BACHELOR THESIS

Study programme / specialisation:
Datateknologi - bachelor

The spring semester, 2022

Open / ~~Confidential~~

Author:
Benjamin Mydland, Ove Oftedal

Benjamin Mydland, Ove Oftedal
(signature author)

Course coordinator: Tom Ryen

Supervisor(s): Naeem Khademi

Thesis title: *CoolEngine: Simulating Realistic Fire and Smoke in a Unity3D-based Tunnel Fire Rescue Game*

Credits (ECTS): 20

Keywords:
CFD
FDS
Fire Simulation
Unity3D
Tunnel safety

Pages: 99

+ appendix: 10

Stavanger, 14.05.2022
date/year

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

Benjamin Mydland
Ove Oftedal

May 14, 2022

Abstract

Simulating fires in a 3D-environment is nothing new; But the tools used for simulating fires in such scenarios are extremely limited, as they are seldom the focal point of the scenario and they are extremely resource intensive and complex to simulate. But it is certainly possible, and desirable, to have well functioning and realistic fire simulation. The goal of this project is to improve an already existing fire simulation game into a more general and useful software for simulating tunnel fires, by first and foremost improving the fire simulation.

With relatively recent improvements to CFDs used to simulate fire, and improvements to computational power of consumer-grade PCs, it should be much more feasible to attempt a realistic scenario than before. This will be achieved by extending a simpler fire simulation game in Unity with the results from the CFD known as FDS.

With some merging of results from FDS and with a focus on generally improving the user experience, a software which is a step in the right direction for the desired product has been achieved.

To summarize: It is possible to further improve a software for simulating a fire scenario in a tunnel, and there is a great deal of potential in this field. Doing this with a CFD, like FDS, is fruitful; but it also has its own set of unique challenges and limitations.

Acknowledgement

We would first and foremost like to thank our advisor Assoc. Prof. Naeem Khademi (University of Stavanger), whose advice always pushed us and our work farther than we could have imagined.

In addition, we would like to thank developers Djurre Siccama and Dag Magne Ulvang (both GEXCON) for telling us of their previous experiences with merging a CFD (FLACS in their case) and Unity, giving us a valuable baseline for us to understand the limits of the field.

We would also like to thank Assoc. Prof II Henrik Bjelland (University of Stavanger) for informing us of some important aspects of FDS and Pyrosim.

Finally: we would like to thank our families for their continuous support, without which nothing would be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

There are a multitude of reasons for this thesis' existence.

One of them is just how dangerous a tunnel fire can be. Fires are in general extremely dangerous, as any person knows well. But it becomes a whole new level of dangerous when it is in a tunnel scenario. Oxygen becomes even more scarce, the open space one could escape to is no longer available, and traffic and structural collapse may clutter a rescue effort. This can be demonstrated via a concrete example: There was a serious and severely dangerous fire in the Bragernes tunnel in 1999, it led to an explosion which left two people dead [1]. This demonstrates the lethal potential of fire, even in tiny Norway.

The point above is exacerbated when one considers how many tunnels Norway has. Finding concrete data for this is difficult, but Oddvar Karmo from Statens Vegvesen stated that Norway is somewhat of a superpower when it comes to tunnels [2]. If one thinks of it as a value like number of tunnels per capita, the number may be extremely high, provided the geography of Norway with a great deal of mountains and fjords in the way of potential regular roads. Which means tunnels become a necessity when creating road infrastructure across the country.

If the goal then is to further understand and prevent tunnel fires in a country which has many, one approach may be to simulate the fire and train people in it. This is a known approach, utilized by SINTEF as an example [3].

They created a sophisticated fire scenario in a tunnel, and placed people in it to measure how an average person may react. Even with this sophisticated simulation, the majority of attendants failed to immerse themselves in the simulation. When a subjective analysis took place at the end of the test, the mode of the attendants felt completely safe in the simulation (10/10), unlike what they would have felt in a real scenario. This *may* indicate that there is still room for improvement in regards to such simulations.

The last significant reason is that fire is generally a chaotic phenomena, which may not be understood all that well. Therefore: any attempt to further the understanding of how fire operates, and especially expose this understanding to casual observers, will have its merits.

1.2 Research Questions

The stated task and the motivations behind this project leads to several research questions:

RQ1. Are there any CFDs suitable for accessible fire simulation?

RQ2. Is it possible to utilize the results from a CFD with an environment like Unity?

RQ3. How far can the realism for fire scenarios be taken with current technology and hardware?

1.3 Objectives

The research questions above manifest themselves as concrete objectives

The main objective is to improve the realism in regards to the fire and the smoke in the already existing tunnel scenario, other important goals are:

- Research the state of the art in CFDs with fire simulation
- Improve the user experience with general fixes and additions

- Improve the interaction between the tunnel generation and how this is used in Unity
- Make interacting with fire simulation easy and accessible
- Make the game ready for VR, for realism in simulation

1.4 Thesis' Contributions

This thesis contributes to the field of fire simulation software in these key regards:

- Collects information about CFDs in the field and compares them
- Proposes a way to utilize simulation data in a Unity environment
- Explores what limits one approaches when using said approach

1.5 Thesis Structure

The chapters of the thesis have the following goals:

Introduction

The introduction introduces the main ideas, goals, questions and motivation for the thesis.

Background

The Background focuses on some of the fundamentals used in the project. Like programming languages used, software used and file formats used.

Related Literature and Software

This chapter focuses on related literature in the fields related to this thesis, and what can be learned from them. It also features a part discussing and comparing the various CFDs suitable for this project.

CoolEngine Software Architecture

As this software is another iteration in a long list of designs, it is prudent to discuss where this software is coming from and where it intends to go in terms of design. This happens in this chapter.

Methodology

This chapter will justify some of the fundamental decisions regarding methods made throughout the project; And look at some consequences of these.

Implementation

This chapter will look at some of the finer aspects of the implementation in more detail.

Evaluation

This chapter will look at some of the results of the project and the discovered limitations of the implementation.

Conclusion

This chapter will briefly summarize the results of the project, and discuss the way forward for *CoolEngine* .

Chapter 2

Background

2.1 Software

2.1.1 CFD

CFD Stands for Computional Fluid Dynamics. It is defined by Simscale, a company with their own CFD, as: "Computational Fluid Dynamics (CFD) is the process of mathematically modeling a physical phenomenon involving fluid flow and solving it numerically using the computational prowess." [4]

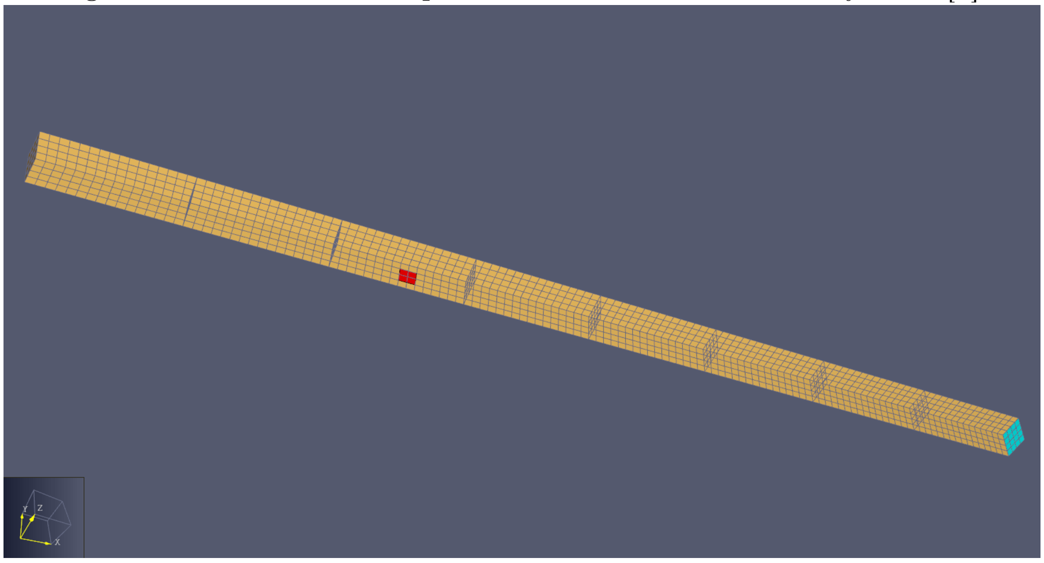
It is, in other words, a way to model diverse physical phenomena. This type of software is widely used in several fields of engineering, to model whatever needs to be modeled. It is of no use to list of some examples here, as many examples will be looked at later.

The chief example for this thesis, however, is the simulation of fire. Fire is, and has been for many centuries, a source of chaos. Thus: the ability to simulate, and predict, fire-behavior has massive potential for different applications. As the field of simulating fires itself seems ripe, it seems prudent to further explore the possibilities of having it used in settings closer to people who are not related to engineering fields; like people who play video games, or people with a surface level interest in fire and tunnels.

Ways of Achieving Fluid Dynamics

There are multiple ways a software can approach such a task of simulating. The most relevant one in regards to CFDs and this thesis is splitting the domain/mesh into smaller cells and computing the temperature, as an example, for each cell.

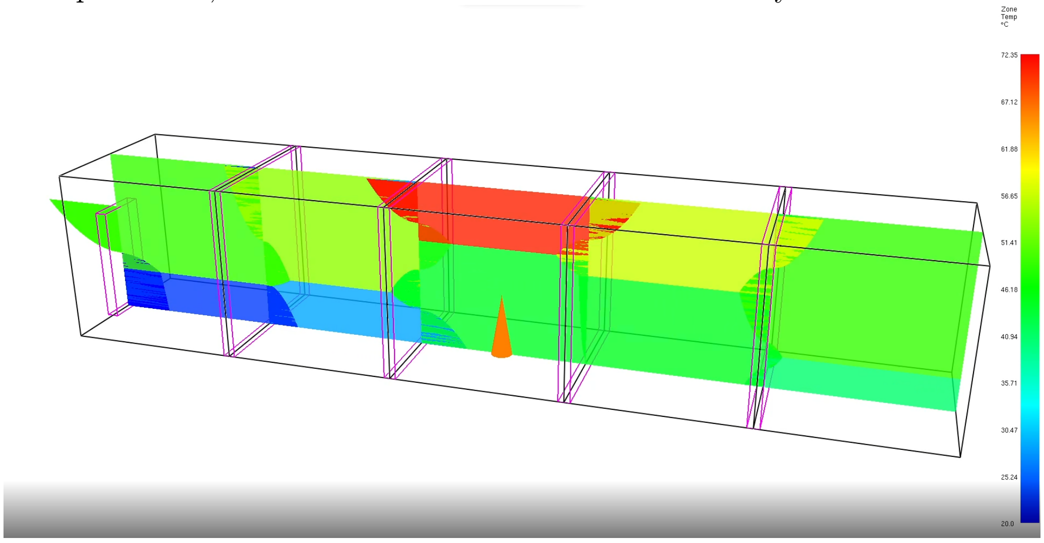
Figure 2.1: A mesh model produced with the FDS GUI Pyrosim [5].



This method is used by CFDs like FLACS and FDS.

Another relevant way for this thesis is the so-called two-zone method. This method, instead of working with a great deal of cells, splits a room into two zones: One upper, and one lower. This method uses the fact that heat and smoke will rise to make a upper layer dedicated to that, and this makes computations much quicker and simpler. It is slightly unclear if this is even to be considered a "true CFD", but it has the same purpose for this thesis.

Figure 2.2: A custom model from CFAST[6] viewed in Smokeview[7]. It has multiple rooms, and the data is therefore somewhat noisy.



This method is used by software like B-RISK and CFAST.

Which of these methods is preferred will be discussed later.

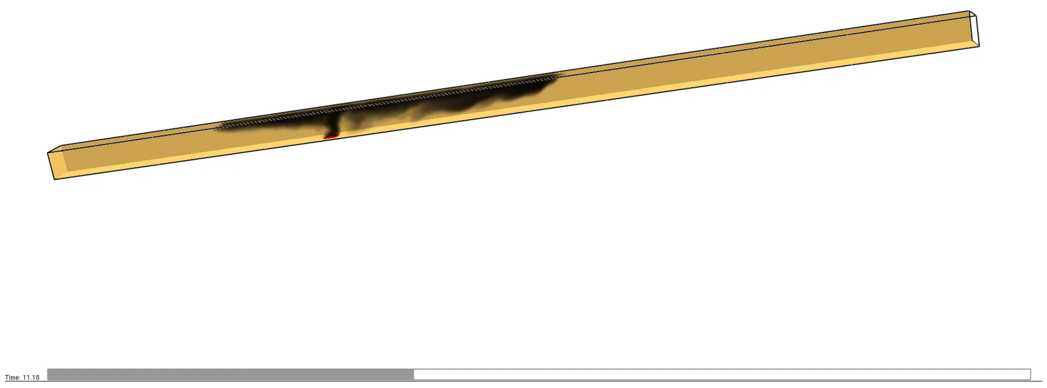
2.1.2 Smokeview

Smokeview is a software used to visualize the results of CFDs that focus on fire models [8]. The software is developed by NIST, and is mainly developed for use with their own fire models; although some other models also use Smokeview.

Most CFDs have equivalents to Smokeview, as it is extremely practical to have the results be visualized immediately upon completion of calculation.

In addition: its focus on fire specifically will illustrate well to receive some preliminary results on how the fire looks before it is implemented in the software, making it extra useful for any work in this field.

Figure 2.3: Example Smokeview[7] output from a tunnel-like FDS simulation



2.2 File Formats

2.2.1 Plot3D

Plot3D was originally a software developed by NASA to visualize solutions computed by scientists working on diverse fluid-dynamics problems, in their case: airflow of spacecrafts [9].

It is today used as a common data format for CFDs to output their solutions, outputting .q or .xyz files as possible examples.

2.3 Programming Languages

2.3.1 Fortran

Fortran is a programming language focusing on high-performance, usually used for science and engineering projects [10].

It is used by some CFDs, like FDS, and will therefore hold relevance in this thesis.

2.4 Development Philosophy

2.4.1 Test Driven Development

Test Driven Development is a style of programming which encourages relying heavily on testing to create quality code. [11]

Specifically: It encourages users to create tests before they finish the code they want tested. The point is to make it clear what exactly the code is to do before it is created, to further guide the programmer as to how to create the code.

This approach will be useful for the project, as this project's focus on both CFDs and Unity at the same time makes it exceedingly complicated, which necessitates some testing to enable control.

Chapter 3

Related Literature

3.1 Earlier Work at the UiS

3.1.1 3D-Self Rescue Game

In the spring of 2021, there was developed a game as a part of a bachelor thesis at the University of Stavanger. The main goal of the game is to simulate fire scenarioes in tunnels and provide the player an idea of what to expect in these situations [12].

This game was built on the top of the procedural tunnel modeling already developed, and features many features essential to a fire simulation scenario. Some of the features are:

- Flame and smoke that will reduce the player's health
- Stations along the tunnel which have fire extinguishers in them
- Ability to extinguish the fire with the extinguisher
- Emergency exits placed along the tunnel
- Ability to win the game by exiting through the fire exit

The game works well as a video game on its own, it has a great deal of functionality one would usually find in a video game, but the realism of the fire and smoke itself is limited as it gamifies the concepts excessively.

This gamification shows itself clearly when the player takes "damage" when touching smoke, a simple abstraction for the risk of death real smoke carries.

The main point of this thesis is to rework the fire specifically used in this earlier thesis, but some of the other points may be reworked, which will be discussed later.

3.1.2 Digital Tunnel Twin

At the same time as the aforementioned video game was developed, there was developed a master thesis based upon improving the capabilities of the Digital Twin model which the game also uses [13].

As this project was also based in Unity, there is some overlap between the two projects. Both projects took measures to take traffic into account, but only the master thesis seemingly got it to work within their time frame. Thus part of this work will be merging some of the better aspects of these two theses to make a unified product.

As the master thesis is more based upon a grand and complex design with databases and such, it was chosen to create the code for this thesis based upon the bachelor thesis; and then later improving some parts of it with inspiration from the master. It is generally better to start simpler and make it more complex when it is needed, than to start complex. This adheres to the principle of KISS [14], which is an important principle when continuing already massive and complex work like done in this thesis.

3.2 Earlier Work in General

3.2.1 Brandforsk

The University of Lund in Sweden has dabbled in importing logic from CFDs into Unity before [15]. Their approach was to use a computationally simple model based upon a two-zone model, CFAST in this case, and use physical formulas based upon physical phenomena like light scattering to improve the visualization.

An important aspect of this work is how they state that more complex models than those based on two-zones would not enable interactivity as the computations are too heavy. This will be discussed later when the choice of CFD is made.

3.2.2 Gexcon

Gexcon is a business which has developed the CFD FLACS, which will be discussed later, used for simulating explosions and similar phenomena on oil rigs. A portion of their team has developed a plugin between FLACS and Unity, showcased in their Youtube video: *FLACS VR-Safety* [16].

This work differs from the work by the University of Lund in an essential way: it does not seemingly use a two-zone model, due to the nature of FLACS. This makes the fire showcased in the video somewhat limited in interactivity and visualization, but it allows the consequences of the fire to be showcased more clearly.The video clearly demonstrates the radiation the player endures changing based on how close they are to the fire, and it demonstrates a lethality percentage for the player being calculated.

3.2.3 Compare Two-Zone and Pure CFD

Wen Jiann Bong from the University of Canterbury wrote a thesis in 2011 comparing the viability of more CFD-centric software to software relying on the zone-models [17]. The author notes that it is difficult to compare and evaluate them accurately, but also states that the zone-models should be avoided if the fire is small compared to the enclosure. Given that this thesis works within a tunnel-scenario, where the fire is a relatively small part of the tunnel itself: it may be slightly prefered to have a CFD-based software compute the simulation if accuracy if the chief concern.

3.3 Commonly Used Fire Simulation Software

There has been a great deal of research into simulating fire, and there is consequently a great deal of software that can simulate fires. Therefore: a non-trivial amount of work had to be put into researching the different software, to make sure the right one was chosen.

3.3.1 Summary of Viability

Table 3.1: Overall viability, in descending order of viability

CFD	Pros	Cons	Correct use case	Two-Zone
CFAST	Fast	Reduced accuracy	Yes	Yes
FDS	High accuracy	Resource intensive	Yes	No
Smartfire	Academic	Old	Yes	No
OpenFoam		Complex	Yes	No
FLACS		Complex	Yes	No
B-RISK		Probability-focus	Yes, but smoke is lacking	Yes
Simscale	Cloud-based	Cloud-based	Not in this case	No
Kameleon FireEx			Yes	No
BehavePlus			Fire, No, no smoke	No
AVL Fire			No, focuses on combustion	No
COMSOL			No, focuses on heat transfer	No

The table above is roughly sorted with regards to how fitting the CFDs are estimated to be for the project. CFAST being among the most viable, and COMSOL being among the least viable. There is also a double line in the table indicating the divide between viable and unviable for the current project.

3.3.2 Ease of Use

Along with this general summary, there are other factors which should be taken into account. This includes not just how accurate or how satisfactory the performance is for the models are, but other factors that decide how easy it is to work with.

Table 3.2: Ease of use parameters

CFD	Documentation	Open Source	License	API
CFAST	Free	Yes	Free	No
FDS	Free	Yes	Free	No
Smartfire	Unknown	No	Yes, physical	No
OpenFoam	Expensive courses	Yes	Free	Yes
B-RISK	Free	No	Free	No
FLACS	Courses	No	Yes, physical	Yes, Python
Simscale	Free	No	Yes, digital	Yes

The CFDs that are completely unviable have been left out of the table.

3.3.3 Advanced Geometry

The models also vary in how they model the rooms/spaces themselves, and this will be of importance later. They mainly differ in if they use a box-model, or if they use more flexible models; often CAD.

Table 3.3: Advanced geometry

CFD	Curve support
CFAST	No, box-model
FDS	No, box-model
Smartfire	Yes, CAD
OpenFoam	Yes, CAD
B-RISK	No, box-model
FLACS	Yes, CAD
Simscale	Yes, CAD

Note that the assertion of a yes/no here is not absolute, some softwares like FDS have third-party support which allows pretty complex geometry through various methods. This will be discussed later.

3.3.4 CFAST

CFAST is a CFD developed by NIST. It mainly utilizes the two model zone, but has support for a single-zone model for larger domains. It is in active development, and is open source on GitHub [18]. The computational power needed to support this simulation is not excessive, but as a direct result it is may not be as accurate as other simulation software. CFAST has existed since 1990, and is therefore noticeably feature robust. [19]

3.3.5 FDS

Fire Dynamics Simulator is a CFD meant for simulating fire and different fluids pertaining to fire. [20] Like CFAST, it is developed by NIST and is also open source. It is a great deal more resource intensive than CFAST, but has more granular data as previously mentioned.

It also has a great deal of third party support, with software like Pyrosim giving it a complete GUI [5].

3.3.6 OpenFoam

OpenFoam is a free and open source CFD-software [21]. It has a specific module, FireFoam, which tackles pyrolysis [22].
This is a software which seemingly has a great deal of potential for the given task. As it seems it is one of the better and more widely used CFDs. But it is fairly inacessible as it is more geared towards industrial use, and therefore incentivises its users to take expensive courses to grasp its concepts.

3.3.7 FLACS

FLACS is a CFD developed by Gexcon. It covers many fields, with fire being one of the main ones. [23].
This also seems like a robust way to model fire. But it is marginally out of scope considering the project, as it seemingly has its own way to render entire enviroments. While this project just requires fire that interacts with structures. It also requires a great deal of training, like OpenFoam.

3.3.8 Kameleon FireEx

Kameleon FireEx is a CFD managed by DNV. It has a main focus on simulating fires and disasters on oil rigs, based upon the focus of the website [24]. It seemingly creates exceedingly realistic fires and smoke. But is hindered by not being readily available for consumers, again taking the industrial focus. It seems they do not even offer user-licenses for individuals.

3.3.9 Smartfire

Smartfire is a fire simulation software developed by the FSEG of The University of Greenwich in England [25]. It is a software developed over many years, and has been acknowledged by many as being robust. [26]
This seems to be a solid simulation software in its own right, and has the major advantage of being created in an academic setting; which fits well into the the given task. But it has a major problem: it is noticeably old, and its creators may be done updating it. This seems to be the case since, if all the instructions are followed for downloading as they recommend new users, the only response the user receives is a "license expired" error.

3.3.10 Simscale

Simscale is one of the most unique CFDs here. It is developed by Simscale, and has the unique property of being cloud-based [27].

It seems this also is a solid CFD, seemingly with capabilities for fire simulation. It also boasts rich API possibilites.

The fact that it is cloud based is simultaneously its greatest boon and its greatest curse. On the flipside: it allows for outsourcing all the heavy computations to a powerful cloud, which would make heavy computations easier. On the downside: not having the calculations done locally may make the software less dynamic, as it depends on queries to a server for changes, instead of just calculating them themselves. It was for this project decided to have the calculations done locally for simplicty's sake, but this CFD is worth keeping an eye on.

3.3.11 B-RISK

B-RISK is a fire simulation software developed by BRANZ which focuses on the probabilities of fire [28].
B-RISK is seemingly pretty similar to CFAST, but with more of a focus on risks. Which for this task, which focuses more on the effects and physics of it, will be unfitting.

3.3.12 COMSOL

COMSOL is a CFD which can simulate several factors; including airflow, electricity, and chemical properties. It is therefore quite robust in terms of physics.
Unfortunately, it is limited when it comes to fire. As it mainly focuses on simulating heat-transfer, and heat flow [29]. This limits its usefullness in this task.

3.3.13 AVL Fire

AVL Fire is a CFD that focuses primarily on combustion engines developed by AVL. [30].
This CFD is presumably competent in what it does. But as it mainly focuses on combustion and ignition and such topics, it is unfit for this task.

3.3.14 BehavePlus

BehavePlus is a fire modelling system mainly focused on modelling possible forest-fires, developed under the supervision of the USDA forest service and kept in a United States government website (.gov) [31].
This seems to be a serious and to the point fire modeller, but it unfortunately is too specific to fire and forest-fires that it is unusable for the task of simulating general fires and smoke.

Chapter 4

CoolEngine Software Architecture

4.1 Future Design

As there have been a few different directions this project, when looked upon over multiple years; there will be an attempt to further providing ideas to how the software can be developed in the future, regardless of how far this project manages to do this.

An essential component which will have to be added sooner or later is VR-compatibility. The ability to come even closer to the simulations and the results produced will let the user immerse themselves even more in the scenario, and may boost the current software massively for purposes like training of personnel to fire scenarios.

Another component which would be in line with the vision would be the addition of multiplayer. As it would massively help the potential users who may want to use this software to train teams of people in cooperation in these sorts of scenarios as an example. This also helps the current design since the simulations of *CoolEngine* will only become better and better the more computational power one has access to; If the main simulation could be outsourced to a super-computer which broadcasts the results to a team of fire-fighters, as an example, that would really boost the usefulness.

Another aspect that should be looked at more closely for a more complete piece of software is the traffic components. The traffic components right now are pretty limited, as the user can only watch as cars drive by. But in a more realistic scenario, it would be the user who drives in on a burning scenario, as people rarely walk around in tunnels like the current architecture. So the user should be able to drive cars, but this would take a great deal of effort.

Another interesting extension that may be a part of this complete software would be extending this to general structures. Like if the same software that simulates the tunnels could simulate the local church or a cinema. This would rely on more initiatives that map real life structures to data-models however, and would also have a larger potential to be exploited by bad-faith actors like arsonists.

The last extension worth mentioning would be to aim it towards a specific branch or work-group. An example would be implementing tools that the fireman needs to respond to a fire in a tunnel, or creating an interface that would enable tunnel-architects to more clearly see how small changes in their design may effect the security in such a scenario.

To summarize all the points above: The software which can be built upon the current iteration of *CoolEngine* is envisioned to be a flexible program which allows for a lot of fire scenarios, a lot of models that interact with the fire, a lot of actions for the user can take; all wrapped up in an immersive simulation.

4.2 Current Architecture

The current design of the game has taken an approach to become more of a general software instead of a game, comparable to taking some parts of a game and some parts of a simulation and meshing it together into a *software*.

In the center of the current architecture is the *CoolEngine* - pipeline. This pipeline aims to be the groundwork for more realistic fire simulations, giving the user and the program all of the essentials one would need to further the realism of combustion. The pipeline ensures that all the data that a compatible CFD may produce can be inserted into Unity and used for the software.

An additional essential part of the current design is the freedom to experiment with several different things. The previous game technically supported multiple tunnels imported from the NVDB, but it did not have code which scaled well with several different scenarios. This architecture, however, focuses on letting the user choose myriad of things:

- Parameters for the simulation
- Simple traffic and some levels of traffic
- Support for many tunnels
- Ability to choose where to spawn in a tunnel
- The user can explore more of the tunnel

4.3 Software Foundation

4.3.1 Generation of tunnels

The fundamental basis for this work is the generation of the tunnels from the NVDB database into .fbx files. This is an essential part of all further projects, but does not carry with it any design by itself as it is mostly a tool

4.3.2 Tunnel rescue game

The main inspiration and progenitor to this project is the game that came before it [12]. In that thesis, the aim was to create a game to help teach people self-rescue. This game therefore focused a great deal on the aspects which makes it a *game*: It uses game logic like damage-points to hurt the player upon exposure to fire and smoke, as an example.

It also creates scenarios which are not particularly realistic, to focus on the drama often found in video games. This is showcased when there are generated buses standing diagonally in the tunnel to block the player, and how the fire extinguisher without fail extinguishes the fire regardless of how intense the fire may be.

When the time came to further the concept in this project, part of the goal was to create a design which is as realistic as possible. This means removing some unrealistic components, like the aforementioned over-powered fire extinguishers, if they could not be ensured to be based on some sort of reality.

4.4 Potential Userbase

Simulations can be exceedingly difficult for an average consumer to wrap ones head around. When the goal is to predict how phenomena, like fire, will behave: the bar for entry for understanding a simulation software or method is extremely high, and this just is not feasible for most people.

The primary group this software is intended to appeal to is therefore these people, the people who are unfamiliar with fire but want or need to interact with it. Examples would be an ambulance worker who may have to save people from a fire in extraordinary circumstances, or the aforementioned architect who has not taken a day of fire-theory in their life.

This would be the most powerful niche to target. As the people who have fire as their main field of study, like firemen, probably have access to sophisticated tools that a software of this size simply can not compete with.

But another niche this software may access is the institutions or people who do not have a sizable budget, but still a clear need for this sort of software. Some examples could be fire-stations in less developed or central parts of the world, which must prioritize the actual function of their tools instead of some software for simulation. This is due to the largely open-source design of this software, which would be readily available to a wider audience.

The last, but not as niche group of potential users, is the crème de la crème: the people with budget and interest in the field. Like firemen working in a large city. This may be a difficult group to please as they probably are well-trained and have a great deal of resources, but it is also one of the most natural groups to target if the software gets several of improvements.

Chapter 5

Methodology

5.1 Potential Approaches to Fire Simulation and Selected Method

When the task is to improve how realistic a fire scenario is, there is an abundance of options one could pursue. Fire propagation is a well-researched field, with several solutions.

One could look into the more physical and mathematical side of it, and model it based on those disciplines, not unlike what Brandforsk did. But in this scenario, which places emphasis on the computational part of it, this is less relevant and useful.

One could also look into the software that attempts to simulate crowd movement in fire scenarios, like PathFinder by ThunderHead Engineering [32]. However, this is most usefeul when one wants to explore how masses escape a dangerous situation, and this simulation only concerns the user/player.

Thus: in order to both satisfy the need for a computer solution, and to focus on the fire/smoke itself, CFDs is the natural solution. As it focuses on the physics behind the phenomena, without requiring much focus on the formulas.

5.2 Choice of FDS

When it comes to the choice of CFD, this thesis will be implementing the simulation with the help of FDS. There are many reasons as to why the choice fell on this particular CFD, and they were discussed in the chapter regarding related literature. The reasons can be summarized as:

- It is easy to aquire a license as a student
- It specializes in fire, and is exceedingly realistic; meaning it is better suited for our task
- It does not require too much time to learn

A sizeable proportion of the other CFDs also fit some of these criteria, but they all fail in at least one of them. CFAST for instance is more than adequate; but it uses the two-zone model previously discussed, meaning it is unrealistic in comparison to FDS.

Smartfire also seems to be well suited, but is extremely difficult/expensive for a student to acquire a license.

OpenFoam is also an excellent candidate, and may be preferred if one is to work on a simulation like this over a longer period of time with a great deal of time and resources, but it also requires a great deal of training which does not fit into the limited timeframe of a thesis like this.

Therefore: FDS was deemed to be the most fitting in the most important aspects. Though it is worth noting that any CFD conforming to the requirements for file formats should be able to use the framework developed, if they follow some criteria which will be established later.

5.3 Choice of Coupling Between Unity and FDS

Like previously discussed with the Brandforsk report [15], there are two main routes to go if one wants to use CFDs to simulate fire:

1. Implement some physical and mathematical concepts from the CFDs directly into the game, like the report does.

2. Couple the game and the CFD together: The CFD computes the data, and the game visualizes and utilizes it.

It was chosen for this project that the latter of the two be attempted: coupling. There are positive and negative aspects to both solutions, and the other option may be the one that offers the "best" results; but as the scope of this project is both limited and computational, the direct coupling is preferred.

5.4 From Plot3D Output to Unity

There are several ways one can access the information in the FDS simulation as an output. The one chosen for this project was reading the data through the Plot3D format which FDS supports.

```
0.74000E+02,-0.20000E+01, 0.00000E+00, 0.10207E+01
0.76000E+02,-0.20000E+01, 0.00000E+00, 0.10231E+01
0.78000E+02,-0.20000E+01, 0.00000E+00, 0.10162E+01
```

Listing 5.1: A snippet from one of the Plot3D files in .csv format.
In this case, it displays x, y, z coordinates and the corresponding soot density for each cell

This data will be as granular as the simulation is allowed to be, if there are more and smaller cells, the data will have finer granularity.

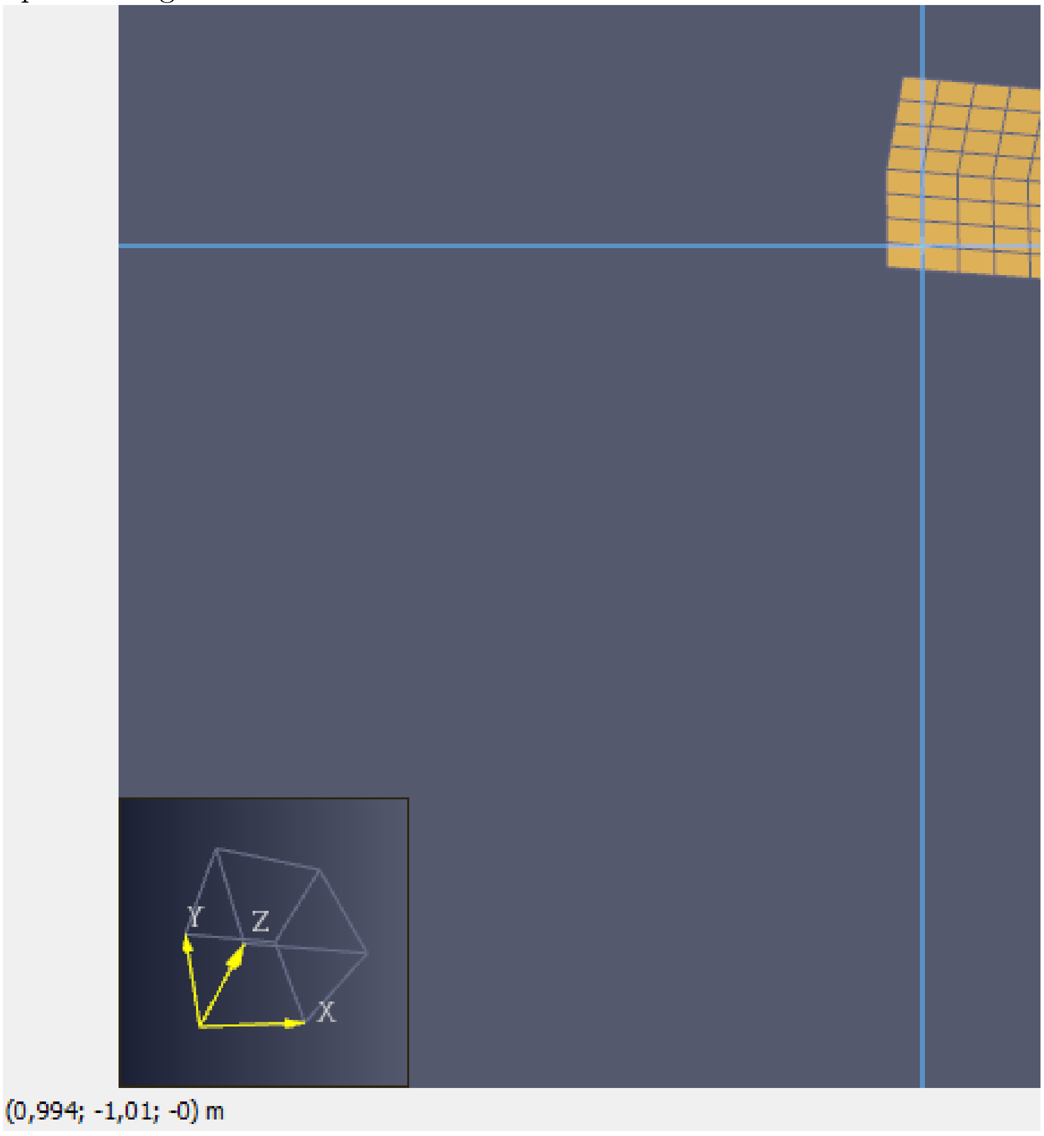
Now that the data from the simulation is acquired, the only question is how exactly it will be used and displayed in Unity. The goal at the end of the day is to have this data influence the player to offer them the impression of being in a real fire scenario, after all.

There are several possible ways the data may be used. It may be viable to create a mesh to represent the data, as this is the way the data is represented in Plot3D after all. This, however, has the drawback of being ill-suited for 3D, as meshes are best at visualizing surfaces like they usually do in video games.

For this project, a slightly naive solution was chosen. Unity has a particlesystem which can represent particles, like smoke and fire in this scenario. This has the advantage of being easy to implement and understand: the coordinates of the cell is mapped to the coordinates of the smoke-particle as an example.

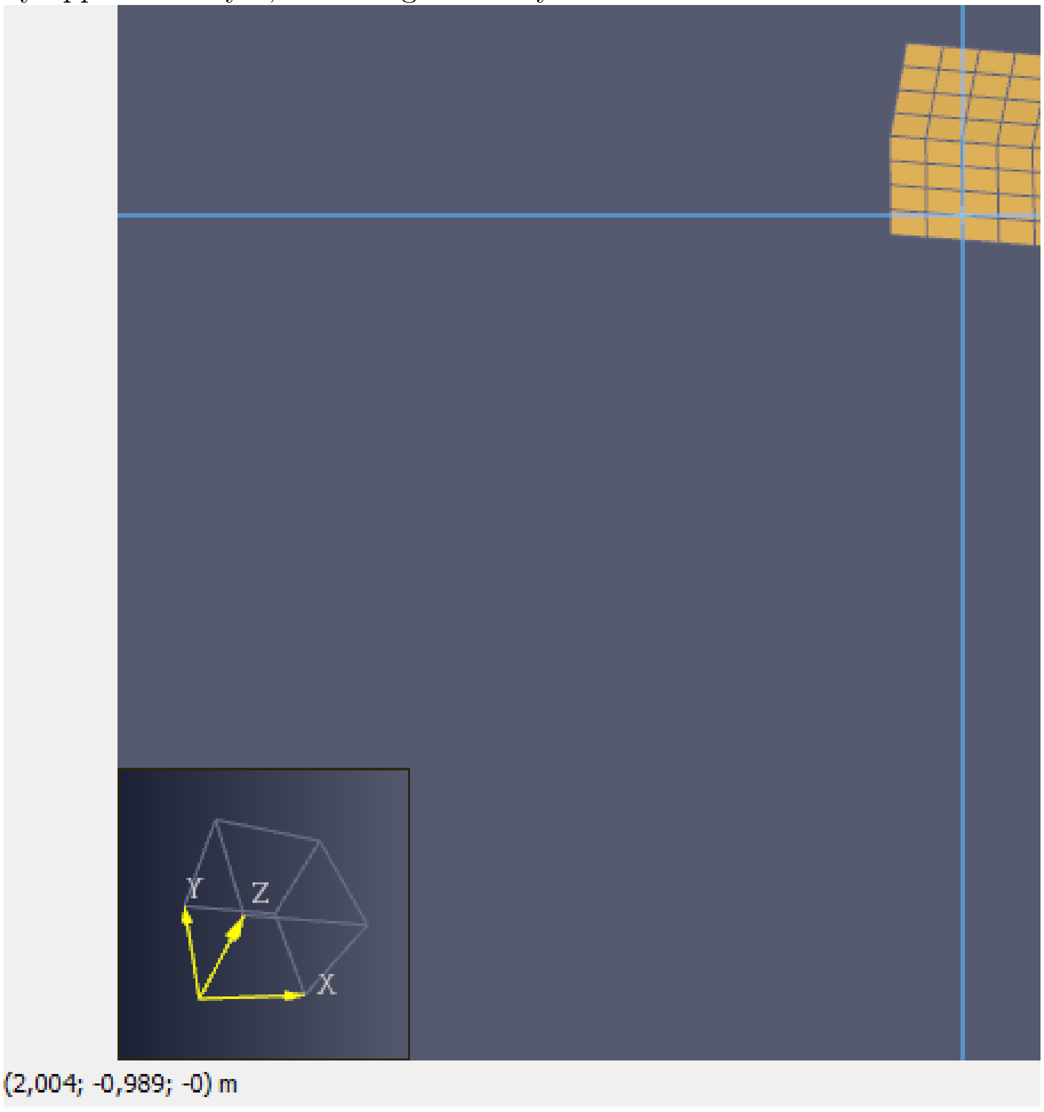
There are some drawbacks however. One of them is the offset that naturally exists because of this decision. If a particle/point should be used to approximate a cell, then it should be in the middle of the square, as is natural. The problem is, however, that the coordinates listed in the .csv file is one of the corners of the cell, not the center.

Figure 5.1: Cells of size 1x1 have their coordinates in their corner, as this cells coordinate is (1,-1,0) in theory, which rougly fits with the coordinates up in the right corner.



(0,994; -1,01; -0) m

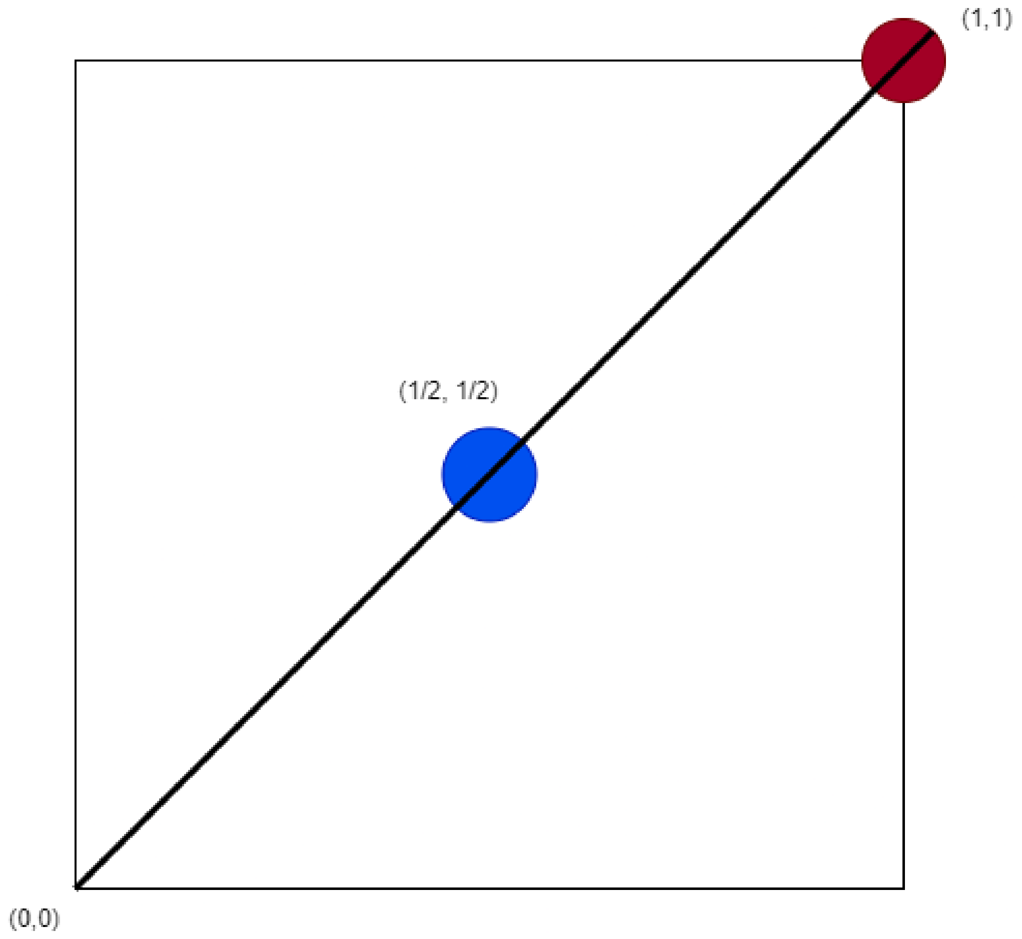
Figure 5.2: Moving to the next cell reveals the X coordinate has incremented by approximately 1, according to theory



(2,004; -0,989; -0) m

This will lead to a skewing of the entire simulation, as every particle does not map entirely with the cell.

Figure 5.3: A visualization of a 1x1 cell with the blue point demonstrates where the point should be, and the red point where its actual location is. [33]



This means the simulation will always be slightly offset with a vector of (-1/2, -1/2). This is no real problem, as it is a minor inaccuracy in the grand scheme of things, but the assumption that this is fine is worth mentioning as a fundamental aspect of the approach to this project.

5.5 Choice of Simulation Domain

The choice of domain/mesh for the simulation is also an important one. The simulation calculations required for the improvements to the visualization are extremely intensive, and the main such bottlenecks are:

- The time of the simulation
- The granularity of the data (size of cells)
- How large the domain is

All of these will be discussed in turn later. The size of the domain is the most interesting part present.

5.5.1 Domain options

1-to-1 models

The most natural choice would be to just model the domain after the actual tunnel proportions directly. But there are some problems with this approach. First: it would be a massive calculation, and would take a great deal of time and resources. This could, however, be circumvented by further reducing the domain after the main domain is generated, and just running the simulation in a smaller portion. This is a fair possibility.

Another problem, however, is how limited this sort of design is with the chosen CFD: FDS. It does not support anything more complicated than a box-model by default, like discussed before. This can however be circumvented by the selection of third-party software like the aforementioned Pyrosim [5]. Pyrosim has a feature where one can convert various 3D-models into models that work with FDS. One of these 3D-models is the .fbx format that this project uses for the tunnels. The problem with this, however; is that either the original transformation from NVDB to .fbx was not designed with such conversions in mind, or the Pyrosim software is limited. The figures below demonstrates an example of conversion.

Figure 5.4: The .fbx model viewed from above prior to conversion.

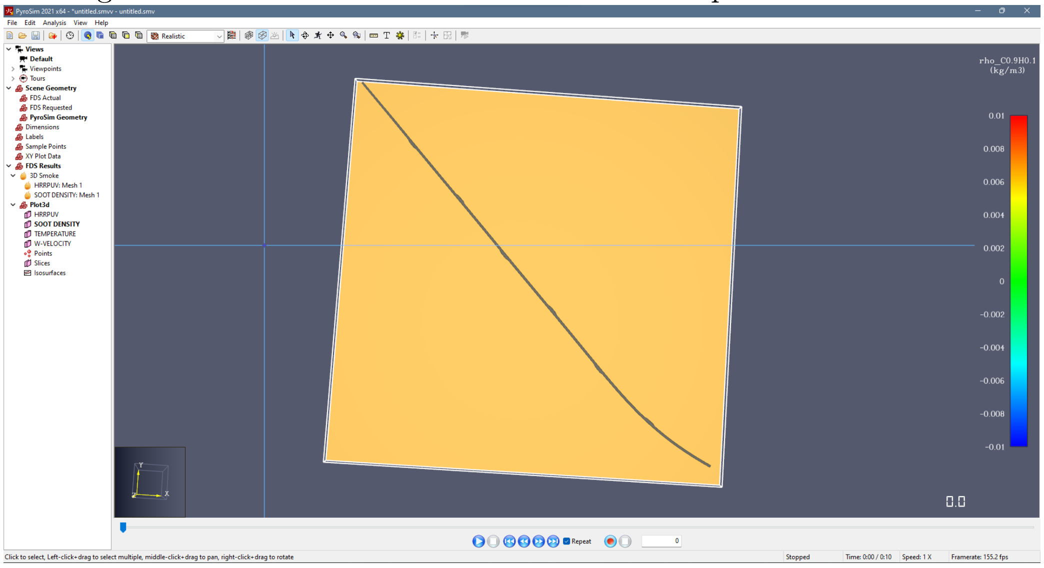
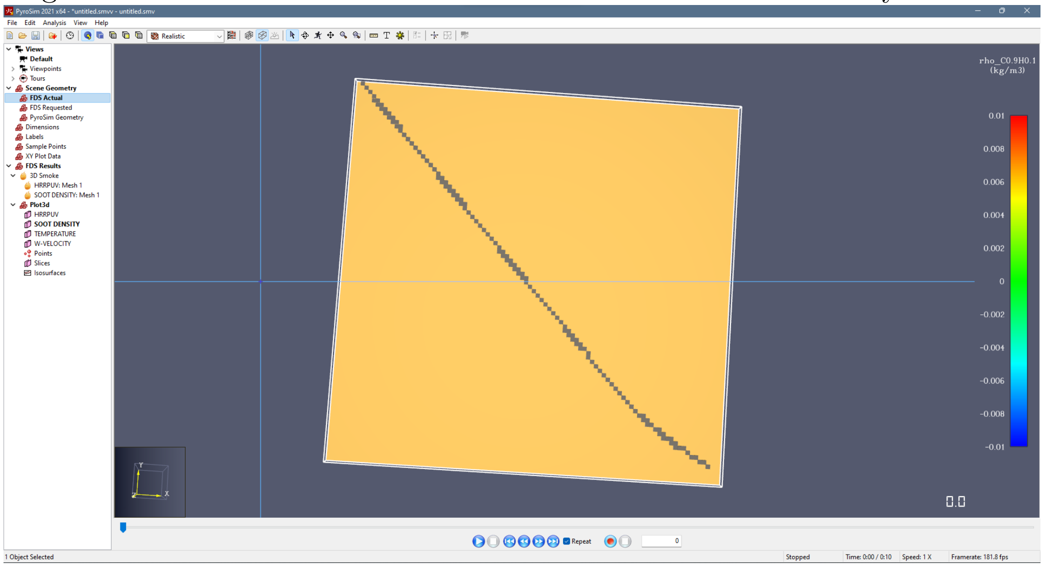


Figure 5.5: The result after conversion from .fbx to FDS-friendly domains.



As the figures illustrates, the result after conversion is a quite limited and box-based model which it is somewhat difficult to imagine smoke and heat flowing in between. The conclusion is that this is an interesting possibility, but it was not deemed viable for this project.

The chosen domain

The chosen domain, at the end of the day, is a simple and connected box model-like displayed in figure 2.1 as an example. This solves most of the problems discussed relating to the previous solution:

- It is easy to control the size, and therefore performance
- It will be easy for the smoke and temperature to flow, as it is a continuous model

It does, however, introduce new limitations one should be aware of.

The most pressing limitation is the geometry itself. It is an improvement compared to the .fbx conversion, but it is still unlike the mostly curved tunnels seen in real life.

Figure 5.6: Simulation fit inside the tunnel [33]

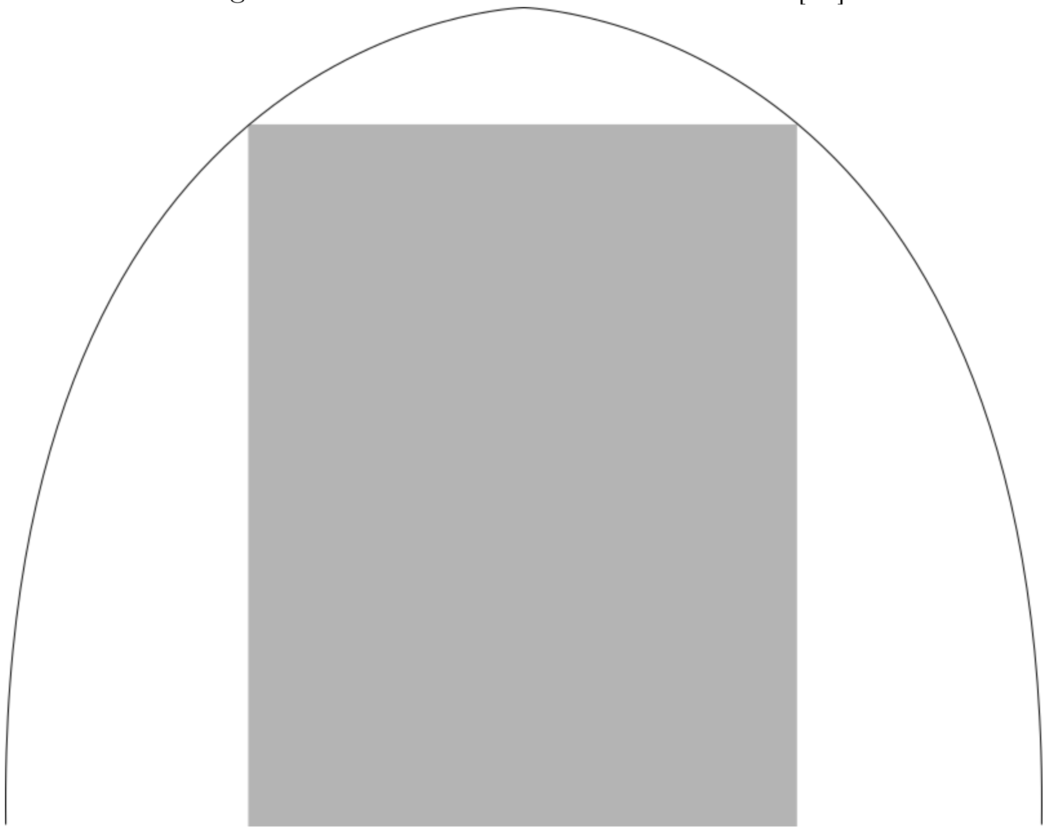
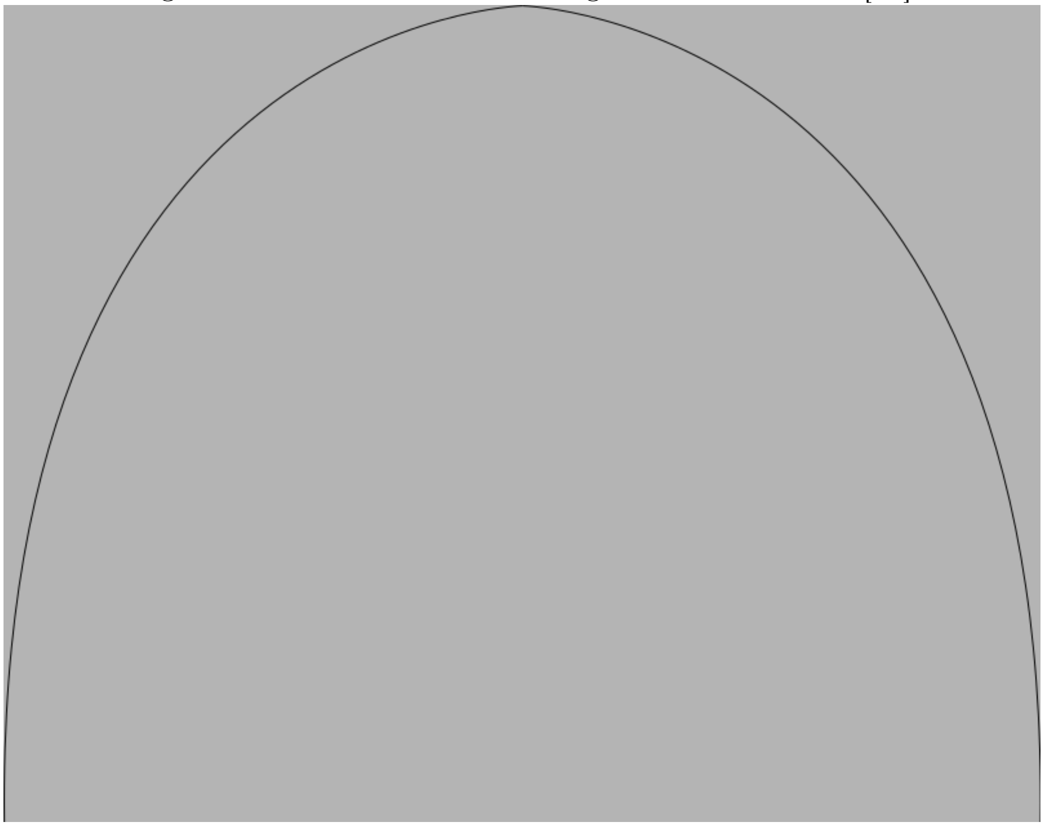


Figure 5.7: Simulation overflowing outside of tunnel. [33]



This will lead to smoke, as an example, being placed outside the tunnel in the lower illustration's case; or being placed inside the tunnel but not along the entirety of the tunnel as demonstrated in the upper illustration's case.

This is not a huge problem at the end of the day however, the majority of smoke and data would fit nicely into the square even in a real fire-scenario. But it is important to realize the limits of an implementation like this.

Chapter 6

Implementation

All the code discussed here is maintained in the GitHub-repository for the project [34]. If the reader wants access, they should contact Assoc. Prof. Naeem Khademi of the UiS regarding access to this private repository.

6.1 The *CoolEngine* - pipeline

The most essential part of this project is the pipeline leading all the way from the simulation parameters, through all the steps it needs on the way, and finally to the data in the game. It will therefore be explained carefully, and in detail.

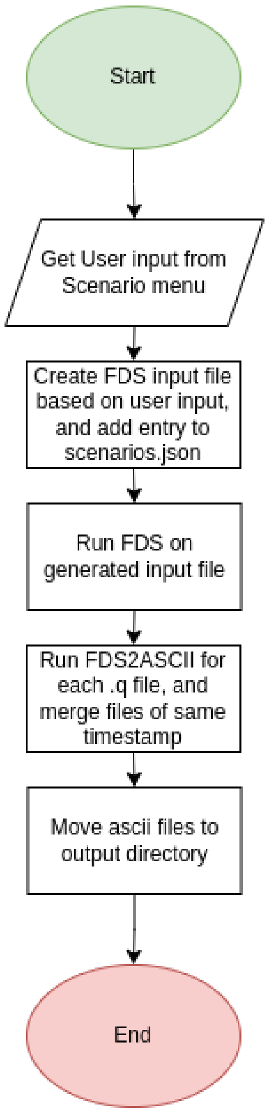
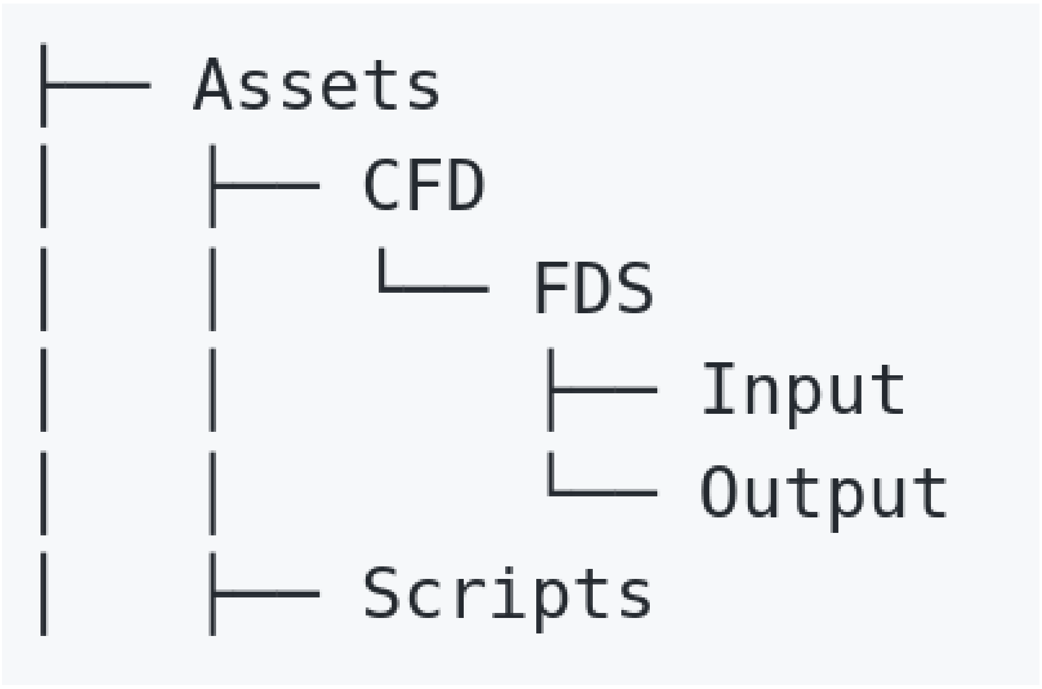

Start
Get User input from Scenario menu
Create FDS input file based on user input, and add entry to scenarios.json
Run FDS on generated input file
Run FDS2ASCII for each .q file, and merge files of same timestamp
Move ascii files to output directory
End

Figure 6.1: Overview of pipeline. Made with Diagrams.net [33]

6.1.1 File Structure

The key directories are as follows; the FDS folder contains the input folder where the simulation results are stored and converted to ASCII. The output folder contains the finished converted and aggregated data from a simulation. The Scripts directory contains all the code used in the Unity project.



```
├── Assets
│   ├── CFD
│   │   └── FDS
│   │       ├── Input
│   │       └── Output
│   ├── Scripts
```

Figure 6.2: Important directories under project/Assets

Both the *Input* and *Output* directories contain scenario directories which are named after the GUID (Global Unique IDentifier) it is assigned when created.

6.1.2 Creating FDS Input File

Everything that happens in the simulation is decided by the parameters set in the .fds file. FDS will, upon execution, run the simulation according to the initial parameters. For the purposes of this pipeline, there has been created a template.fds file which creates a tunnel-like simulation with as little overhead as possible, and allow inputs from the user of the software.

FDS can be somewhat finnicky, and it requires understanding a significant amount of moving parts to start using the software, so it will be looked at in detail.

```
&HEAD CHID='simulation', TITLE='Simulation' /
&TIME T_END=replace_time. /

&MESH IJK=16,4,4, XB=0.0,16.0,-2.0,2.0,0.0,4.0, MULT_ID='mesh' /
```

```
5  &MULT ID='mesh', DX=16., I_UPPER=7 /
6
7  &PRES TUNNEL_PRECONDITIONER=T /
8
9  &DUMP PLOT3D_QUANTITY(1)='DENSITY', DT_PL3D=1.0,
       PLOT3D_SPEC_ID='SOOT' /
10
11 &REAC FUEL='PROPANE', SOOT_YIELD=replace_soot_yield /
12
13 &RAMP ID='q', T=0, F=0 /
14 &RAMP ID='q', T=30, F=1 /
15 &RAMP ID='q', T=180, F=1 /
16 &RAMP ID='q', T=300, F=0 /
17
18 &SURF ID='fire', COLOR='RED', HRRPUA=replace_hrrpua,  RAMP_Q
       ='q' /
19
20 &VENT SURF_ID='fire', XB=63.5,64.5,-0.5,0.5,0.0,0.0, COLOR='
       RED' /
21 &VENT ID='Mesh Vent: Mesh01[0,0,0] [XMIN]', SURF_ID='OPEN',
       XB=0.0,0.0,-2.0,2.0,0.0,4.0/
22 &VENT ID='Mesh Vent: Mesh01[7,0,0] [XMAX]', SURF_ID='OPEN',
       XB=128.0,128.0,-2.0,2.0,0.0,4.0/
23
24 &TAIL /
```

Listing 6.1: template.fds

Each line has its purpose, and each component will be looked at in more detail with information based upon experience and the aforementioned FDS user guide [20].

Head and tail

```
1 &HEAD CHID='simulation', TITLE='Simulation' /
```

```
1 &TAIL /
```

Line 1 and 24 encapsulate the entire file. They essentially just tell FDS where the simulation parameters start and end. But the declaration of the CHID is somewhat important, as that is the identification of the simulation which the tool FDS2ASCII will later use.

Time

```
&TIME T_END=replace_time. /
```

The Time-line specifies information regarding time.

T_END specifies when the simulation ends, or in this case: how long the simulation runs. This is a quite important parameter, as the time to run massively impacts the runtime. It is therefore extremely important to provide the end-user access to this parameter, so that they can themselves change this to fit the limits of the hardware they are using.

Mesh

```
&MESH IJK=16,4,4, XB=0.0,16.0,-2.0,2.0,0.0,4.0, MULT_ID='mesh
    ' /
&MULT ID='mesh', DX=16., I_UPPER=7 /

&PRES TUNNEL_PRECONDITIONER=T /
```

Line 4-7 all have to do with the simulation's mesh.

The Mesh-line specifies a rectangular, three-dimensional mesh. It takes several important arguments. The IJK parameter specifies how many cells there should be in the corresponding x, y, z directions, effectively setting the granularity of the simulation.

The XB-parameter defines the lower and upper limits to the x, y, z coordinates. For this project, it was decided that the distance between the corresponding XB coordinates and the IJK values should be equal. This creates cells of the dimensions 1x1x1, essentially a unit cube. These unit cubes accomplish three things:

- Has a significant degree of detail, all things considered
- Is not excessively detailed to hamper computation
- Unit cubes makes later calculations easier

The Mult-line specifies how a mesh is to be repeated.

The DX-parameter specifies how much the next mesh should be displaced in the X direction. In this case, it places the next mesh 16 meters after the mesh before it, matching the length of the mesh so that there will not be any overlap.

The I_UPPER-parameter specifies how many times this is to be repeated. In this case, there will be 8 total meshes, because there is a base mesh defined in &MESH and it will be repeated 7 times.

The reason why this repeating approach was used instead of just creating a large continuous mesh was because any future improvements to the model in the *CoolEngine* -pipeline would need multiple meshes to emulate more complex geometry, so the pipeline is already ready for expansion in this regard.

The last line here is the Tunnel_Preconditioner line, which is set to true. This line must be applied when simulating tunnel-like scenarios with multiple meshes like here, otherwise the FDS guide warns of numerical instabilites in the simulation.

Dump

```
&DUMP PLOT3D_QUANTITY(1)='DENSITY', DT_PL3D=1.0,
    PLOT3D_SPEC_ID='SOOT' /
```

The Dump-line specifies how the output from the simulation is to be.

The parameter DT_PL3D specifies how long there should be, in seconds, between each generated Plot3D file. A whole second is chosen to improve performance, both when the files are read in Unity, and when the simulation itself is run.

The parameters PLOT3D_QUANTITY and PLOT3D_SPEC_ID both specify what sort of information should be output in the Plot3D file, in this case, the density of the soot per cell is chosen.

Reactant

```
&REAC FUEL='PROPANE', SOOT_YIELD=replace_soot_yield /
```

The Reac-line defines what sort of material will burn in the simulation.

The Fuel-parameter specifies which fuel will be the basis for the fire. It is in this project specified to be propane, as the chemical make-up of burning cars was deemed to be slightly out-of-scope for this project, and propane is a known reactant for fires.

The Soot_Yield-parameter specifies how much of the fuel should be turned into soot-smoke, measured in kg soot out of kg burned (kg/kg). This is a required value, and will depend on several factors, which is why it is provided control to the user to choose it. If the chemical make-up of propane can not support the provided soot-yield, it will not run.

Ramp

```
&RAMP ID='q', T=0, F=0 /
&RAMP ID='q', T=30, F=1 /
&RAMP ID='q', T=180, F=1 /
&RAMP ID='q', T=300, F=0 /
```

The Ramp-lines specifies how the fire will change (ramp-up) throughout its lifespan.

The T-parameter defines that this line applies to the specified time in seconds.

The F-parameter defines the fraction of the total HRR a fire should have at the specified time.

In this case, the fire will reach max HRR after 30 seconds, and be completely gone by 300 seconds. In the actual *CoolEngine* -pipeline, these values will be decided based on fire curves and what vehicle is burning, as these commands allow for linear approximation which will be looked at later.

Surface

```
&SURF ID='fire', COLOR='RED', HRRPUA=replace_hrrpua,  RAMP_Q
   ='q' /
```

The Surf-line defines a surface, but is in this case used to define the fire.

The HRRPUA-parameter is the HRR (Heat Release Rate) per Unit Area. It effectively controls the HRR, as the fire area is defined to be a unit-square in the XB-parameter. HRR is measured in kW, HRRPUA in kW/m^2.

Vents

```
&VENT SURF_ID='fire', XB=63.5,64.5,-0.5,0.5,0.0,0.0, COLOR='
   RED' /
&VENT ID='Mesh Vent: Mesh01[0,0,0] [XMIN]', SURF_ID='OPEN',
   XB=0.0,0.0,-2.0,2.0,0.0,4.0/
```

```
&VENT ID='Mesh Vent: Mesh01[7,0,0] [XMAX]', SURF_ID='OPEN',
    XB=128.0,128.0,-2.0,2.0,0.0,4.0/
```

The Vent-lines define open areas in the mesh, or "holes" to put it simply.

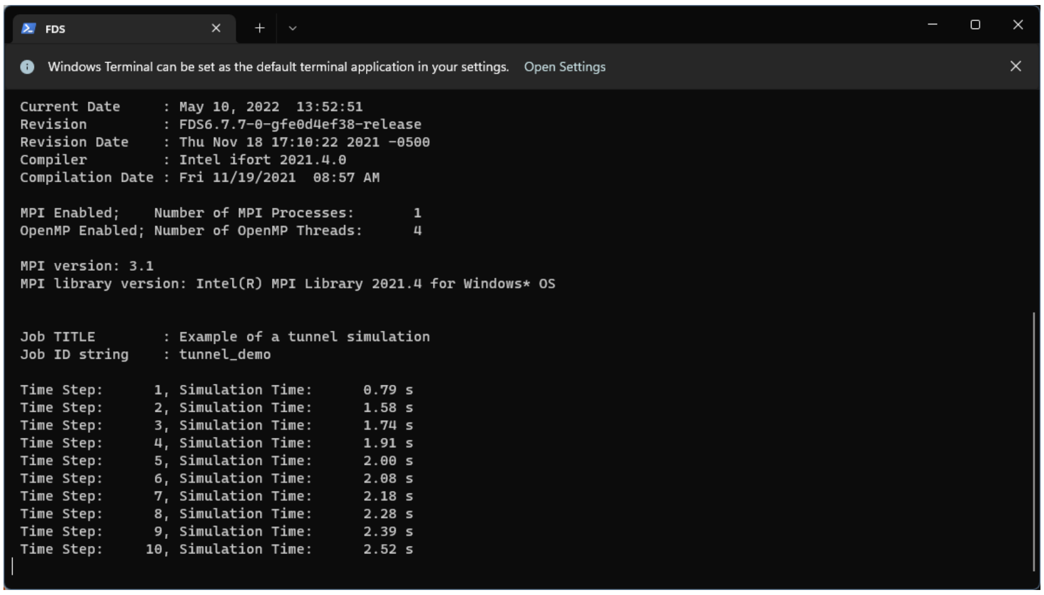
The Surf_Id-parameter defines what sort of opening this is. The two types used here are:

- OPEN - Meaning a standard hole
- fire - Meaning an opening for the fire to burn

Simulations require some sort of vent to function, just like fires need access to air to burn.

6.1.3 Running Simulation

The simulation is started via cli using the command: `fds_local simulation .fds` when the simulation.fds file is done generating. It will then create a subprocess running cmd, and displaying it as a popup to the user. The result of the simulation as binary Plot3D files are moved to a subdirectory called q_files for conversion to ASCII.


FDS
Windows Terminal can be set as the default terminal application in your settings.
Open Settings

```
Current Date     : May 10, 2022  13:52:51
Revision         : FDS6.7.7-0-gfe0d4ef38-release
Revision Date    : Thu Nov 18 17:10:22 2021 -0500
Compiler         : Intel ifort 2021.4.0
Compilation Date : Fri 11/19/2021  08:57 AM

MPI Enabled;    Number of MPI Processes:       1
OpenMP Enabled; Number of OpenMP Threads:      4

MPI version: 3.1
MPI library version: Intel(R) MPI Library 2021.4 for Windows* OS


Job TITLE        : Example of a tunnel simulation
Job ID string    : tunnel_demo

Time Step:      1, Simulation Time:      0.79 s
Time Step:      2, Simulation Time:      1.58 s
Time Step:      3, Simulation Time:      1.74 s
Time Step:      4, Simulation Time:      1.91 s
Time Step:      5, Simulation Time:      2.00 s
Time Step:      6, Simulation Time:      2.08 s
Time Step:      7, Simulation Time:      2.18 s
Time Step:      8, Simulation Time:      2.28 s
Time Step:      9, Simulation Time:      2.39 s
Time Step:     10, Simulation Time:      2.52 s
```

Figure 6.3: fds_local running a simulation.

Note that this approach of displaying the cli to the user takes advantage of FDS' innate way of revealing how far along the simulation has computed, it displays which time-value is the last one to be computed in real time. This provides the user a feel for how far along the simulation is.

6.1.4 FDS2ASCII

The idea behind FDS2ASCII, a program packaged with FDS, is simple: It allows the user to convert .q files, in this case, to ASCII. It gives the user a long list of all the files found for the given simulation, with indices for every file, and asks the user to specify which index it wants to convert.

Compiled With Newer Version

The included FDS2ASCII with FDS version 6.7.7 does not support more than 2000 binary files at once in the listing, and will not index the files properly if it exceeds the maximum. Therefore a newer version of FDS2ASCII was compiled specifically to circumvent this problem, since the file limit was changed to 500000 after the latest FDS release. It is included under the FDS directory specified in subsection. This made it easier to see which files FDS2ASCII is working on, and was necessary to make the scripting work.

```
INTEGER, PARAMETER :: FILE_DIM = 500000
```

Listing 6.2: FDS2ASCII file limit, line 21

It also includes a modified build of FDS2ASCII, which does not print all the files available every time it completes a file conversion. This creates less overhead in the computation and makes the input more readable for the user. The way this achieved by commenting out the code in listing 6.3.

```
      ! WRITE(6,'(I6,3X,A,A,I4,A,F5.0)') I,TRIM(PL3D_FILE(I))
   ,', MESH ',PL3D_MESH(I),', Time: ',PL3D_TIME(I)
```

Listing 6.3: Commented out line from FDS2ASCII, line 274

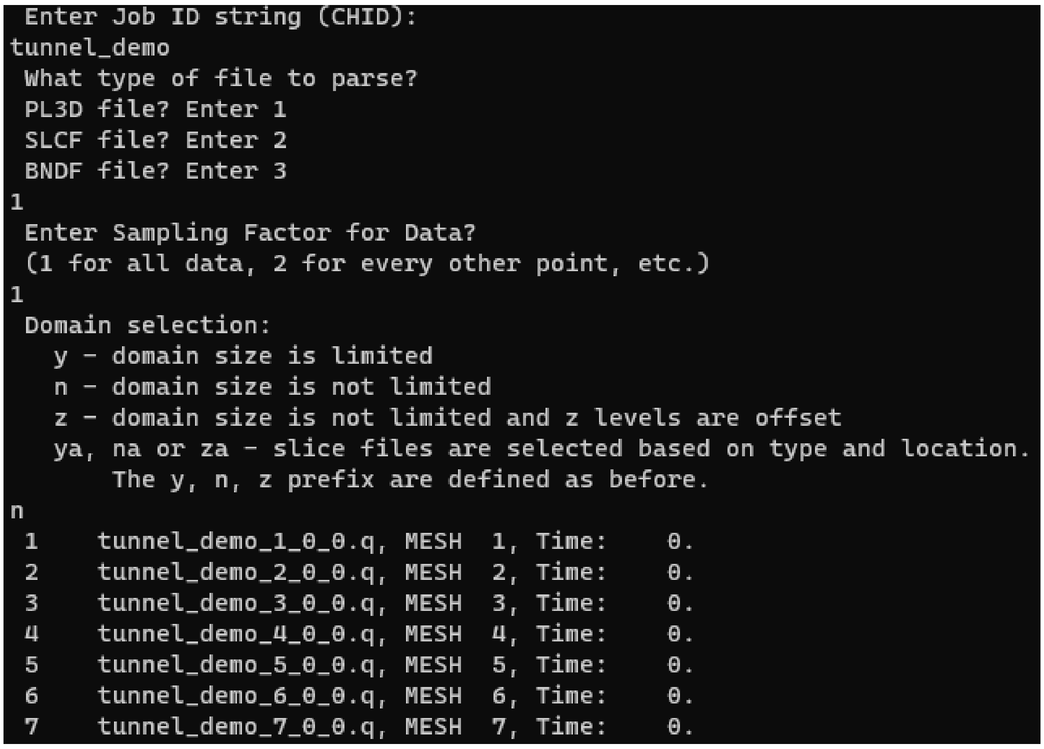
Specifying Data

When first calling the command, the software asks for some rudimentary information which is needed for it to function. It will first asks for the *chid*, which identifies which simulation in the directory it will convert. It then needs the ask for the file format and the domain size.

Typical input:

1. Specify chid
2. Type of data/file format
    - **Plot3D**
    - SLCF
    - BNDF
3. Sampling Factor
    - **All data**
    - Every other data
    - Etc.
4. Domain
    - Limited domain
    - **Unlimited domain**
    - Type and location

(Bold text is the default selected when getting this data to Unity)



```
 Enter Job ID string (CHID):
tunnel_demo
 What type of file to parse?
 PL3D file? Enter 1
 SLCF file? Enter 2
 BNDF file? Enter 3
1
 Enter Sampling Factor for Data?
 (1 for all data, 2 for every other point, etc.)
1
 Domain selection:
   y - domain size is limited
   n - domain size is not limited
   z - domain size is not limited and z levels are offset
   ya, na or za - slice files are selected based on type and location.
       The y, n, z prefix are defined as before.
n
 1    tunnel_demo_1_0_0.q, MESH  1, Time:    0.
 2    tunnel_demo_2_0_0.q, MESH  2, Time:    0.
 3    tunnel_demo_3_0_0.q, MESH  3, Time:    0.
 4    tunnel_demo_4_0_0.q, MESH  4, Time:    0.
 5    tunnel_demo_5_0_0.q, MESH  5, Time:    0.
 6    tunnel_demo_6_0_0.q, MESH  6, Time:    0.
 7    tunnel_demo_7_0_0.q, MESH  7, Time:    0.
```

Figure 6.4: Example of FDS2ASCII usecase, mirroring how it is used in the *CoolEngine*

Translation Table

The first FDS2ASCII build will be called first, input the standard parameters to receive all the Plot3D but will cancel when a file is prompted for. This output will be used to construct a translation table in code, since the FDS2ASCII index for a file is not sorted in any discernible pattern. This translation table will be used to create a script file.

Converting to ASCII

After the translation table has been created, the other build of FDS2ASCII will be called with the script file via the command redirection operator in command prompt. It will then convert all the files to .csv-files.

```
ProcessInfo = new ProcessStartInfo("cmd.exe", "/C " + $"{command} < {script_file}");
```

Listing 6.4: cli.cs, to start the subprocess

```
public static void CreateScriptFile(string[] Q_Files, Dictionary<string, int> Table, string base_output_name, string directory)
{
     using (StreamWriter sw = File.CreateText($"{directory}/script.txt"))
     {
         sw.WriteLine("./simulation");
         sw.WriteLine("1");
         sw.WriteLine("1");
         sw.WriteLine("n");
         for (int i = 0; i < Q_Files.Length; i++)
         {
             string[] q_files_array = Q_Files[i].Split('\\');
             string q_file_name = q_files_array[q_files_array.Length - 1];
             sw.WriteLine(Table[q_file_name]);
             string file_name = Path.GetFileName(Q_Files[i]);
             var name_list = file_name.Split('_');
             string ms_percent = name_list[name_list.Length - 1].Split('.')[0];
             string timestep_seconds = name_list[name_list.Length - 2];
             string mesh_num = name_list[name_list.Length - 3];



             sw.WriteLine($"{base_output_name}_{mesh_num}_{timestep_seconds}_{ms_percent}.csv");
         }
         sw.WriteLine("0");
     }
}
```

Listing 6.5: FDS2ASCII.cs, to use the table to convert all the files

Aggregation

As mentioned previously, the domain is split into 8 meshes, so they have to be aggregated into one file for Unity's sake.

All the text files which have a matching timestamp will be merged into one file. It will then be moved into Assets/CFD/FDS/Output/*scenario_ id*.

```
for (int i = 0; i < CSV_Files.Length; i++)
    string[] file_array = CSV_Files[i].Split('_');
    string milliseconds = file_array[file_array.Length - 1].
  Split()[0;
    string seconds = file_array[file_array.Length - 2];
    string output_file_path = $"./Assets/CFD/FDS/Output/{Id
  }/{base_output_name}_{seconds}_{milliseconds}.csv";
    if (!File.Exists(output_file_path))
    {
        using (var strm = File.Create(output_file_path)) { }
    }
    //Merges file with others of same mesh
    using (var outputStream = File.Open(output_file_path,
  FileMode.Append))
    {
        using (var inputStream = File.OpenRead(CSV_Files[i]))
        {
            inputStream.CopyTo(outputStream);
        }
    }
}
```

Listing 6.6: Aggregation of files in FDS2ASCII.cs

If a file does not exist which matches the corresponding time stamp, it will create an empty csv file with the appropriate name. Afterwards it will append the content to the end of the file created.

6.2 Unity Logic

6.2.1 Scenario Storage

Scenarios created via the scenarios menu is stored as a JSON file as seen in listing 6.7. It sums up all the necessary settings for creating FDS input file. It also includes the information for changing the size of the fire when inside a tunnel.

```
{
    "Records": [{
        "id": "528dd7fc-2c18-4eaa-9bc5-ef8cb3224708",
        "name": "test",
        "soot_yield": "0.15",
        "time": "10",
        "fire_material": {
            "name": "Car",
            "hrrpua": 3000,
            "TimeToMax": 300,
            "TimeToDiminish": 1500,
            "TimeToFinished": 2700
        }
    }]
}
```

Listing 6.7: scenarios.json example

This is the link between simulation.fds and the user input, where the information the user gives is stored, and sent forward to FDS and further down the *CoolEngine* - pipeline.

6.2.2 Lethality Calculation

The Probit

One of the most powerful aspects of a software that uses *CoolEngine* is how it allows the user to be impacted directly by the simulation scenario. Similar software may have to depend on a several assumptions on how e.g smoke would behave in regards to the user, but this software can use scientific models based on physical data.

The most clear example of this is the way the software calculates the smoke lethality. The idea of smoke lethality is to illustrate the likelihood of a person dying if they are exposed to a certain dosage of smoke for a certain amount of time.

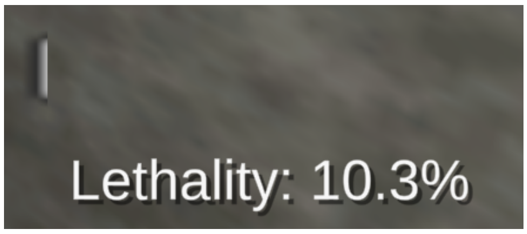


Figure 6.5: The current chance of a player dying of smoke poisoning, displayed both as a numerical percentage and a white bar.

This value is calculated based upon equation 6.1, and can be found in a paper from the Health and Safety Executive of the U.K. about human vulnerability in offshore environments [35].

$$Pr = -38.8 + 3.7ln(C * t) \tag{6.1}$$

Where C is the concentration of CO measured in mg/m^3, and t is the time in minutes.

This formula leads to a probit, a value which corresponds to how many percent lethal a certain amount of exposure is. It is worth noting that this project approximates soot to being chemically equivalent to CO for the purposes of this calculation. This is not an extremely accurate approximation in physical terms, but it works well for keeping the calculations and design simple.

Plot3D outputs the soot density per cell in kg/m^3, and Unity can keep track of how much time has passed inbetween frames. Thus: the probit can be calculated.

The Soot Density

But before that, there must be done a last approximation. Since the smoke has been implemented as a particle system, consideration must be taken. Particles in Unity can not carry data on their own, as they are meant to be lightweight visualizations of certain visual effects (like smoke). Therefore they can not directly carry the soot density per cell.

This is circumvented, however, by visualizing the smoke in a particular way. When the simulation data is loaded into Unity, it also finds out the max soot density in the simulation. It thereafter calculates, for each particle to be rendered, how many particles each soot density should be. This is done by calculating how much of the max density the cell has in density, $\frac{currentDensity}{maxDensity}$, and then multiplying that by three; a constant chosen since it allows for both pretty detailed data, and for satisfactory performance.

This means, at the end of the day, that each particle has a fixed amount of mg/m^3 it will afflict the user with, and more particles will be rendered where the smoke is extra thick.

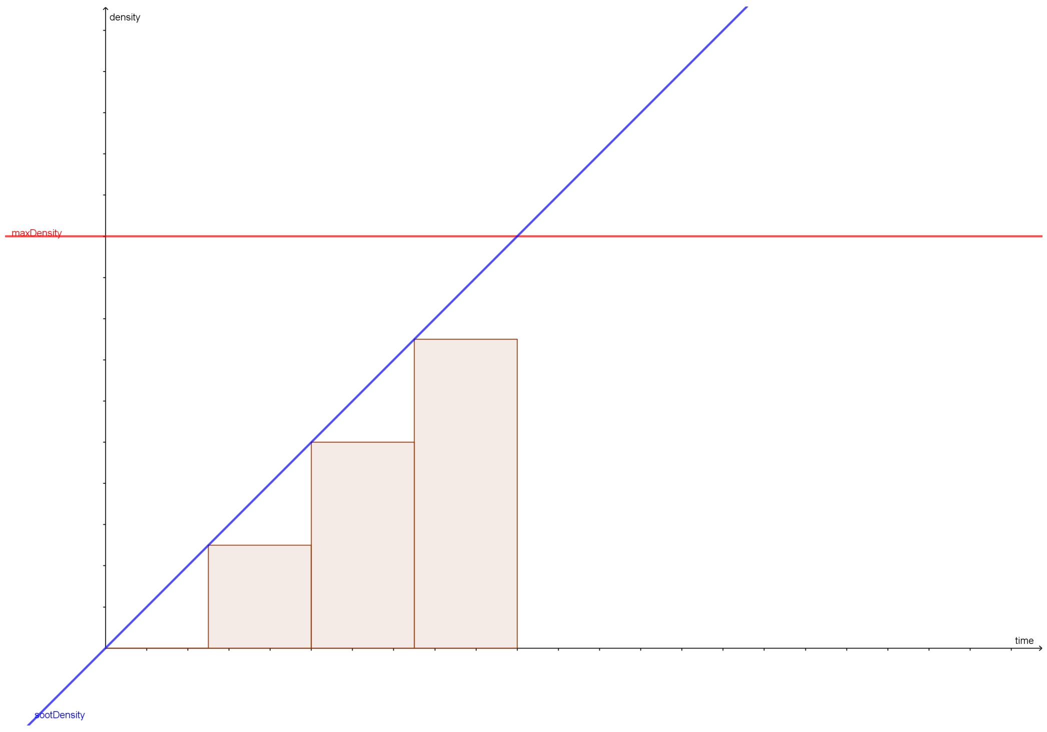


Figure 6.6: A rough sketch of how the density approximation works. The linear blue graph represents the density in a cell as a function of time, the red constant represents the max density, and the rectangles represent the fixed values one, two and three particles will afflict. Figure created with GeoGebra [36].

Note that the soot density is to be considered neither a function of time nor a linear function, but it serves to illustrate the point.

Now that an approximation of the density in each cell is established, the only remaining functionality missing is the interaction with the player. When the user collides with a particle in Unity, a *OnParticleCollision*-event is triggered, and when this collision occurs, the user receives lethality-damage.

```
    void OnParticleCollision(GameObject other)
    {
        if (other.tag == "smoke3D")
        {
            ct += ConvertDensity(SETTINGS.densityPerParticle)
    * ConvertTime(timeDelta);
        }
    }
```

Listing 6.8: PlayerLethality.cs

The use of tags for the particles makes sure only the correct particles trigger lethality-damage.

As a theoretical example: A user may walk into cell C_1, this cell has a soot density corresponding to 2/3 of the max value of soot in this simulation. Unity will therefore render $2/3 * 3 = 2$ particles in that cell-coordinate, and the user will increase their probit by $2 * maxDensity * timeBetweenFrames$.

The Lethality

Now that there is established both a way to calculate a probit and a way for the user in Unity to be affected by this, the data must be translated into the lethality which the user will see in the simulation.

The probit is calculated into a percentage with a formula from a research paper from 2005 also regarding offshore platforms [37].

$$P = 50 * (1 + \frac{Pr - 5}{|Pr - 5|} * erf(\frac{|Pr - 5|}{\sqrt{2}})) \tag{6.2}$$

Where erf is the error-function from statistics.

Using this calculation yields the same results as those in table 25 in the technical report with the calculations [35]. This is verified by *PlayerLethalityTests.cs*, which passes in the test-runner.

Table 6.1: Which ppm-values over which minute-values lead to a certain lethality in percentage. All the values are from table 25 in [35]

Concentration of CO in ppm / Exposure time needed for % lethality			
1-5%		50%	
1100	60 min	2000	60 min
2200	30 min	4000	30 min
3200	20 min	6000	20 min
6400	10 min	12000	10 min
13000	5 min	24000	5 min

```
 [Test]
 public void testProbitAndLethality()
 {
     GameObject gam = new GameObject();
     PlayerLethality pl = gam.AddComponent<PlayerLethality
>();

     double ct = 24000 * 5 / 0.862; //We divide by 0.862
to convert from ppmV to mg/m^3
     int result = (int)pl.LethalityCalculation(pl.
ProbitCalculation(ct));
     Assert.AreEqual(50, result);

     ct = 13000 * 5 / 0.862;
     result = (int)pl.LethalityCalculation(pl.
ProbitCalculation(ct));
     Assert.AreEqual(1, result);
 }
```

Listing 6.9: PlayerLethalityTests.cs

Limitations of This Model

There are some limitations of this model which is worth pointing out.

The particles are not extremely accurate compared to the original mesh design. Since the particles will not match entirely with the cells in the design, two possible inaccuracies may occur:

1. There may be areas which are in a cell, but are not colliding with the circular particle

2. There are areas where multiple particles overlap, and create excessive amounts of exposure

The smoke particles in this project are pretty large, so the latter is the more likely scenario.

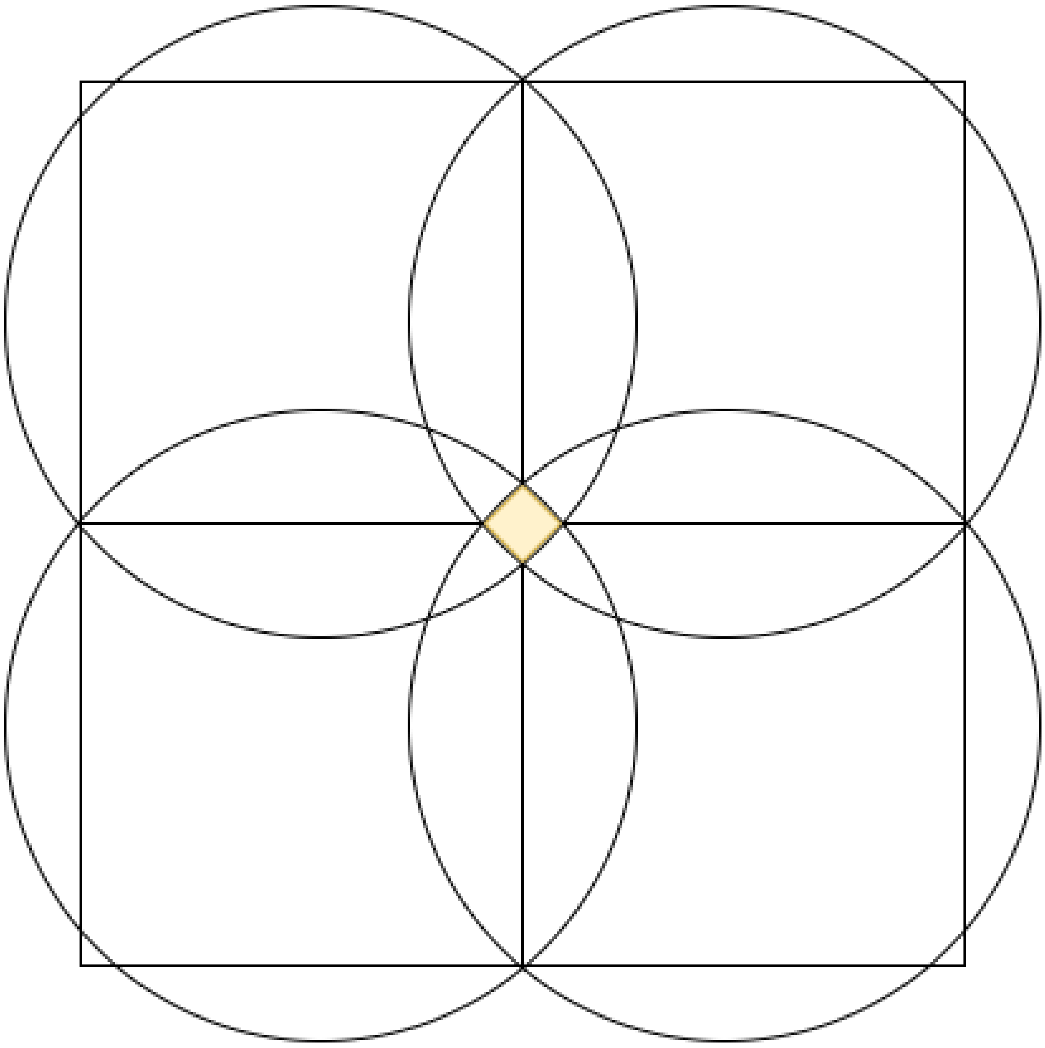


Figure 6.7: A figure demonstrating how large particles representing cells may overlap and create intense zones of massive smoke exposure. Made with Diagrams.net [33]

This may lead to inaccuracies, and may illustrate why it is preferrable to implement the simulation logic in another data-structure than the particles in the future.

The other issue is that this is a extremely slow rising value. Even though carbon-monoxide and smoke are poisonous, it takes a great deal of time and exposure at the concentrations gained through this simulation to receive impactful values.

This is why it should, eventually, be implemented similar logic to this but for heat radiation exposure. The same model for smoke toxicity may be used for heat radiation with probits and lethalities, in fact, the papers used to build this smoke model touches on the same topic [35][37]. This would offer users a clearer response, as heat radiation needs less exposure to become lethal.

6.2.3 Dynamic Fire and Fire Curves

The FDS simulation already takes care of the growth of the fire model, as previously discussed. But this data is not outputed to the Plot3D data which has been covered so far. Thus another solution has to be used here.

The chosen way to model the fire in this project is linear aproximation of the fire curve. This is mainly because it is easier to work with and therefore adheres to KISS, but it also is a well-accepted way to model fires based upon a book on Tunnel Fire Dynamics from 2015 [38]. Specifically figure 6.1 from chapter 6 demonstrating how the linear aproximation compares to the exponential model.

When using linear approximation, all that needs to be known is:

- How much is maximum HRR
- How long until the fire has peaked in terms of HRR
- How long until the fire will reduce its HRR
- How long until the fire is finished

All this information can be found in the book regarding the topic [38], [39], this information has been reproduced in the table below for brevity. A custom fire curve for the purposes of this project has been added.

Table 6.2: Fire curves based upon table 6.5 from the book [38] which is based upon French regulations [39], note that the car entry has been modified to fit with just one car

Vehicle	Q_Max	t_Max	$t\_Diminish$	t_Finish
Car	3	5	25	45
Truck	30	10	70	100
Basic	2	0.5	3	5

Where Q_Max is the maximum HRR in MW, all t-values are in minutes, t_Max is the time for the HRR to maximize, $t\_Diminish$ is the time for it to start decreasing, t_Finish is the time the fire stops.

The basic vehicle type is just data that fits well into the limitations of the simulation. As the simulation should rarely be longer than 5 minutes unless the user is prepared to wait a significant amount of time for it to compute.

The user can choose when creating their scenario which of these three vehicles they want, and the simulation will run with these parameters. The same source that provide this information to FDS, a .json file, provides it to Unity at runtime to model the fire. It scales the size of the fire linearly with the fire curves linear growth in HRR.

```
            if (raw_time < _fireMaxTime)
            {
                scale = raw_time/_fireMaxTime;
                var fire = this.gameObject.transform.Find("FirePosition").Find("Fire(Clone)");
                foreach(Transform child in fire)
                {
                    child.localScale = new Vector3(scale, scale, scale);
                }
            }
```

Listing 6.10: CFDFire.cs, showing the scaling of the fire when it increases in the beginning of the fire.

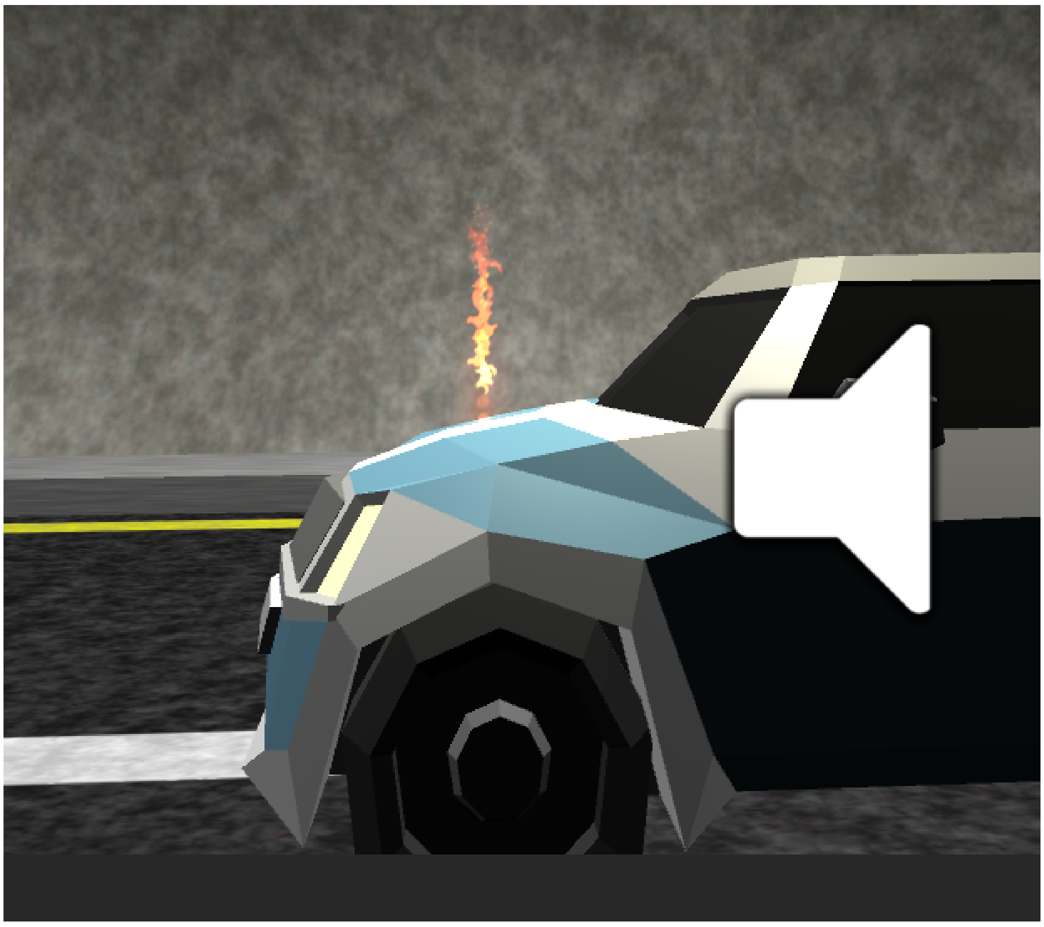


Figure 6.8: The fire still small at the beginning of a simulation

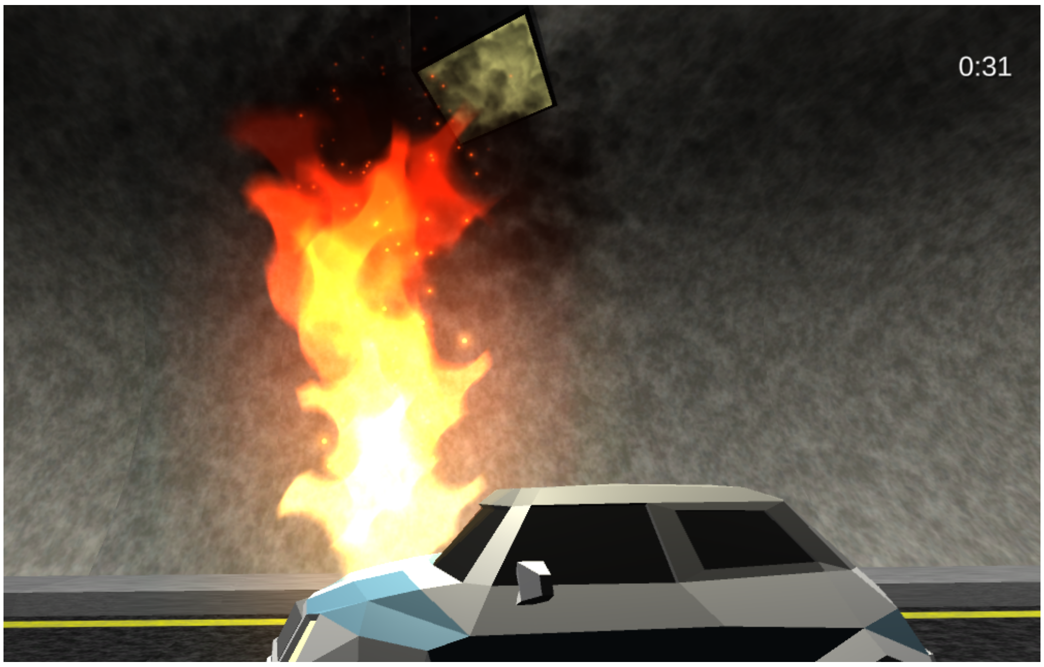


Figure 6.9: The fire at its maximum size

This approach has the benefit that it is not extremely computationally heavy. Meaning the fire will still behave as intended after the software has run out of Plot3D data.

6.2.4 Parsing FDS Data

Data Structure

The data structure is defined as a 2-dimensional list, or a nested list, containing data points as defined in listing 6.11.

```
    public class DataPoint
    {
        public double[] Position;
        public double Soot_Density;
        public TimeSpan Time_Stamp;
    }
}
```

Listing 6.11: DataPoint.cs

Each item in the highest order list will be equivalent to a point in time, while the second list inside will be equivalent to lines inside the file.

Reading Files

The CFD parser loops through all the files in the scenario directory with the .csv file extension, and loads them into memory using the specified structure. When type casting, the culture is specified because of the specific format FDS2ASCII uses when converting to ASCII.

```
using (var reader = new StreamReader(cfd_file))
{
    while (!reader.EndOfStream)
    {
        //Parses filename to TimeSpan object
        Date_Parser Parser = new Date_Parser();
        TimeSpan time_stamp = Parser.FileNameToTimeSpan(cfd_file);

        var line = reader.ReadLine();
        var values = line.Split(',');
        var first_value = values[0].Trim();
        //Ignore strings, like X,Y,Z
        if (!double.TryParse(first_value, System.Globalization.NumberStyles.Float, CultureInfo.CreateSpecificCulture("en-US"), out double _))
        {
            continue;
        }
        if (line == "\n" || line == "")
        {
            break;
        }

        //Trims string, and converts from string to array. Second argument because csv file decimal numbers are seperated by dot and not comma
        double x = double.Parse(values[0].Trim(), System.Globalization.NumberStyles.Float, CultureInfo.CreateSpecificCulture("en-US")); //System.Globalization.CultureInfo.InvariantCulture
        double y = double.Parse(values[1].Trim(), System.Globalization.NumberStyles.Float, CultureInfo.CreateSpecificCulture("en-US")); //System.Globalization.CultureInfo.InvariantCulture
        double z = double.Parse(values[2].Trim(), System.
    Globalization.NumberStyles.Float, CultureInfo.
    CreateSpecificCulture("en-US")); //System.Globalization.
    CultureInfo.InvariantCulture
        double soot_density = double.Parse(values[3].Trim(),
    System.Globalization.NumberStyles.Float, CultureInfo.
    CreateSpecificCulture("en-US"));
        if (soot_density > _densityM) { _densityM =
    soot_density; } //Simple track of max density
        double[] position_array = new double[] { x, y, z };
        DataPoint data_point = new DataPoint();
        data_point.Position = position_array;
        data_point.Soot_Density = soot_density;
        data_point.Time_Stamp = time_stamp;
        data_points.Add(data_point);
    }
}
```

Listing 6.12: Excerpt from CFD_Parser.cs

This happens when a user chooses where they want to spawn in the tunnel, which may take some time to load.

6.2.5 Particle System

With the template fds input file, the simulation created is in the form of a hyperrectangle. To find the correct placement, a few challenges has to be overcome.

It is worth noting before any details are discussed, that the z-axis is "up" in FDS, while it is the y-axis which is "up" in Unity.

Finding Start of Fire

First Unity needs to know where the fire starts in the simulation, to place it correctly overlapping where the fire is located.

Looping through the first second of the simulation, and then finding the highest amount of soot density will offer a decent approximation of where the fire starts. This is a fair assumption, as there will typically be the most smoke right above the fire in this scenario.

Alignment

To align the particle system, an assumption is first made about how the paricle system is initially aligned. It is assumed it will be pointing towards the positive z axis, and is also starting out sideways.

To align it properly; it will be rotated 90 degrees on the x-axis to fix it being sideways.

Then it is rotated by the degrees between the direction vector of the particle system and the road direction vector closest to the fire position.

Lastly, the degrees to rotate is multiplied by the mathematical sign on the z-axis position the particle system is going to move to, to take into account the tunnel being in different quadrants.

```
z_angle = Angle(particle_system_direction, road_direction)
z_angle *= Math.Sign(trans.position.z)
particle_system.rotation = (-90, 0, z_angle)
```

Listing 6.13: Pseudocode for rotation of particle system

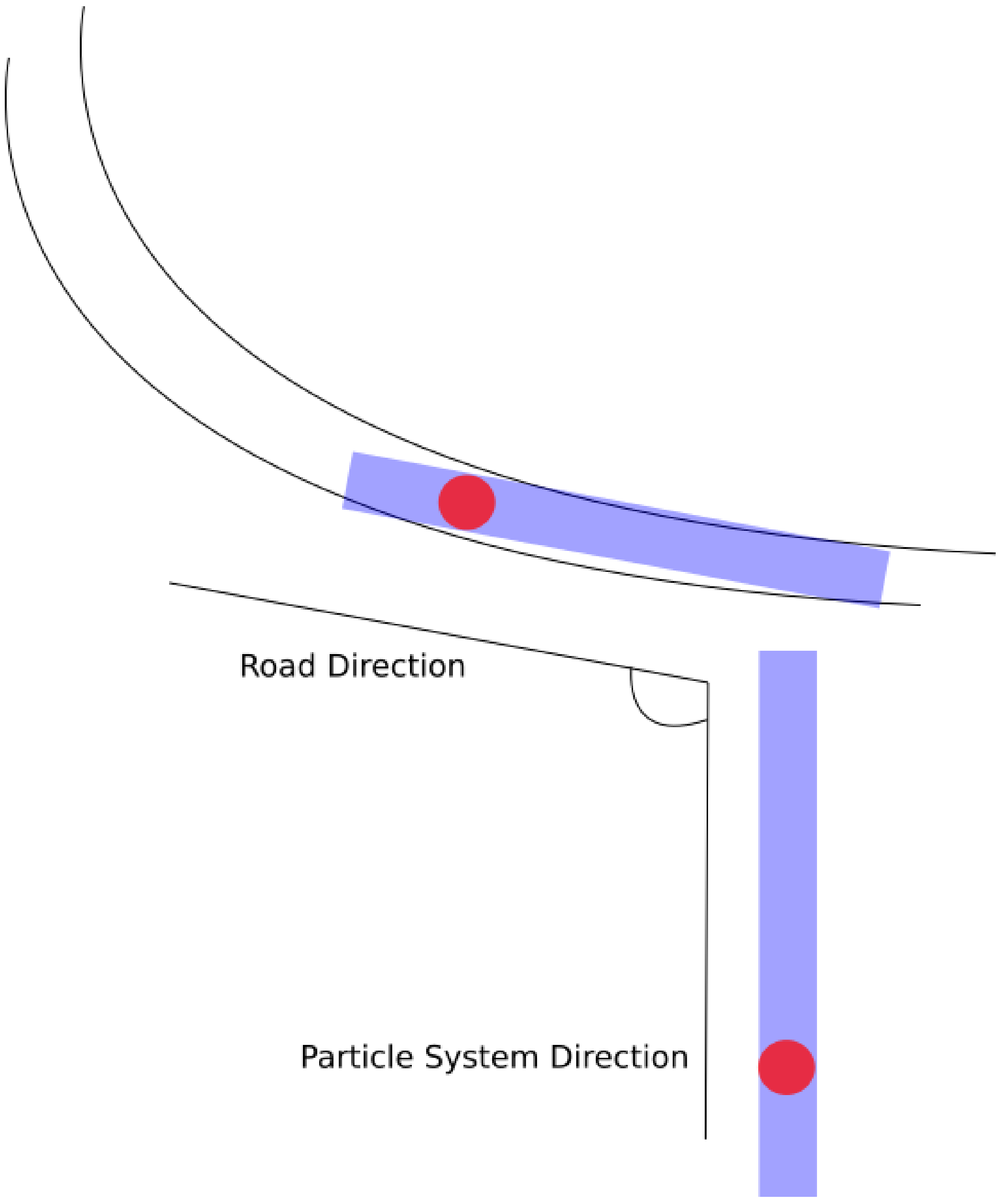


Figure 6.10: Aligning the particle system with the tunnel where blue is particle system and red is fire position. Made with Diagrams.net [33]

Position Offset

After finding the start of the fire, the offsets on the x-axis and z-axis needs to be calculated as illustrated in figure 6.11, to make the start of the fire in the simulation match that of the fire in Unity.

The offset is the x and the z coordinates, illustrated in the figure as the catheti of the triangle, which needs to be added to the particle system position to make it align correctly.

The angle between the road direction and the *z_axis* variable is the same angle which is illustrated in green in figure 6.11.

The catheti can then be calculated as displayed in listing 6.14 where *position[0]* is the distance from the origin of the particle system to the start of the fire, which is also the hypotenuse. This is achieved using the definition of sine and cosine.

```
        Vector3 z_axis = new Vector3(0, 0, trans.position.z);
        Vector3 road_direction = TMath.DirectionVectorBetweenRoads(GetRoad(fire_index), GetRoad(fire_index + 1));

        //Calculate the X and Z offset when rotating the tunnel on a non-straight road
        //Find absolute value of the fire start position in-case the positions are negative
        var x = Math.Abs(position[0] * Math.Sin(Vector3.Angle(z_axis, road_direction) * Math.PI/180));
        var z = Math.Abs(position[0] * Math.Cos(Vector3.Angle(z_axis, road_direction) * Math.PI/180));
```

Listing 6.14: TScene.cs

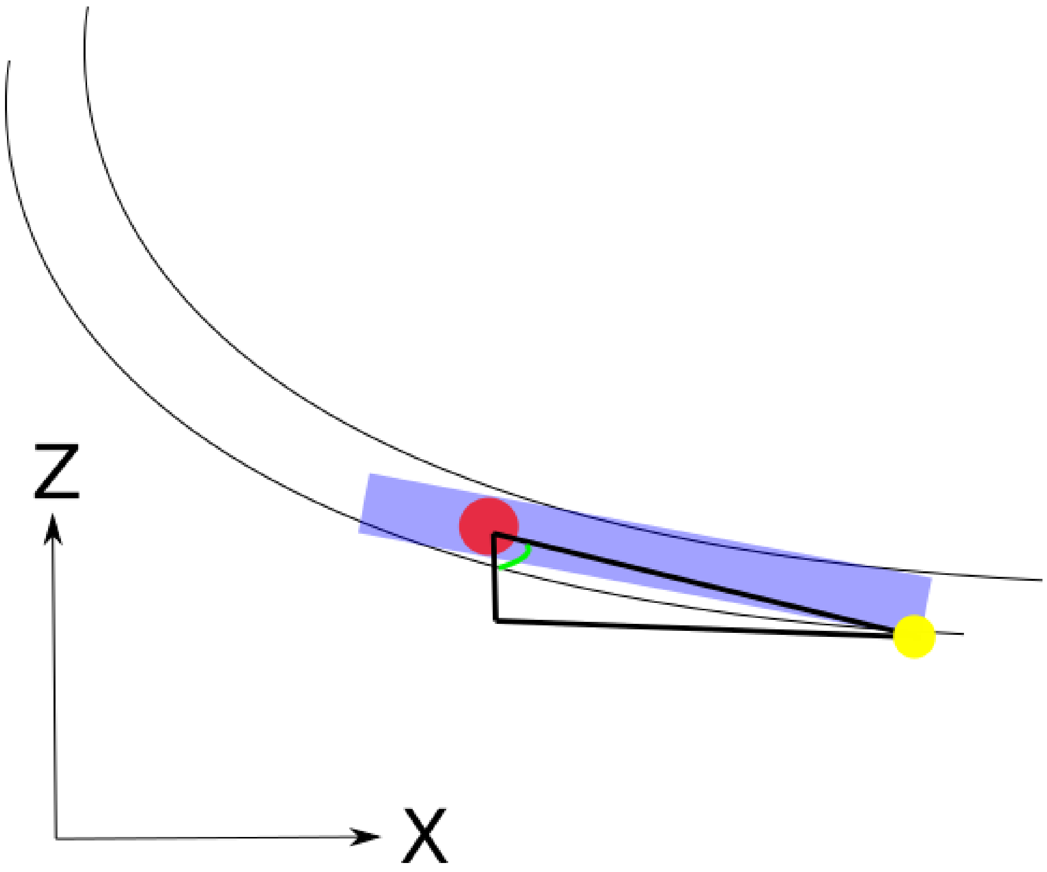


Figure 6.11: Example of calculating offsets, where blue is simulation, red is fire position, green is the angle used in computation and yellow is the original simulation fire position.
Both a rotation and a position offset is needed to make the particle system align correctly with the tunnel. Made with Diagrams.net[33]

The triangle is defined by the particle system position and the fire position which outlines the hypotenuse.

6.2.6 Choose Spawn Location

Choosing spawn location is implemented using a separate scene generating the selected tunnel. It generates the whole tunnel, then filtering away every game object which is not a road. The highest and lowest x and z coordinates from the roads is found, and calculating the midpoints between the respective axes. It then uses those values for the camera coordinates.

```
        Center = new Vector2((X_Upper + X_Lower)/2, (Z_Upper
    + Z_Lower)/2);
```

Listing 6.15: SelectTunnelObj.cs

The y-axis coordinate is calculated via a constant, 4.5, multiplied by the amount of road segments. This constant is found by experimentation, and leads to the camera being placed high enough for most tunnels.

```
        Camera.main.transform.position = new Vector3(Center.x
    , (float)(4.5 * index), Center.y);
```

Listing 6.16: SelectTunnelObj.cs

When it loops through all the game objects and removing everything not being roads, it also creates a dictionary with the road game object as the key and the index for the road as the value.

Figure 6.12: Player choosing spawn location

To find what road segment has been clicked on, a ray is cast from the camera, and returns the game object. It then compares it to the table created when the tunnel was generated to find the index of the road. This is set in settings as the spawn index, which is read when the tunnel scene script runs.

6.2.7 Traffic Generation

Pathfinding

The path generated for the cars is an ordered list of vectors representing the coordinates for all the road segments. This is found by iterating over all the roads, which is already in the correct order from one tunnel end to the other.

Driving

The car is moving by interpolating positions between the road positions with the built in `.MoveTowards()` which takes a start vector, end vector and a step. [40] The step is calculated by `speed*Time.timeDelta`, where the *timeDelta* is the difference in time between the previous frame and current frame. This is done every frame the software renders, and at 60 frames per second the time delta would be $1/60 = 16.\overline{66}ms$. The target for the next road also has a y-axis offset, since the the car would be inside the road if only the center y coordinate was used. When it arrives at the next road segment, is where it will rotate.

```
            transform.position = Vector3.MoveTowards(
    transform.position, target_position, step);
```

Listing 6.17: Car.cs

To rotate the car properly, the vector between the current road and the next road is first calculated. It will then use those angles to change the rotation of the car gameobject, and applying a 90 degree and 180 degree rotation to make it match the tunnels rotation.

When driving, the cars will cast a ray in front of it to check for obstacles. It will then brake if it detects a gameobject with the tag "Car".

6.2.8 Updating Object Placement

Wall Ceiling Problem

To achieve being able to input varying sized tunnels, the tunnel placement code from Self Rescue Game had to be updated. The code originally assumed that all tunnel segments had a single encompassing wall segment. This, however, is not the case: The procedural modeling software does not necessarily produce models with only one single wall per segment, and it will vary depending on the tunnel dimensions. This is shown in the figures below.

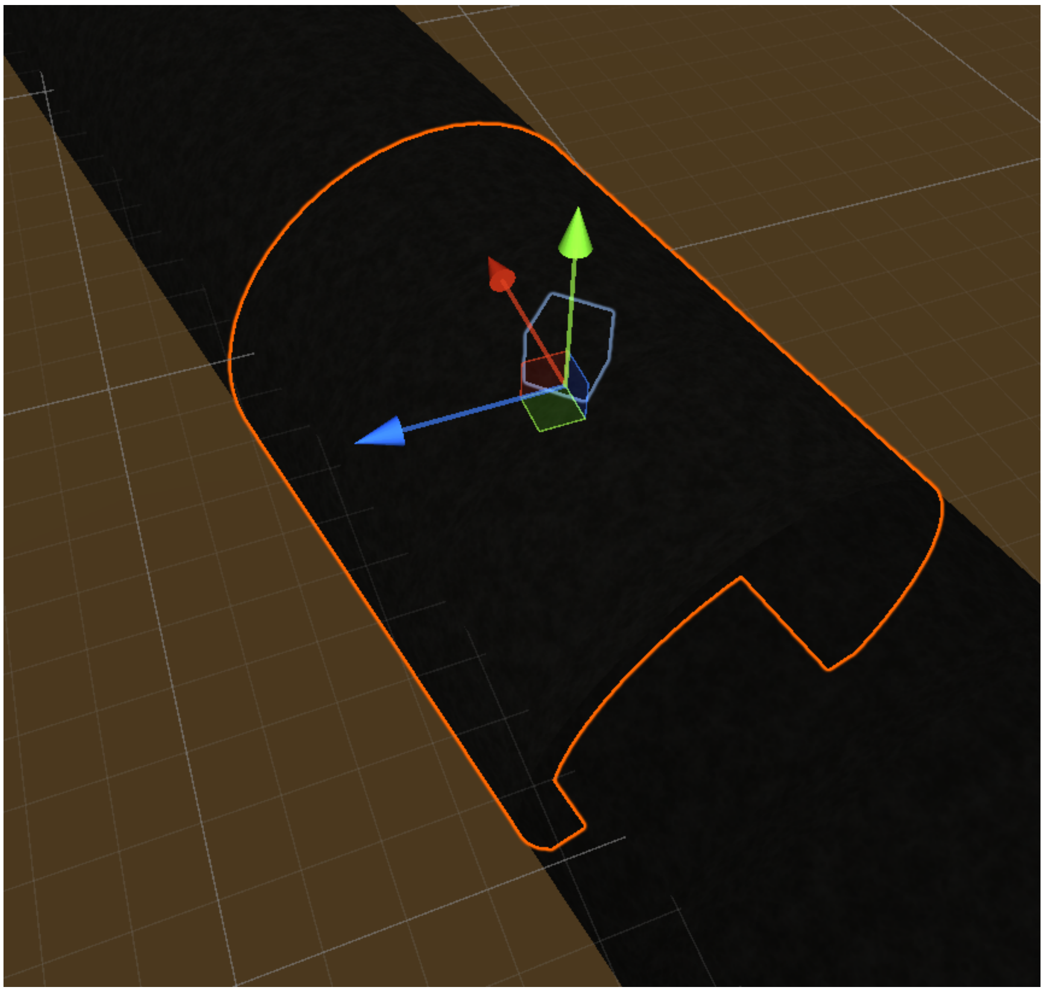


Figure 6.13: A tunnel wall segment selected, where walls and ceiling is all one object.

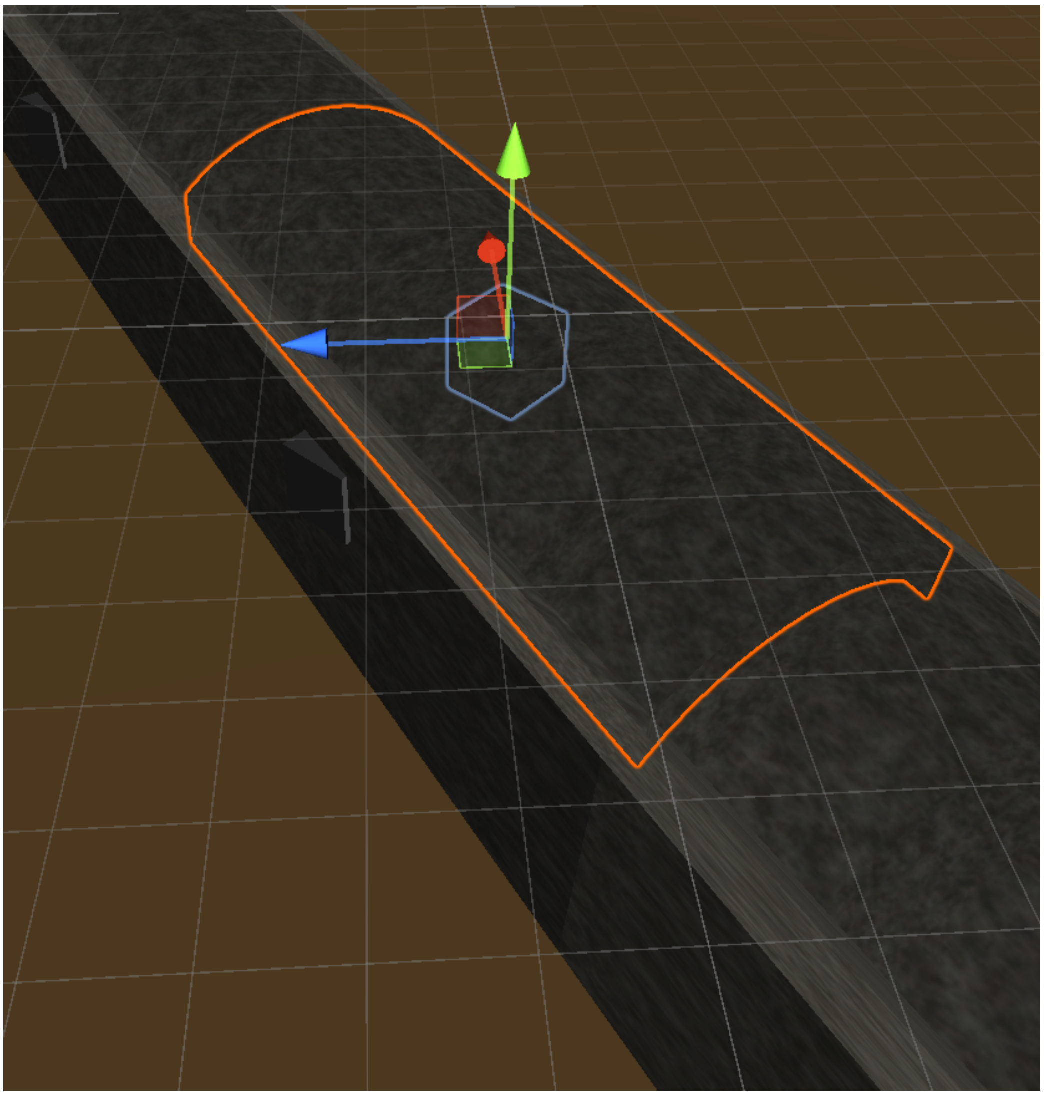


Figure 6.14: A tunnel ceiling part of the segment, where the left and right part of the wall are different objects. Making it three total instead of one.

The code previously used for the placement of things always assumed that there were a single encompassing wall for each road segment, but this is not the case as has been proven above. This means that the code would place things incorrectly because of this assumption.

Signs

The aforementioned faulty assumption regarding the meshes lead directly into the placement of the exit signs, as these objects also assumed this. This was remedied by taking the amount of wall-segments and divide them by the amount of road-segments, leading to a value of one if there is a single wall and a value of three if there are three. This could later be used to make sure only every third wall segment received a sign, and then the algorithm worked.

```
int walls_per_segment = (int)Math.Round((double)(walls.Count
    / roads.Count));
for (int i = 2; i < walls.Count; i+=walls_per_segment)
{
    if (walls_per_segment > 1 || i % 5 == 0)
...
```

Listing 6.18: DirectionSigns.cs, only looping over the correct wall segments to place signs.

Lights

The prefab was modified to look slightly more realistic, in addition to lighting up the roof of the tunnel which the earlier version did not do.

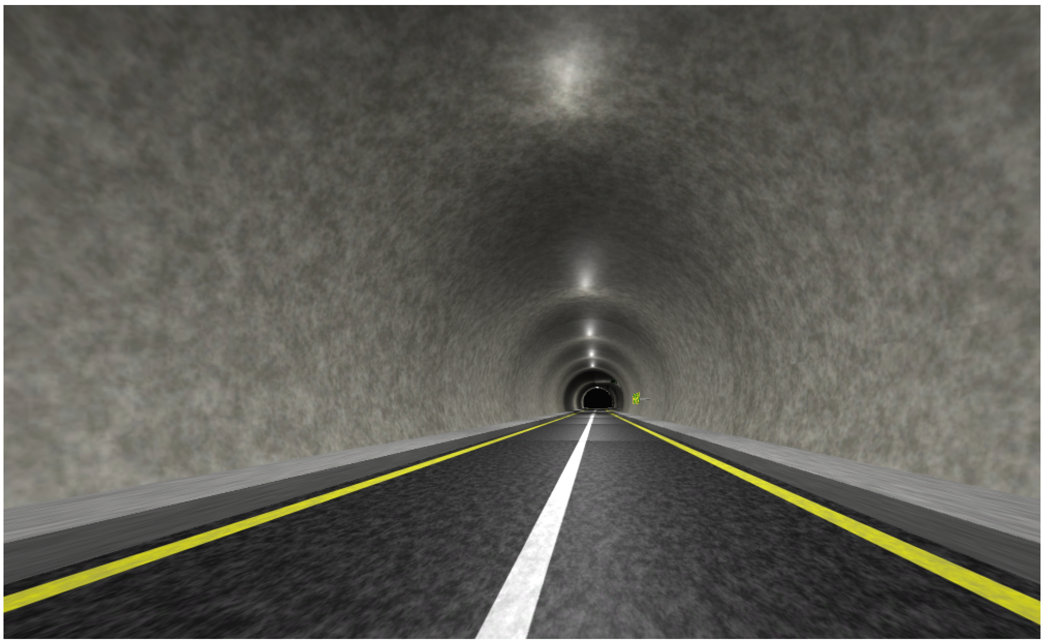
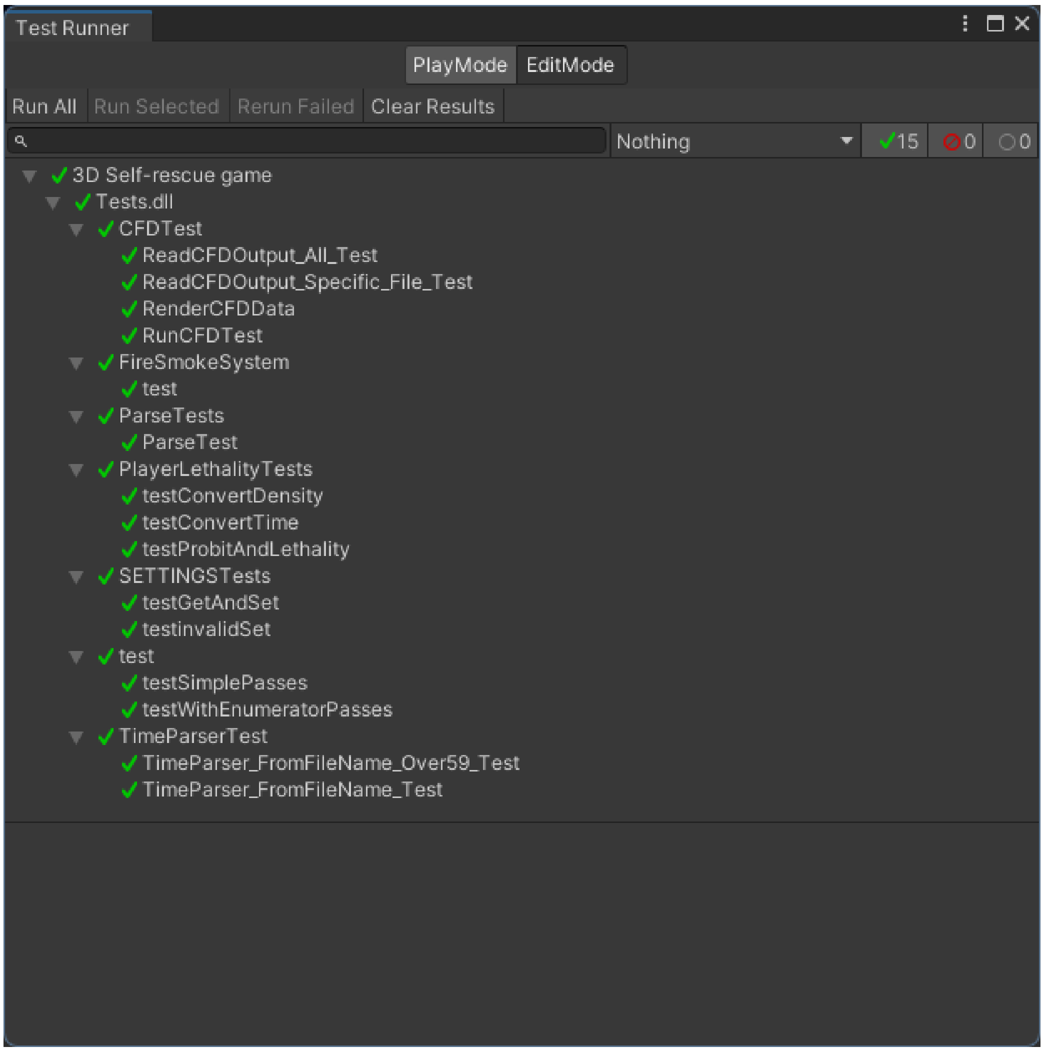


Figure 6.15: Updated lights, more reminiscent of standard lighting in tunnel compared with the earlier version.

6.2.9 Testing of Fundamental Scripts

As described in the chapter about the project background, test-driven design is a virtue. And although this project can not claim to have followed all the paradigms of that philosophy exactly, it has used testing via the Unity Test Framework to test some of the most fundamental scripts in the project.



```
✓ 3D Self-rescue game
  ✓ Tests.dll
    ✓ CFDTest
      ✓ ReadCFDOutput_All_Test
      ✓ ReadCFDOutput_Specific_File_Test
      ✓ RenderCFDData
      ✓ RunCFDTest
    ✓ FireSmokeSystem
      ✓ test
    ✓ ParseTests
      ✓ ParseTest
    ✓ PlayerLethalityTests
      ✓ testConvertDensity
      ✓ testConvertTime
      ✓ testProbitAndLethality
    ✓ SETTINGSTests
      ✓ testGetAndSet
      ✓ testinvalidSet
    ✓ test
      ✓ testSimplePasses
      ✓ testWithEnumeratorPasses
    ✓ TimeParserTest
      ✓ TimeParser_FromFileName_Over59_Test
      ✓ TimeParser_FromFileName_Test
```

Figure 6.16: The tests have all completed successfully in the Unity editor

This sort of testing has become more and more important as the project has grown in complexity. The testing of how the game works takes more and more time as the scope increases, so ensuring the basics work is extremely important. This will be even more important when the project moves onward.

6.3 Additional Features

6.3.1 Procedural Modeling Updated to Python 3

Python 2 was used when the procedural modeling software was written, when AutoDesk Maya only supported Python 2 and not Python 3. Since the code had been written, Maya has received support for Python 3, and the language Python 2 is deprecated and no longer recommended to use. [41] Therefore the procedural modeling code was updated to work with Python 3, and since it already followed most of the conventions used in Python 3 it did not require much work.

The most significant changes from Python 2 to Python 3 was how print and comparisons works. In Python 2 when trying to compare two variables with different datatypes, it would try to automatically type cast it. In Python 3 that is longer the case, and most fixes was adding implicit casting to required variables.

```
if l2 < 1000:
  allowedDeficiency = allowedDeficiency/2
absLen = abs(l1-l2)
```

Listing 6.19: Previous Python 2 code, where l2 is a string

```
if float(l2) < 1000:
  allowedDeficiency = allowedDeficiency/2
absLen = abs(float(l1)-float(l2))
```

Listing 6.20: Updated Python 3 code, where l2 is still a string

Another important change was Python 3 string variables being Unicode by default, where the previous code one had to look out for strings not being Unicode. Fixing these issues made the code work with the Python 3 interpreter

```
profile = tunnelTube[u'Tunnelprofil'].encode('utf8')
```

Listing 6.21: Previous Python 2 code, where *profile* is treated as a bytearray

```
profile = tunnelTube[u'Tunnelprofil']
```

Listing 6.22: Updated Python 3 code, where it is treated as unicode

Chapter 7

Evaluation

7.1 Current Limitations of *CoolEngine*

7.1.1 Simplistic Traffic

The traffic implementation is noticeably simplistic, and does not take multiple different behaviors which can occur in real life. Cars reacting to the fire, and trying to escape it by either driving past it or trying to turn around in the tunnel to escape the other end is not implemented, and in the software they will just stand still after detecting the burning vehicle. No drivers will go out of the vehicle either to try and flee on foot, which is also not realistic.

They also will not utilize the lanes. This is mainly because the lanes are based upon simple textures which may appear quite differently depending on which tunnel is loaded.

The traffic also only appears one way, this is due to the earlier discussed limitations of the tunnel models in Unity, it may be wise to reassess how the tunnels are utilized in Unity moving forward.

7.1.2 Tunnel Alignment

Curves are currently not implemented in most accessible CFDs, especially those focused on fire and smoke propagation. Currently the solution *CoolEngine* uses is an hyperrectangle, and will not be realistic depending on the curvature of the tunnel it is used in. As seen in 7.1.2, the simulation does not properly match the geometry of the tunnel, and smoke will be generated outside of it.

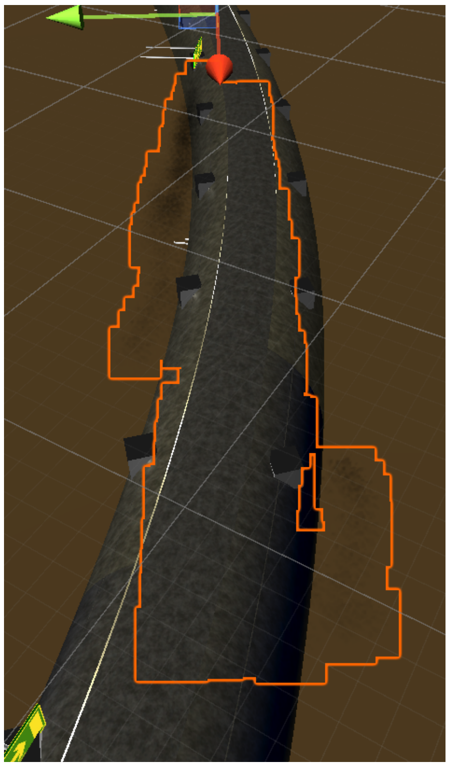


Figure 7.1: Misaligned when curved

If the tunnel is both especially circular and especially large in comparison to the simulation domain, the smoke may expose its cubic nature more clearly than it will an average run, as seen in 7.1.2

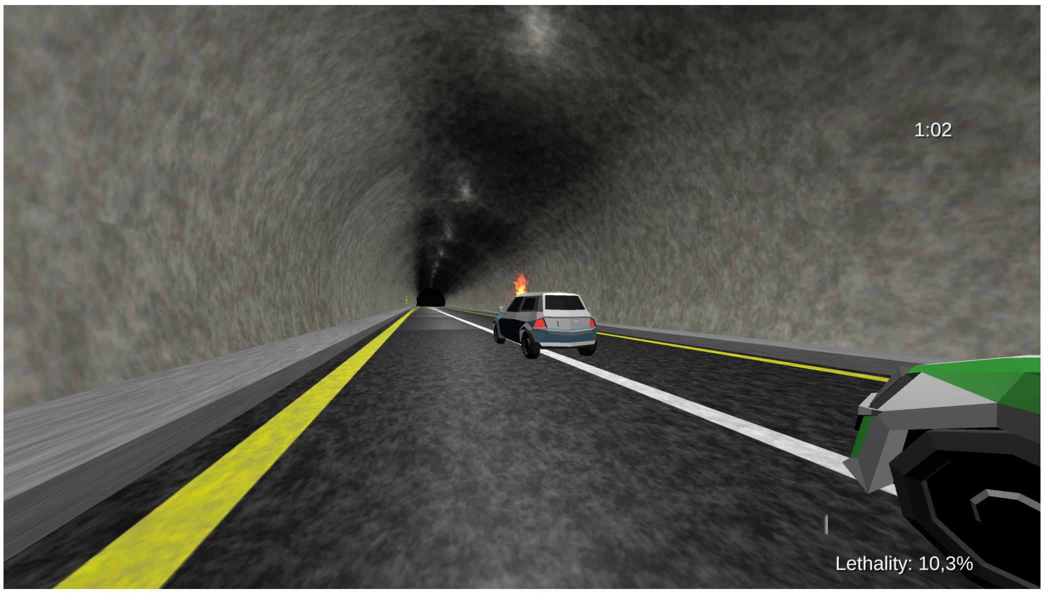


Figure 7.2: Sometimes the cubic nature of the simulation meshes poorly with the actual geometry

In addition, it does not handle vertical or steep tunnels especially well. As mentioned in the methodology, it is assumed the tunnel is aligned alongside the ground, which clearly does not work when the tunnel climbs upwards or downwards.

Overall, this is one of the biggest limitations of *CoolEngine* .

7.2 Known Issues

7.2.1 Traffic Congestion

After the software has been running for a while, the car path finding can bug out and start driving where they are not intended to drive. Somewhere along the way, the cars will not detect that there is one in front of it and stop, so it will try and continue to follow the given coordinates. This will make it drive wildly off course.

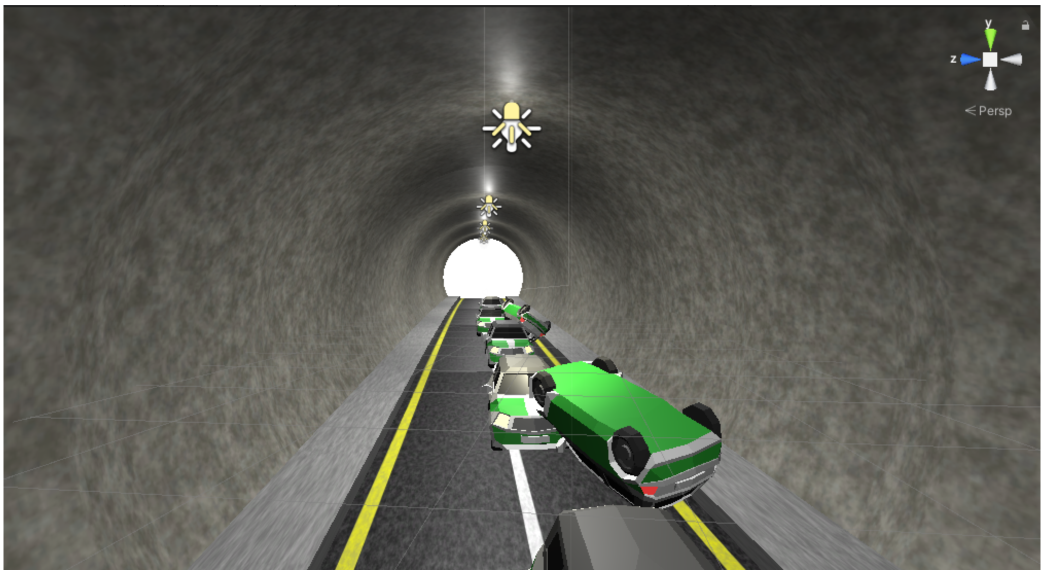


Figure 7.3: Traffic overflow

Some tunnels, especially more vertical ones, are more prone for this issue to arise. Some tunnels of this vertical nature barely have functioning traffic at all, as the implementation does not take vertically into account.

Proposed Solution

One possible fix for the more horizontal tunnels where this issue appears is to check for how many cars are nearby, and wait until they either have moved or just wait indefinitely based on the nature of the current traffic implementation

7.2.2 Traffic Rotation in Different Quadrants

Traffic rotation does not seem to work when the tunnel is in certain quadrants. In the third and fourth quadrants the cars are rotated properly and drives along the given path. In the first and second quadrants it does not seem to work, and the cars are facing the exit for the tunnel instead. It will still cast a ray in front of it and detect it as a wall and will therefore not move.

Examples of provided tunnels that work with traffic and are in the right quadrants are Bergeland and Kleivene Nordgående.

Proposed Solution

A proposed solution is to force every tunnel to be in the same quadrant, to make the math consistent regardless of which tunnel has been generated. Compared to now with the apparent randomness.

7.2.3 Tunnel Modeling

Scaling

The scaling of the tunnels does not always match with real life judging by the size of the player, cars and road signs in different tunnels. There also seems to be a discrepancy between the scaling between tunnels and tunnel tubes. It is unclear if this because of the NVDB dataset, the textures it uses throwing of the feel, the way Unity utilizes the tunnel, or if the procedural modeling is approximating too much.

Approximation

Bergelandstunnelen is a tunnel that was used a great deal for testing. One phenomenon which was noticed was that even though the tunnel inside the software has a round shape across the whole tunnel, in reality it seems to go from the more circular to more rectangular as seen in 7.2.3 and 7.2.3.

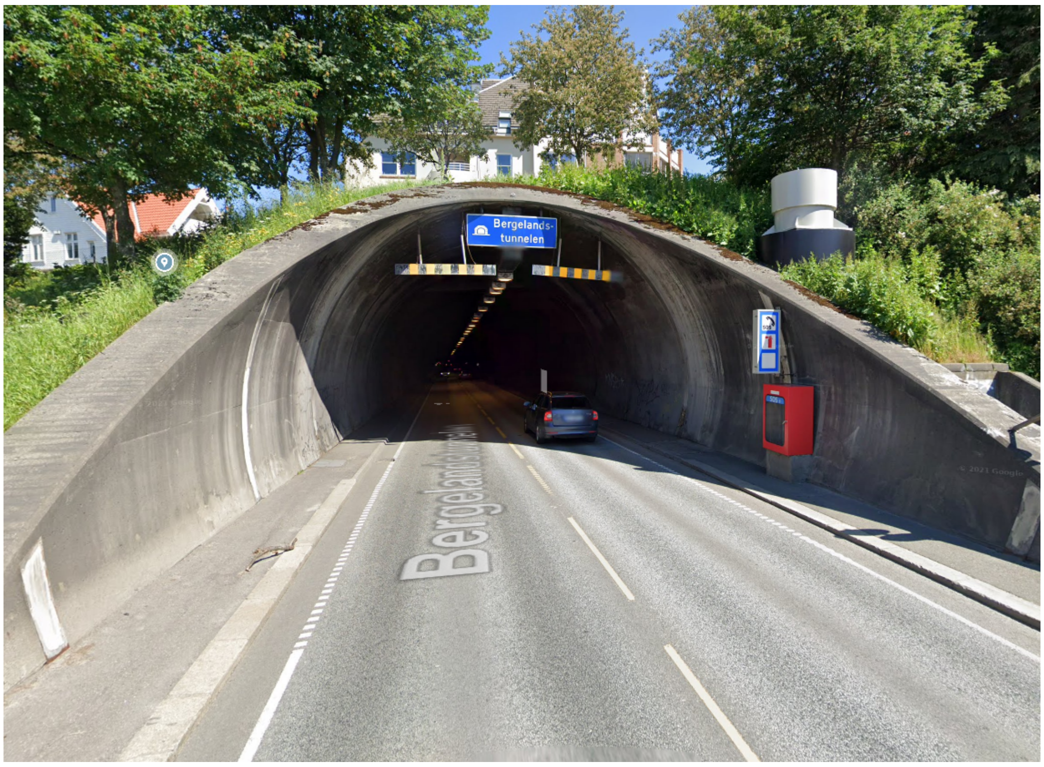


Figure 7.4: Bergeland circular part [42]

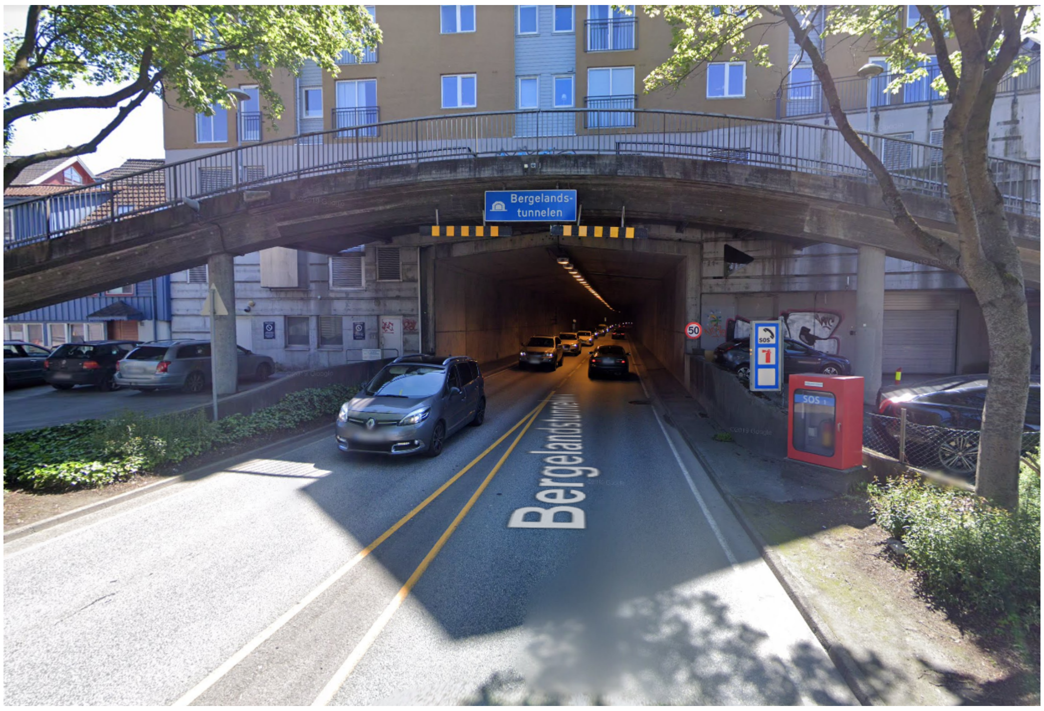
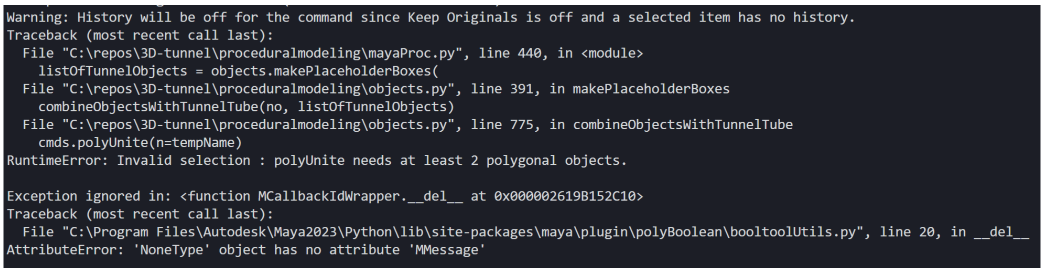


Figure 7.5: Bergeland rectangular part [43]

Certain Tunnels Not Generating

When generating certain tunnels, it will sometimes inexplicably fail. In the image 7.2.3, it displays an error where it seems to be missing polygonal objects. This can have a multitude of different reasons, like the data set being incomplete from NVDBs side or the procedural modeling making an assumption upon generation which does not apply to all tunnels.



```
Warning: History will be off for the command since Keep Originals is off and a selected item has no history.
Traceback (most recent call last):
  File "C:\repos\3D-tunnel\proceduralmodeling\mayaProc.py", line 440, in <module>
    listOfTunnelObjects = objects.makePlaceholderBoxes(
  File "C:\repos\3D-tunnel\proceduralmodeling\objects.py", line 391, in makePlaceholderBoxes
    combineObjectsWithTunnelTube(no, listOfTunnelObjects)
  File "C:\repos\3D-tunnel\proceduralmodeling\objects.py", line 775, in combineObjectsWithTunnelTube
    cmds.polyUnite(n=tempName)
RuntimeError: Invalid selection : polyUnite needs at least 2 polygonal objects.

Exception ignored in: <function MCallbackIdWrapper.__del__ at 0x000002619B152C10>
Traceback (most recent call last):
  File "C:\Program Files\Autodesk\Maya2023\Python\lib\site-packages\maya\plugin\polyBoolean\booltoolUtils.py", line 20, in __del__
AttributeError: 'NoneType' object has no attribute 'MMessage'
```

Figure 7.6: Error when creating the tunnel tube with ID 79611660

7.2.4 UI Scaling

The UI scaling is handled by Unity, which can behave unpredictably. Specifically the back button when choosing spawn location was observed to have unexpected behavior when the aspect ratio or resolution is exceedingly different from the editor resolution and aspect ratio.

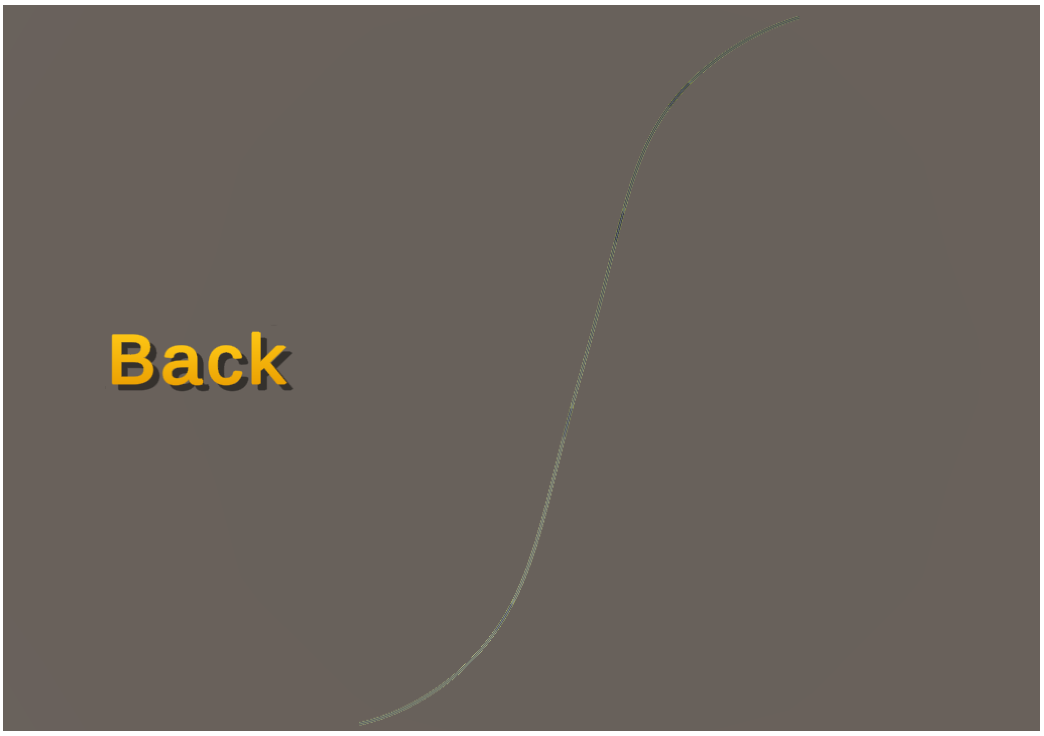


Figure 7.7: Back button position with 3:2 aspect ratio, it was supposed to be in the corner, but behaved unpredictably.

Chapter 8

Conclusion

The software has successfully been amended to include a much more realistic fire and smoke propagation based upon data from FDS.

Although there are several unique limitations and challenges with creating a software like this, like discussed in the previous chapter, completely linked with a simulation software; it also works really well all in all when all is considered. This field is something the authors hope will be explored much more deeply in the future, as it seems to be an unexplored field.

8.1 Research Questions Answers

It is prudent to review what has been learned in regards to the initially posed research questions

8.1.1 RQ1: Are There Any CFDs Suitable for Accessible Fire Simulation?

There are multiple CFDs which are both suited for fire simulation and accessible to everyone. The chief CFDs are definitely the ones from NIST: FDS and CFAST. They are lightweight and allow for complex scenarios at the same time. Other promising ones to keep an eye out for are OpenFoam and Simscale.

8.1.2 RQ2: Is It Possible to Utilize the Results From a CFD With an Environment Like Unity?

It is certainly possible to do this, as has been demonstrated in this thesis. There are several ways to do it when one considers all the different CFDs which may be able to do it, and concrete CFDs like FDS which has several of ways to expose their simulation information, which is discussed more in the appendix chapter B.

8.1.3 RQ3: How Far Can the Realism for Fire-Scenarios Be Taken with Current Technology?

It is somewhat difficult to quantify how far realism can be taken, it is nevertheless worth pointing out the findings. The realism can be massively improved, like in this task, when the propagation of smoke specifically is improved. This, however, comes at costs like interaction and the smoke rendering in real time. Therefore when the smoke moves and propagates more realistically, other aspects have to be sacrificed. There should be focus on better computations, like cloud systems outsourcing the computations, as an example to counteract this.

8.2 Completion of Goals

Of the original goals outlined in the beginning of the thesis and project, only one was not implemented: The implementation of the VR. This is because the implementation of FDS and the *CoolEngine* -pipeline was more complex than initially thought, leaving little time for such to flourish.

8.3 Potential Future Work

A great deal of future work was discussed back in the chapter about the architecture, so most of what was discussed there is relevant here. Some additional points which are not necessarily part of the design can be further improved.

8.3.1 Improvement to FDS-model

Although the choice to make the simulation as simple as possible for the sake of performance and simplicity was an important one in the project; now that it is proven that FDS may work as a baseline for the software, it may benefit from improving the simulation further. Some concrete points that can be explored further are:

- The addition of vents and fans in the simulation
- More sophisticated meshes, like curves or dynamic width according to tunnel-width
- Support for different increases in height, with a gravity vector
- More cells if better hardware is acquired
- More consideration into the chemical make-up of cars and smoke

8.3.2 NVDB Update

There were encountered some problems with the tunnels themselves in this project, some if it *may* be due to it still depending on the old and deprecated NVDBv2 when the NVDBv3 is the current and recommended version. [44] The Python script which creates the tunnel was updated from Python2 to Python3, but it may really lift the project if NVDBv3 is used.

8.3.3 Implement VR

Implementing VR was originally a goal of this software, so leaving without this done is somewhat disappointing from the authors' perspective. This should therefore be prioritized, to complete the initial vision.

8.3.4 Improving the *CoolEngine* - Pipeline Structure

CoolEngine achieves a multitude of things; but it is plagued by being somewhat of a naive solution. The two main offenders are:

- The way the Plot3D data is loaded and stored in Unity, mainly how it is all loaded at the start of the runtime instead of gradually.
- The way the data is represented in Unity, the main offender here being the particlesystem

These could probably both be improved if needed, as they are both sort of naive solutions that have concerned themselves more with how to get it to work rather than how to optimize it.

References

[1] Live Oftedahl, *Eksplosjonen som rystet en hel by: 20 år siden smellet i bragernes-tunnelen*, [Online]. Available: https://ambulanseforum.no/artikler/eksplosjonen-som-rystet-en-hel-by-20-ar-siden-smellet-i-bragernes-tunnelen/, (accessed: Apr. 25, 2022), 2019.

[2] Oddvar Karmo, *Safety security in tunnel management*, Guest Lecture, The University of Stavanger, Jan. 2022.

[3] G. D. Jenssen, J. Skjermo, Å. Snilstveit, P. Arnesen, H. Frantzich, and D. Nilsson, "Simulering av evakuering i tunnel", SINTEF, PO-box: 4760 Torgarden, 7465 Trondheim, Norway, Tech. Rep. ISBN: 978-82-14-06915-0, 2018.

[4] Simscale, *What is cfd / computational fluid dynamics?*, [Online]. Available: https://www.simscale.com/docs/simwiki/cfd-computational-fluid-dynamics/what-is-cfd-computational-fluid-dynamics/, (accessed: May 10, 2022), 2021.

[5] Thunderhead Engineering, *Fire dynamics and smoke control*, [Online]. Available: https://www.thunderheadeng.com/pyrosim/, (accessed: Apr. 25, 2022), 2019.

[6] NIST, *Cfast*, [Online]. Available: https://www.nist.gov/services-resources/software/cfast, (accessed: May 12, 2022), 2022.

[7] ——, *Smokeview (smv)*, [Online]. Available: https://github.com/firemodels/smv, (accessed: May 14, 2022).

[8] Glenn Forney, *Smokeview, a tool for visualizing fire dynamics simulation data volume i: User's guide*, [Online]. Available: https://pages.nist.gov/fds-smv/manuals.html, (accessed: Feb. 14, 2022), 2021.

[9] Pamela P. Walatka, Pieter G. Buning, Larry Pierce, Patricia A. Elson, *Plot3d user's manual*, [Online]. Available: `https://ntrs.nasa.gov/citations/19900013774`, (accessed: Feb. 15, 2022), 1990.

[10] Fortran, *High-performance parallel programming language*, [Online]. Available: `https://fortran-lang.org/`, (accessed: Apr. 25, 2022).

[11] Agile-alliance, *What is test driven development (tdd)?*, [Online]. Available: `https://www.agilealliance.org/glossary/tdd/`, (accessed: Feb. 23, 2022), 2022.

[12] Emil Haavardtun, Audun Stjernelund Lien, Dag Hermann Valvik, *3d self-rescue game for tunnel fire*, B. thesis, The University of Stavanger, Kitty Kiellands hus, Rennebergstien 30, 4021 Stavanger, May 2021.

[13] B. K. Nohut, "Digital tunnel twin using procedurally made 3d models", M.S. thesis, The University of Stavanger, Kitty Kiellands hus, Rennebergstien 30, 4021 Stavanger, 2021.

[14] Yannic Schröer, *Kiss principle (keep it simple stupid)*, [Online]. Available: `https://code-specialist.com/code-principles/kiss-principle/`, (accessed: Apr. 25, 2022), 2020.

[15] J. Wahlqvist and P. van Hees, "Visualization of fires in virtual reality", Division of Fire Safety Engineering, Lund University, SE-221 00 Lund, Sweden, Tech. Rep. ISRN: LUTVDG/TVBB-3222-SE, 2018.

[16] Official Gexcon Account, *Flacs vr-safety*, [Online]. Available: `https://www.youtube.com/watch?v=vhaLZnJPlP0`, (accessed: Apr. 25, 2022), 2016.

[17] W. J. Bong, "Limitations of zone models and cfd models for natural smoke filling in large spaces", M.S. thesis, University of Canterbury, Private Bag 4800 Christchurch, New Zealand, 2011.

[18] Richard D. Peacock, Paul A. Reneke, Glenn P. Forney, *Cfast – consolidated model of fire growth and smoke transport (version 6) user's guide*, NIST, 100 Bureau Drive Gaithersburg, MD 20899, 2013.

[19] Richard Peacock, *Version history*, [Online]. Available: `https://github.com/firemodels/cfast/wiki/Version-History`, (accessed: Jan. 24, 2022), 2017.

[20] Kevin McGrattan, Simo Hostikka, Jason Floyd, Randall McDermott, Marcos Vanella, *Nist special publication 1019 sixth edition fire dynamics simulator user's guide*, NIST, 100 Bureau Drive Gaithersburg, MD 20899, 2021.

[21] OpenFoam, *Openfoam*, [Online]. Available: `https://www.openfoam.com/`, (accessed: Jan. 24, 2022), 2022.

[22] ——, *Firefoam*, [Online]. Available: `https://openfoamwiki.net/index.php/FireFoam`, (accessed: Jan. 24, 2022), 2021.

[23] Gexcon, *Flacs-cfd*, [Online]. Available: `https://www.gexcon.com/products-services/flacs-software/`, (accessed: Jan. 24, 2022), 2022.

[24] DNV, *Kameleon fireex*, [Online]. Available: `https://www.dnv.com/services/fire-simulation-software-cfd-simulation-kameleon-fireex-kfx-110598`, (accessed: Jan. 24, 2022), 2022.

[25] FSEG, *Fseg introduction*, [Online]. Available: `https://fseg.gre.ac.uk/fire/index.html`, (accessed: Jan. 24, 2022), 2003.

[26] ——, *Smartfire introduction*, [Online]. Available: `https://fseg.gre.ac.uk/smartfire/index.html`, (accessed: Jan. 24, 2022), 2003.

[27] Simscale, *Simulation software reinvented for the web*, [Online]. Available: `https://www.simscale.com/`, (accessed: May 10, 2022).

[28] Branz, *B-risk: Design fire tool*, [Online]. Available: `https://www.branz.co.nz/fire-safety-design/b-risk/`, (accessed: Jan. 24, 2022), 2022.

[29] COMSOL, *Comsol heat transfer module*, [Online]. Available: `https://www.comsol.com/heat-transfer-module`, (accessed: Jan. 25, 2022), 2022.

[30] AVL, *The leader in powertrain cfd: Avl fire*, [Online]. Available: `https://www.avl.com/fire/`, (accessed: Jan. 25, 2022), 2022.

[31] US Forest Service, *Behaveplus fire modelling system*, [Online]. Available: `https://www.frames.gov/behaveplus/home`, (accessed: Jan. 25, 2022).

[32] Thunderhead Engineering, *Agent based evacuation simulation*, [Online]. Available: `https://www.thunderheadeng.com/pathfinder/`, (accessed: Mar. 17, 2022), 2019.

[33] J. Ltd., *Diagrams.net*, [Online]. Available: `https://www.diagrams.net`, (accessed: May 11, 2022).

[34] *Coolengine-realistic-fire*, [Online]. Available: `https://github.com/TunnelSafety/3D-tunnel/tree/coolengine-realistic-fire`, (accessed: May 10, 2022).

[35] HSE, *Methods of approximation and determination of human vulnerability for offshore major accident hazard assessment*, [Online]. Available: https://www.hse.gov.uk/foi/internalops/hid_circs/technical_osd/spc_tech_osd_30/spctecosd30.pdf, (accessed: Apr. 30, 2022).

[36] GeoGebra, *Geogebra for teaching and learning math*, [Online]. Available: https://www.geogebra.org/, (accessed: May 11, 2022).

[37] R. Pula, F. I. Khan, B. Veitch, and P. R. Amyotte, "Revised fire consequence models for offshore quantitative risk assessment", *Journal of Loss Prevention in the Process Industries*, vol. 18, no. 4, pp. 443–454, 2005, ISSN: 0950-4230. DOI: https://doi.org/10.1016/j.jlp.2005.07.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950423005001269.

[38] H. Ingason, Y. Li, and A. Lönnermark, *Tunnel Fire Dynamics.* Jan. 2015, pp. 160, 162, ISBN: 978-1-4939-2198-0. DOI: 10.1007/978-1-4939-2199-7.

[39] D. Lacroix, "New french recommendations for fire ventilation in road tunnels", in *9th International Conference on Aerodynamics and Ventilation of Vehicle Tunnels*, Aosta Valley, Italy, Oct. 1997.

[40] Unity, *Vector3.movetowards*, [Online]. Available: https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html, (accessed: May 14, 2022), 2022.

[41] Python Software Foundation, *Sunsetting python 2*, [Online]. Available: https://www.python.org/doc/sunset-python-2/, (accessed: May 12, 2022), 2020.

[42] Google, *Google maps streetview*, [Online]. Available: https://www.google.com/maps/@58.9662384,5.7350116,3a,75y,82.24h,88.06t/data=!3m6!1e1!3m4!1sB8ygMujrRaO8OI5iSYMa3Q!2e0!7i16384!8i8192, (accessed: May 10, 2022).

[43] ——, *Google maps streetview*, [Online]. Available: https://www.google.com/maps/@58.9706052,5.7430649,3a,75y,177.8h,88.71t/data=!3m6!1e1!3m4!1sxd8Y6P8sK36o-VmsqfYq3A!2e0!7i16384!8i8192, (accessed: May 10, 2022).

[44] Statens vegvesen, *Api-dokumentasjon*, [Online]. Available: https://api.vegdata.no/, (accessed: May 14, 2022).

[45] NIST, *Fds*, [Online]. Available: https://github.com/firemodels/fds, (accessed: May 12, 2022), 2022.

[46] ——, *Fds-smv*, [Online]. Available: `https://pages.nist.gov/fds-smv/downloads.html`, (accessed: May 14, 2022).

Appendices

Appendix A

Running the project

A.1 Procedural Modeling

A.1.1 Requirements

Applications

- AutoDesk Maya 2022 or later
- Python 3

Python Dependencies for Procedural Modeling

- Requests
- Colorama
- Maya
- PyMel
- SciPy

A.1.2 Installation of Python Dependencies

To automatically install all needed python dependencies, run `pip3 install -r requirements.txt` in the procedural-modeling directory.

A.1.3 Configuration

The path for AutoDesk Maya has to be set manually for procedural modeling to work, especially since the path is different depending on the year version. To set it, navigate to *procedural-modeling/config.py* and set the *MAYA_ PATH* to the appropriate path.

```
MAYA_PATH = 'D:\\Autodesk\\Maya2022\\bin\\mayapy.exe'
```

Listing A.1: config.py

As long as the path specified directs to mayapy.exe as the listing shows, it should be fine.

A.1.4 Usage

1. To start the procedural modeling software, run the command `py gui.py` or runApp.bat if Windows.
2. Acquire the id (example: 78728413) from NVDB from the following URLs:
   - For tunnels: https://nvdbapiles-v2.atlas.vegvesen.no/vegobjekter/581
   - For tunnel tubes: https://nvdbapiles-v2.atlas.vegvesen.no/vegobjekter/67

   Note: The URLs will provide a list of tunnels, which will be limited by the amount of API calls allowed by NVDB. The atlas for NVDB API v2 is no longer available. [44] The IDs between API v2 and v3 with limited testing seems to be the same, so going on the newer atlas might have some success. The newer atlas is available at https://vegkart.atlas.vegvesen.no.
3. Insert the ID, choose the correct type and then click generate.

4. The 3D-model file will appear as an .fbx file under proceduralmodeling/tunnels, to import it to *CoolEngine* move it to *Assets/Resources/Tunnels* in the *project/editor* directory.

5. Create a new build as explained in A.3.2.

6. The new build with the new tunnel(s) should now, after following the necessary steps in A.3.2, be ready to use.

A.2 Compiling FDS2ASCII

A.2.1 Requirements

- Intel MPI Library or OpenMPI
- Intel Fortran compiler or GNU Fortran compiler
- Make Software

Note: Based on experience, it is recommended to use the Intel library and compiler for Windows, and OpenMPI and GNU Fortran compiler for Linux.

A.2.2 Compiling

First of all, make sure that all the wanted changes in `fds2ascii.f90` have been made before attempting a compile.

In the FDS repository, navigate to *Utilities/fds2ascii* and to the appropriate directory based on compiler and operating system. [45] It will contain a make file, and directories for different operating systems with different compilers containing the appropriate operating system script file to build FDS2ASCII.

A.3 Unity Project

A.3.1 Requirements

Supported Operating Systems

- Windows 10
- Windows 11

Application Requirements

- Unity 2020.3.27f1
- FDS (6.7.7) [46]

A.3.2 Setup

Creating Unity Build

To manually build the project, open the project with the Unity Hub. Go to File -> Build Settings -> Build, and a popup will appear asking where to place the build.

Note: When cloning the repository, make sure to pull everything. This is done by first doing `git pull` and then `git lfs pull`

Afterwards, to complete the build additional assets needs to be moved manually. The directories and its contents Assets/CFD and Assets/Resources needs to be moved to the Assets folder in the build location.

A.3.3 Running Software

To start the project, simply run the executable included in the build folder.

Appendix B

Additional FDS parameters

There was spent a great number of hours understanding the jargon of the FDS-syntax in this project, so much so that some information learned did not fit into the final product. This section will therefore spend some time detailing other parameters learned that may be of use in other similar projects.

B.1 Gravity Vector

```
&MISC GVEC=-1.70,0.0,-9.65 /
```

The GVEC parameter-line specifies the gravity vector which will be used in the simulation. This is set to

$$(0, 0, -9.81)$$

by default, which is what the gravity vector is on a flat surface at ground level if the z-axis is orthogonal to the ground-plane.

This parameter can be used to model a tunnel which is not on the ground level, as different gravity vectors lead to different flows of smoke akin to tunnels with different heights.

B.2 Slice Files

```
&SLCF QUANTITY='TEMPERATURE', VECTOR=.TRUE., PBY=0. /
&SLCF QUANTITY='H', CELL_CENTERED=.TRUE., PBY=0. /
```

The &SLCF parameter line specifies output of slice files (.sf). These files can be in the form of lines or planes, and an be used to aggregate information into these mathematical models instead of illustrating each point like done in this project with Plot3D .

This could be used in the future for more sophisticated interpretations of the data into *CoolEngine* . An example would be to output the heat radiation as a plane, and use this to calculate the heat lethality based upon where on the plane the user is; opting for a model which effectively only values how close the user is to the heat source and not which height it has.

But this requires that Unity supports .sf-files or that the fds2ascii application supports parsing this information to something usable, both of which could not be verified within the time-frame of this project.

B.3 Isosurface

```
/ &ISOF QUANTITY='soot density', VALUE(1)=0., VALUE(2)=1.,
   VALUE(3)=10. /
```

The &ISOF parameter-line specifies how a isosurface based on a certain physical value shall be created.

This means that a surface can be generated which is effectively a level field of a more complex function. An example: you specify temperature=1000 K, and the surface will be created on every point which fits this description.

This can be used for better visualizations for certain things in the future. As an isosurface may be smoother in appearance than data taken from a cell-grid like used in this project. It may, however, be difficult to use these surfaces in Unity. Further clarification should be applied on this.

Appendix C

Hardware Specifications

As this is a project based upon the game-engine Unity, and simulation of realistic fire, the computational resources needed may be fairly heavy. Underneath are some specs from computers used in this project, which provides a rough idea of the requirements needed to run *CoolEngine* from a hardware-perspective.

Table C.1: Specifications of computers used in the project

Model/Name	CPU	GPU	Memory	Laptop	Recommend?
Surface Pro 6	Intel i7-8650U @ 4.2GHz	Intel UHD Graphics 620	16GB 1867MHz	Yes	No
Acer Nitro 5	Intel i7-9750H @ 2.6GHz	Nvidia GTX 1660 Ti	16GB 2667MHz	Yes	Sort of
Desktop PC	AMD RYZEN 7 3700X	Nvidia RTX 2080	16GB 3200MHz	No	Yes

Note that the Acer-PC runs at a acceptable frame-rate for the most part, but the performance is not as great as on Desktop. Hardware similar to the components the Acer has is therefore to be viewed as a sort of minimum requirement.

Appendix D

Software User Guide

D.1 Generating Scenario

To generate a scenario, navigate to **Scenarios** from the main menu. Here are some tips for the scenarios:

- Set the same preferably to something unique to identify.
- Set the soot yield to your liking. A general recommendation for soot yield is that it should not exceed 0.5kg/kg, as the chosen fuel propane does not support much more yield than that.
- Set the time to something desirable. The time selected is recommended to not be over 6 minutes, since both the wait times for the generation and the wait times for the loading of the simulation are extremely long even with the highest specifications the group has tested with.
- If the simulation seemingly does not work, one of the values you have inputted may be invalid. You should start with some basic values, like 30 seconds duration and 0.15 soot yield which is a scenario that is confirmed to work, and then change some values gradually to see if FDS wants to accept them.

D.2 Playing the scenario

To start a scenario, head into **Start** from the main menu. Here are some additional tips for the starting of the scenario:

- Choosing Unity Rendered simulation means that you will get a basic and procedural smoke and fire based upon simple rules. This is great if you do not want to load an entire simulation, or if you want to compare simulations.
- The tunnels you can choose are based upon which you have inputted, detailed further in how to create procedural tunnels.
- The traffic levels are based upon the pseudo-random values Unity generates. Therefore even if you choose high traffic, the traffic may be low due to randomness.

D.3 In-game

Once you are in game, there are some points that should be emphasized:

- The visualization of the smoke is based upon the max value of the soot density, so the smoke you see may not be entirely accurate, it is to be viewed more as indication of there being a great deal of smoke there. This means you will always see some smoke, no matter how short the simulation is.
- The CO-lethality begins at around 10% to make it easier for the user to see the effects of CO-lethality, this does NOT mean a person would always have 10% lethality in a scenario like this.
- The software does not take into account the damage fires themselves do to a human due to time constraints, this does NOT mean the user could survive being so close to a fire as the software may imply.