from a user perspective where considered. This made the basis for how the game should be developed and also what it should look like at the end. A user story is a short and direct description of a feature told by a person wanting the new capability [30]. Examples of user stories that were made are "I want to be able to fail the game, if I die. 30" and "I want the player to be placed inside of the tunnel 4". The number following the user stories is the total hours estimated to make the feature.

In total, seventy-five user stories were collected and ranked by importance. Then the time to complete the user story was estimated. This way, it is possible to look at the time available in the team and see which user stories are possible to finish within the deadline. The goal were to create a minimal viable product and the deadline for this was set to 15. April. To get a minimal viable product, the team had to finish the first eighteen user

stories with all the essential features in the game. After that, the next month consisted of implementing more user stories and features built upon the foundation laid in the first month.

GitHub was used to organize these user stories into issues, with some issues being divided into several sub-issues. Researchers could then be assigned to issues. The issues were placed in a project, making it easier for everyone involved in the project to see what the other researchers were working on, and what features needed to be done.

5.2 Interpretations of the research

The research shows that it is possible to create a self-rescue game in Unity, using tunnel models created procedurally. The research shows that Unity is an excellent choice of a game engine, as it has a lot of built-in methods for 3D game development, easy access to Assets, and support for VR.

5.3 Implication of the research

The thesis on Procedural 3D Modeling of Road Tunnels made it possible to generate any single tube tunnels in Norway procedurally [5]. The Procedural and Parametric 3D Modeling of Road Tunnels thesis imported these tunnels into Unity and used scripts to replace the placeholders with 3D models such as lights and an emergency station and implementing a VR environment, where the user could extinguish the fire that appeared with an interactive fire extinguisher [23]. In this thesis, this is taken a step further by populating more objects to the tunnel and implementing features that remind of a typical game.

The goal of this thesis was to make tunnel safety practicing an immersive experience and inspire others to explore the possibilities tied to this topic. Firstly, Bouvet has made a full-scale 3D model of the Ryfast tunnel used for practicing tunnel safety in VR support [24]. However, the research of this project is unique because it allows for safety training in any given single tube tunnel in Norway. This implies that training can be done virtually without closing down the tunnel and that the general public can train on their most relevant tunnels. Also, it is possible to build upon and improve these models and the games created from them.

5.4 Limitations of the research

The researchers have tested a limited number of tunnels, so there might be unknown bugs when the game is used on some tunnel models. This might be due to the tunnel model being generated poorly or differently than the tunnel models used in this research. The research considered tunnels with only one tube, as the procedural modeling algorithm could not generate multi-tube tunnels. Another limitation is the placement of objects. As the virtual placement is approximate and arbitrary, it will most certainly differ from reality.

5.5 Future Directions

This section will highlight some of the planned ideas that were not fully explored because of time restrictions, resources, and other limitations. It will also highlight improvements the researchers think are worth implementing for a better user experience. The section will also give some pointers on what could be interesting to implement from a developer's standpoint. Lastly, some necessary updates of the software used in this project are mentioned.

5.5.1 VR

Virtual reality (VR) would enhance the realism aspect of the game and offer the user a more realistic approach to practicing self-rescuing in a tunnel, which has been the goal for this game. The researchers did not go into the direction of VR because of the Covid-19 situation, which made access to the university's VR equipment very limited. Changing the game into a VR game would only require changes to the character and camera controller.

VR controller

A VR controller uses `Raycasting` like the FPS character controller to select objects, but the movement is different. In VR, "Point & Teleport" locomotion is often to prevent motion sickness [31]. In this game, teleporting would not be worth implementing as the self-rescue game aims for realism. Teleporting is not very realistic—other ways to move are walk-in-place and joystick locomotion techniques. The Controller requires minor changes only, but a good understanding of the Unity *Input Manager* (The `UnityEngine.Input` class) is recommended. Another direction is the *Input System Package*, which implements a system to use any Input device to control the Unity content. This is a replacement for the classic Input Manager and is intended to be more powerful, configurable, and flexible.

VR Camera and sound

In Unity, the game engine does a lot of the work for the camera and related input. After enabling VR in Unity, it automatically renders to the head-mounted display and tracks user head movement as input. When it comes to audio, *Audio Spatializer* plugins are worth exploring. The plugin changes how sound is transmitted from an audio source to its surroundings.

5.5.2 Improvements

First, some improvements can be made to the placement of emergency doors. The doors will sometimes, in some tunnels, be rotated incorrectly. It is possible to create an Objective class for all the objectives and a more visually appealing UI for the objectives in-game. So far, the variations in difficulty are limited to a diverse population of blocking vehicles and vehicles on fire in the spawning scene. Finding a way of instantiating the vehicles more randomly on segments will make it more authentic. Exploring additional ways of making a variation in difficulty can be an exciting task for future projects.

The subsection 4.9.1 explains how vehicles can spread this to other vehicles. If the fire extinguisher stops a vehicle fire, the vehicle will still spread fire to other vehicles. For future developers, this is a potential for improvement.

5.5.3 Excess user stories

These include populating the tunnel with other people, then rescuing another person can be an objective to practice. Even if self-rescuing is the principle, cooperating with other people is still a key to making it out alive. Implementing this feature can give the user practice on how to act when there are other people around. Giving the emergency phone other purposes than just practicing the emergency numbers is another possibility. One is a dialogue system where the player can say that there is a tunnel fire, wherein the tunnel they are, and then score points based on what was said. The player can then get information that help is on the way, but it will not be there for some time as an excuse to continue the self-rescuing.

5.5.4 Possible improvements to the procedural modeling

Section 4.1 explains the problems that occur with the procedural modeling of tunnels. Because of these problems, much of the project was about finding workarounds to these problems. Some improvements that improved the procedural modeling are explained in section 4.3. In the future, it is recommended to look further into making tunnel models that are easier to handle when used in Unity. If it is possible to give segments in the

tunnel transformation and rotation, it is easier to handle general placement. Similarly, giving placeholders their correct position in the tunnel will allow developers to populate tunnel models with a more exact position for the objects.

5.5.5 Change to NVDB version 3

As stated in section 2.4, this thesis uses NVDB API version 2 (v2). The applicable version is NVDB API version 3 (v3). Although the two versions are very similar, v2 will not have support after August 2021 [6], therefore it is soon necessary to change the version. When implementing support for v3, some actions need to be considered. The query string in `nvdb.py` would need to be changed since v2 and v3 uses different query formats. The new format requires the developer to specify the version of the queried object. When retrieving data from NVDB v2, all properties are assumed to have name and value keys. In v3, some properties do not have the value key. Therefore some modifications in the nvdb.getData function needs to be done when changing version. Read the Procedural 3D Modeling of Road Tunnels thesis for more information about the function and what should be done [5].

5.5.6 Change to Python 3

As stated in section 2.5, Maya has a Python library with an API that is used for this thesis. Python 3 was released during this project and is now the default mode for both Windows and Linux [10]. Python 2 no longer has support for macOS [10]. These changes make an update to the newest Python version necessary

Chapter 6

Conclusion

The thesis aimed to use procedurally generated 3D tunnels to create a game in Unity to practice self-rescuing training during emergencies involving fire. This thesis has expanded on earlier research on tunnel safety, showing that it is possible to create a game to practice self-rescue, using procedurally modeled tunnels with shortcomings. The game created in Unity can populate a tunnel model with objects and create a playable self-rescue scenario. Further studies could improve the tunnel models to include placeholders at more correct locations.

A secondary objective was to place relevant objects in the tunnel. This objective was achieved by placing objects like lights, emergency stations, signs, and emergency exits at approximated locations.

A secondary objective was to have interactive objects. This objective was achieved by creating and placing interactive objects like fire extinguishers, emergency doors, and emergency phones.

A secondary objective was to have multiple difficulty levels in the game, and the objective was addressed by creating levels with more fire and blocking vehicles to increase difficulty. Traffic was also increased in line with the difficulty.

A secondary objective was to implement virtual reality. This objective was dropped due to a lack of access to VR equipment because of Covid-19.

6.1 Evaluation

When this project started, we had minimal experience with game development and no experience with the Unity platform. First of all, we have learned the importance of solid preparations and maintaining communication when working on a project of this scale. We have also learned that it is challenging to learn all the features of such a big platform as Unity has to offer, but learning them along the way is a good strategy. From looking at earlier projects, we have learned the importance of making a good foundation for future work. We believe this was a challenging task and a positive experience.

List of Figures

List of Tables

Appendix A

Scripts

The scripts used in the project are categorized into folders. Below is a list of these folders, the scripts located in them and a short explanation of what they do. All the Unity folders can be found in the Assets/Scripts folder. The section A.6 is not a folder, but merely a list of other scripts that were used in the Unity part of the project. `ExitGame.cs`, `Information.cs`, and `SETTINGS.cs` are located in the general scripts folder, while the `TunnelImporterScript.cs` is located in the Assets/Resources/Tunnels folder. The section A.7 is a list of the files that were changed, more than a simple clean up of whitespace, in the procedurally modeling algorithm.

A.1   Interaction

`DoorCollision.cs` - Logic for opening doors when triggered.

`ExitDoor.cs` - Extends `Interactable.cs`. Overrides the `Interact` method.

`Extinguisher.cs` - Extends `Interactable.cs`. Contains methods for using the fire extinguishers.

`Fire.cs` - Logic for fire "health" and collision with foam.

`Interactable.cs` - Parent class for implementing interactable functionality.

`Phone.cs` - Extends `Interactable.cs`. Logic for using the Phone GameObject.

A.2   Player

`CameraController.cs` - Methods for controlling the camera and its interaction with objects.

`Footsteps.cs` - Plays sounds based on character movement.

`PlayerHealth.cs` - Logic related to character health and character collision with particles.

`PlayerMovement.cs` - *CharacterController* calculates the movement related physics.

A.3   Sound

`AudioManager.cs` - Methods for controlling sound and music.

`Sound.cs` - Class to implement sounds through the Unity Developer UI.

A.4   Tunnel

`TAssets.cs` - Getters and setters for Assets used by multiple scripts.

`TData.cs` - Logic to generate and get reference to different type of segment and objects.

`TMath.cs` - Methods to calculate vectors and scalars from segments and objects.

`TScene.cs` - Main script to set up and control a Tunnel scene.

A.4.1   Placement

`DirectionSign.cs` - Calculates where to place direction signs and places them at the calculated position.

`ExitCollider.cs` - Places a box, with a collider trigger, on the specified segment.

`PlaceDoor.cs` - Places doors on specified wall segments.

`PlaceEmergencyDoors.cs` - Logic for determining where to place doors, call `PlaceDoor.cs` to place them.

`PlaceEmergencyStations.cs` - Calculates where to place emergency stations and places them at the calculated position.

A.5 UI

`Clock.cs` - Logic for counting and displaying an in-game timer.

`Fade.cs` - Helper methods to fade in and out text.

`HealthBar.cs` - Displays the health bare based on character health.

`MainMenu.cs` - Methods to control the main menu, including loading a tunnel scene.

`PauseMenu.cs` - Controls the pause menu, including freezing the game.

`PhoneClickHandler.cs` - Logic for interacting with the *Phone* GameObject.

`SelectTunnelDropdown.cs` - Loads all the tunnels in the *Resources* folder, populates the drop-down list in the main menu and returns the selected option.

A.6 Other

`ExitGame.cs` - Methods to exit scenes and quit the game.

`Information.cs` - Helper methods to retrieve tunnel model paths.

`SETTINGS.cs` - Getter and setters for many settings used by other parts of the game.

`TunnelImporterScript.cs` - Controls tunnel models that are imported into the project, changing how the material is loaded.

A.7   Procedural modeling

`emParks.py` - Logic to generate tunnel model lay-bys.

`mayaProc.py` - Main file responsible for generating the tunnel models.

`profiles.py` - Logic for generating each specific segment, such as a wall segment.

Appendix B

Running the Unity Application

1. Install Unity from `https://unity3d.com/get-unity/download`.

2. Download the source code from `https://github.com/TunnelSafety/3D-tunnel/tree/self-rescue-game` as either:

   (a) The build folder, lets the user run the game, but without options to add more tunnels. If this option is selected the game can be played simply by opening the `3D Self-rescue game.exe` file located in the folder.

   (b) The package folder, includes a package that can be imported into any Unity project, preferably a new one. Unpacks into the project with all source code and assets.

   (c) The project folder, a copy of the project files including all source code and assets, equal to the package, but stored differently.

3. If either of the two second options were selected, open 3D Self-rescue game folder with Unity.

4. See [5] on how to generate any extra tunnel models, preferably using the updated version.

5. Import any extra tunnel model(s) into Assets/Resources/Tunnels.

6. Play the game by either:

   (a) From the editor select the Menu Scene and press the play button.

    (b) Press File > Build And Run, select a folder to build into, press Build And Run in the pop-up window.

Bibliography

[1] Microsoft. *Introduction to the C# lanaguage and .NET.* `https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/`, 2021, January 28.

[2] Unity Technologies. *Coding in C# in Unity for beginners.* `https://unity3d.com/learning-c-sharp-in-unity-for-beginners`, n. d. Retrieved 2021, May 5.

[3] Microsoft Visual Studio. In *Wikipedia* `https://en.wikipedia.org/w/index.php?title=Microsoft_Visual_Studio&oldid=1018900603`, 2021, April 4.

[4] Statens Vegvesen. *Nasjonal vegdatabank (NVDB).* `https://www.vegvesen.no/fag/teknologi/nasjonal+vegdatabank`, n. d. Retrieved 20.04.2021.

[5] Henriksen, S., Simonsen Knutsen, G. & Stava, G. *Procedural 3D Modeling of Road Tunnels: a Norwegian Use-case.* [Unpublished bacehlor's thesis]. University of Stavanger, 2020.

[6] Statens vegvesen. *API-dokumentasjon.* `https://api.vegdata.no/`, n. d. Retrieved 21.04.2021.

[7] Bowling Green State University. *Autodesk Maya Tutorial* [pdf]. `https://www.bgsu.edu/content/dam/BGSU/libraries/documents/collab-lab/Maya-Tutorial.pdf`, n.d.

[8] Autodesk. *MEL Overview.* `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-Scripting/files/GUID-60178D44-9990-45B4-8B43-9429D54DF70E-htm.html`, 2020, December 7.

[9] Autodesk. *Using Python.* `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Scripting/files/GUID-55B63946-CDC9-42E5-9B6E-45EE45CFC7FC-htm.html#WS17956D7ADBC6E7362626226A117AE335D4A-7FFC`, 2018, August 13.

[10] Autodesk. *Python 3 (What's New in Maya 2022)*. `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2022/ENU/Maya-WhatsNewPR/files/GUID-DF43840B-4DB1-43F8-BFD1-97D8D031B91D-htm.html`, 2021, March 24.

[11] GitHub. *Hello World*. `https://guides.github.com/activities/hello-world/`, 2020, July 24.

[12] Unity (game engine). In *Wikipedia* `https://en.wikipedia.org/w/index.php?title=Unity_(game_engine)&oldid=1020877807`, 2021, May 5.

[13] Unity Technologies. *Transform*. `https://docs.unity3d.com/ScriptReference/Transform.htmll`, 2021, April 19.

[14] Unity Technologies. *ParticleSystem*. `https://docs.unity3d.com/ScriptReference/ParticleSystem.html`, 2021, April 19.

[15] Unity Technologies. *MonoBehaviour*. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.html`, 2021, April 19.

[16] Unity Technologies. *MonoBehaviour.FixedUpdate()*. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html`, 2021, April 19.

[17] Unity Technologies. *MonoBehaviour.LateUpdate()*. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html`, 2021, April 19.

[18] Unity Technologies. *MonoBehaviour.Start()*. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.htmll`, 2021, April 19.

[19] Unity Technologies. *MonoBehaviour.StartCoroutine*. `https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html`, 2021, April 19.

[20] Unity Technologies. *Plug-ins*. `https://docs.unity3d.com/Manual/Plugins.html`, 2021, April 25.

[21] Unity. *TextMeshPro*. `https://docs.unity3d.com/Manual/com.unity.textmeshpro.html`, 2021, April 25.

[22] LeanTween. *LeanTween documentation*. `http://dentedpixel.com/LeanTweenDocumentation/classes/LeanTween.html`, n. d. Retrieved 23.04.2021.

[23] Kagan Nohut, B. *Procedural and Parametric 3D Modelling of Road Tunnels*. University of Stavanger, 2020.

[24] Bouvet. Digital tvilling av ryfast-tunnelen i vr for full-skala tunnelsimulering. `https://www.bouvet.no/prosjekter/ digital-tvilling-av-ryfast-tunnelen-i-vr-for-fullskala-tunnelsimulering`, n. d. Retrieved 2021, May 11.

[25] SINTEF. *Vi bør etablere redningsrom i tunneler.* `https://www.sintef.no/ siste-nytt/2020/-vi-bor-etablere-redningsrom-i-tunneler/`, n. d. Retrieved 10.05.2021.

[26] FBX. In *Wikipedia* `https://en.wikipedia.org/w/index.php?title=FBX&oldid= 986594903`, 2021, May 13.

[27] Aakre, A., Appel, K., Kronborg, P., Wendelboe, J. T. & Yde, P. *ITS terminology.* `https://nvfnorden.org/wp-content/uploads/private/ 2012-ITS-Terminology.pdf`, 2012.

[28] Unity Technologies. *Time.timeScale.* `https://docs.unity3d.com/ ScriptReference/Time-timeScale.html`, 2021, May 09.

[29] Reichelt, F. Fire auto - car on fire [sound clip]. `https://freesound.org/people/ florianreichelt/sounds/563765/`, 2021, March 17th.

[30] Mountain Goat Software. *User Stories.* `https://www.mountaingoatsoftware.com/ agile/user-stories`, n. d. Retrieved 7.05.2021.

[31] Bozgeyikli, E., Raij, A., Katkoori, S. & Dubey, R. Point teleport locomotion technique for virtual reality. *CHI PLAY* (2016), 2016, October. https://doi.org/10.1145/2967934.2968105.