Sindre B. Garvik
Haakon B. Aarstein
Eirik Hiis-Hauge

Learning in Virtual Reality

Bachelor's project in Game Programming
Supervisor: Øivind Kolloen, Rune Hjelsvold
May 2019

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Sindre B. Garvik
Haakon B. Aarstein
Eirik Hiis-Hauge

Learning in Virtual Reality

Bachelor's project in Game Programming
Supervisor: Øivind Kolloen, Rune Hjelsvold
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

NTNU
Norwegian University of
Science and Technology

Learning in Virtual Reality

Author(s)

Sindre B. Garvik
Haakon B. Aarstein
Eirik Hiis-Hauge

Bachelor in Game Programming
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

20.05.2019

Supervisor Øivind Kolloen
Rune Hjelsvold

Sammendrag av Bacheloroppgaven

Tittel:	**Læring i virtuell virkelighet**
Oppgave no.	2
Dato:	20.05.2019
Deltakere:	Sindre B. Garvik Haakon B. Aarstein Eirik Hiis-Hauge
Veiledere:	Øivind Kolloen Rune Hjelsvold
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Erik Helmås, erik.helmas@ntnu.no, 61135000
Nøkkelord:	IMT, Simulasjon, VR, Jernbane, Utdanning
Antall sider:	34
Antall vedlegg:	6
Tilgjengelighet:	Åpen

Sammendrag:

Norsk fagskole for lokomotivførere ønsker en utvidelse av et eksisterende prosjekt via en prototype. Det eksisterende prosjektet er en lokfører simulator, og utvidelsen vil gå ut på å koble sammen tog i VR (Virtual Reality).

Summary of Graduate Project

Title:	**Learning in Virtual Reality**
Project no.	2
Date:	20.05.2019
Authors:	Sindre B. Garvik Haakon B. Aarstein Eirik Hiis-Hauge
Supervisor:	Øivind Kolloen Rune Hjelsvold
Employer:	Norwegian University of Science and Technology
Contact Person:	Erik Helmås, erik.helmas@ntnu.no, 61135000
Keywords:	IMT, Simulation, VR, Railway, Education
Pages:	34
Attachments:	6
Availability:	Open

Abstract: The Norwegian Railway School seeks to expand an existing project through a prototype. The existing project is a train driver simulator, and the expansion is for coupling trains together in VR (Virtual Reality).

Preface

We would like to thank our two supervisors for all their help, and the Norwegian Railway School for giving us a interesting project to work with.

Contents

1 Project Plan

The original project plan was written in Norwegian, as such, this translation may somewhat differ from the original one. Additionally, this version of the project plan has been slightly changed to reflect the minor changes that have been done to the development goals during the project. The original project plan (in Norwegian) will be appended.

1.1 Project Description

The Norwegian Railway School seeks to expand an existing project through a prototype. The existing project is a train driver simulator, and the expansion is for coupling trains together in VR (Virtual Reality).

1.2 Project Modules

The Norwegian Railway School detailed several modules they wished for us to implement during the project:

- Coupling in VR.
- Physics on the train.
- Network capabilities connecting the existing part of the project with the new one.
- 3D models of the trains and train cars.
- Automated scenarios (has been dropped in favor of improving existing parts).

We were given quite a bit of leeway as to what had to be completed for the bachelor. The list above is ordered by priority, and the two first modules, coupling and physics, are by far the most important ones, while networking and modelling were given less priority. The automation module, as noted above, was dropped, and was something we planned to do if we had enough time to spare.

1.3 Development Environment

The project was developed for Windows PCs with Virtual Reality headsets (more specifically Oculus and Vive) in jMonkeyEngine using Java.

1.4 Background

The purpose of this project was to develop a near-life Virtual Reality experience that could substitute the existing training for coupling trains. The current training is very costly, and potentially very dangerous, and moving the training over to VR will nullify the risk and heavily reduce cost, as well as increase availability.

1.5 Development Goals

We were given an extensive list of various features they wanted us to implement under each module.
Some goals under Coupling and Physics were more closely related to another module, and was therefore moved in this version.

1.5.1 Module 1: Coupling

- Locomotive is maneuverable.
- When one train car is next to another train car, they can be coupled by lifting a physical coupling that can be tightened. Similarly, there are air hoses that can be coupled, with air valves on each train car.
- When two train cars have been coupled, they are connected and will move together.

1.5.2 Module 2: Physics

What happens to a train, a train car, or a locomotive in various situations given certain conditions:

- Train car starts rolling while on an incline without sufficient brakes active.
- What happens when a locomotive hits a train car that is standing still? Does it get pushed away? Less so if the brakes are active.
- Brakes visibly move onto the wheel while active.
- The coupling of air hoses directly affects the braking system of the train, and determines whether the train will be able to drive, start rolling, etc.
- Sound effects.

1.5.3 Module 3: Network Capabilities

- The program running on DeskSim and in VR is the same.
- Potential differences in the program should be applied through settings or automatically by detecting VR equipment.

The two simulators are synced through an internal WiFi connection.

- The simulators are in the same scenario.
- When the two simulators are in the same scenario, one should be able to perceive the other's presence.
- If the locomotive is being moved in the DeskSim, one should be able to perceive this in the VR simulator.
- There should be an avatar to represent the person in VR that is visible in the DeskSim. The avatar should look like its wearing appropriate work attire.
- Actions taken in one simulator should be reflected in the other. For example, coupling together train cars, changing tracks, or applying brakes.
- Mirrors on the locomotive will allow the train driver to observe the VR avatar moving around the train, but not while it is between train cars.

Communication between train driver and shunter (switcher), DeskSim and VR.

- Communication usually occurs through a radio (walkie-talkie). One form of simulated radio communication should be available. Either by talking into a microphone, or by sending predefined messages.
- There are a series of hand signals used for communication, this should be performed by taking specific actions in the VR simulator.
- Communication must be available for both the person in the VR simulator and the person using the DeskSim.

1.5.4 Module 4: 3D Models

Created in Blender, locomotive and train cars resemble their real world counterparts.

- 1 model of a Traxx locomotive.
- 1 model of a shunting locomotive.
- 1 model of a 6-axle container car (jointed).
- 4 models of containers in different colours. Should be placed on previously mentioned train car.
- 1 tank car.
- 1 gravel car.
- 1 closed freight car with sliding doors.

- Components should resemble their real world counterparts.
- Readable labels must be included.
- The VR avatar should have hands that function with Oculus VR controllers.

1.5.5 Module 5: Automated Scenarios

This module has been dropped!
The program runs in a scenario where the program itself controls the locomotive and gives messages to the VR person (the user) using the same kind of messaging system as previously mentioned.

- The Locomotive should move based on messages given by the VR person, such as "move back", "move forward", "stop", "slow down", "30 meters until impact", "15 meters until impact".
- Voice recognition is likely too complicated for this project, so communication has to be done using predefined messages.

The program runs in a scenario where the program controls the VR person and gives messages to the train driver (the user).

- Reverse of the previously described mode.
- The program gives messages to the DeskSim so the user knows what to do.
- The shunting operation must be defined before starting the simulator.

1.6 Roles

Sindre has been chosen to be the group leader.
Haakon is responsible for 3D modelling, and texturing.

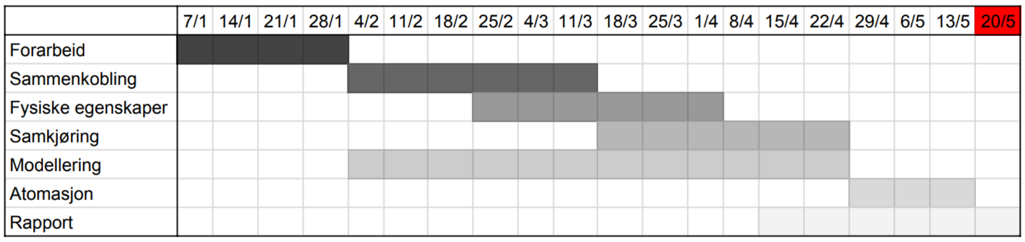
1.7 Routines and Rules

The routines and rules for this project were very relaxed. We had no set times we had to work and we were free to choose when to work. The only rule for work time was how much we should work and we chose to set 30 hours a week as our goal.

Another rule we had was that a person that did not work enough would be punished by having to bake a cake for the rest, and also if it did not get better two weeks after this warning the issue would be brought to our supervisor, and then if it still was not better then the person would have to leave the group.

When it comes to rules about pushing and committing we had a rule that said we should not push something to master that was not working, and we had a rule that said we should not commit something that was not completed. Our rule for commit messages was to follow a set of rules we found. For commenting we had a rule that we should comment often and follow a set of rules we found.

1.8 Development Model



	7/1	14/1	21/1	28/1	4/2	11/2	18/2	25/2	4/3	11/3	18/3	25/3	1/4	8/4	15/4	22/4	29/4	6/5	13/5	20/5
Forarbeid	■	■	■	■																
Sammenkobling					■	■	■	■	■	■										
Fysiske egenskaper								■	■	■	■	■	■							
Samkjøring											■	■	■	■	■	■				
Modellering					■	■	■	■	■	■	■	■	■	■	■	■				
Atomasjon																	■	■	■	
Rapport															■	■	■	■	■	■

This diagram could make it seem like we worked using a waterfall model. This is not entirely correct, as the diagram only shows which modules we prioritized first. We have divided the project based on these modules, and we used Trello to organize and plan tasks under each module, and at what stage of development they are in.
This means we divided up tasks and placing them based on whether they're on the backlog, in development, under testing, or completed.
Therefore our development model was more similar to an incremental one.

1.9 Meetings and Decision-making

We planned to meet almost weekly with our supervisor at the university, and similarly with our employer through online meetings.
We wrote a status report near the end of module 1, around 4th of March, and one around the end of module 2, around 8th of April. If modules weren't completed in time, or larger amounts of work remain, we planned to cut modules of lower priority.

1.10 Documentation and Source Control

The project will be available through a git solution. The project will therefore be logged through commits, which we then used to write the report. We're thinking of making a class-diagram to document the classes and files we have made during the project to better document what we have been working on, and to give our employer a better overview of the project's structure.

We will be following a git commit standard similar to this. Which first specifies what kind of change has been made; if it's a bug fix, a new feature, or something else.
There should also be a short description of what has been done. It would be preferable if a relevant scope is described, but this is not required. It is also highly preferred if there's a longer description detailing the changes under the initial message, but this is also not a requirement.

1.11 Risk Analysis

Security is of little relevance in this project. We're producing a prototype in an existing project for a product that will be used internally by our employer. There are no personal data in the project, one achieves nothing by crashing the program, and there is nothing to be gained by cheating in the simulator.

The networking component is really the only potential security risk, since it involves a networked communication between two machines.
Because of this, the only security we have to prioritize during development is the networking itself, which will include sanitation of input.

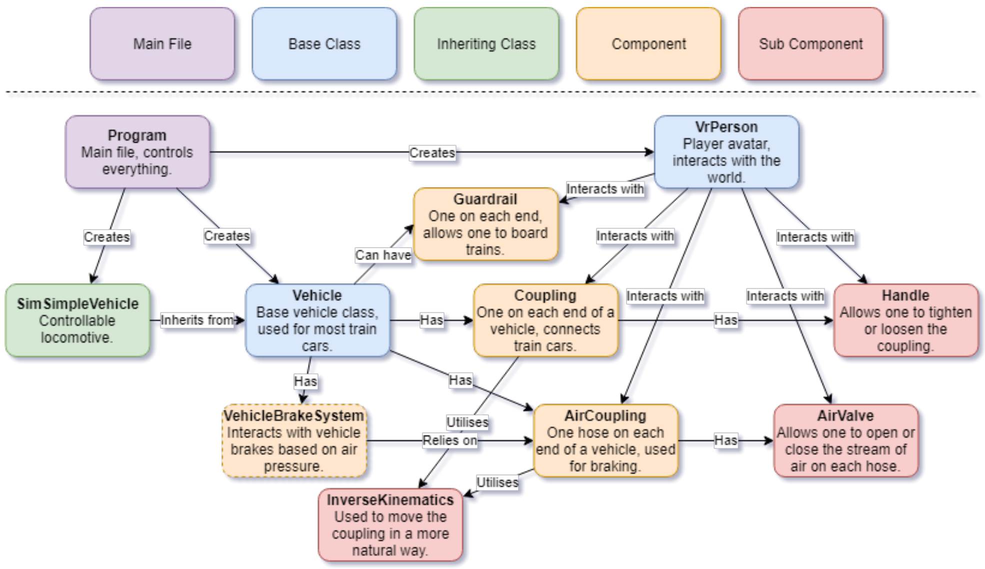
If someone wished to steal the project, one would have to either steal the prototype through our private git repository, or copy over the files from the computers of our employer.

Illness is a risk that is difficult to foresee, but we see it as a small risk, and in the event of one of our group members being absent the remaining group members will have to work harder for a bit.

2 Program Overview

2.1 Program Structure

The project itself is available at https://bitbucket.org/Haaki/digiskift/src/master/, but it is in a private repository.
Due to the sheer size of the original project and the amount of files not related to our expansion on it, the class diagram below only shows classes we have implemented ourselves or done substantial changes to.



Main File
Base Class
Inheriting Class
Component
Sub Component
Program
Main file, controls everything.
Creates
VrPerson
Player avatar, interacts with the world.
Interacts with
Guardrail
One on each end, allows one to board trains.
Creates
Creates
Can have
Interacts with
Interacts with
SimSimpleVehicle
Controllable locomotive.
Inherits from
Vehicle
Base vehicle class, used for most train cars.
Has
Coupling
One on each end of a vehicle, connects train cars.
Interacts with
Interacts with
Has
Handle
Allows one to tighten or loosen the coupling.
Has
Has
VehicleBrakeSystem
Interacts with vehicle brakes based on air pressure.
Utilises
Relies on
AirCoupling
One hose on each end of a vehicle, used for braking.
Has
AirValve
Allows one to open or close the stream of air on each hose.
Utilises
InverseKinematics
Used to move the coupling in a more natural way.

2.1.1 Program

The main file of the program, which is responsible for initiating, running, and stopping the program. This class does a lot of different things, and was there when we started working on the project. We have worked little in the main file itself, and most of the work done here has been moving various functions and variables out of the file and into VrPerson.

2.1.2 VrPerson

This is the file we have primarily been working with. This class is a movable first-person character that contains almost everything directly related to VR and interacting with objects outside of the original train simulator.

2.1.3 Vehicle

This is another class we have been working a lot with. The vehicle class contains the basis for a train or train car, and contains a data file that determines everything from model component placements to the size of its box collider. It also contains the function for colliding with other vehicles, and several important components like the Coupling class.

2.1.4 SimSimpleVehicle

This class inherits from the Vehicle class. Its primary function is to add a user-interface to the vehicle, so that it can be used as a controllable locomotive.

2.1.5 Coupling

The coupling system has been a vital part of the project. This class contains the logic needed for the Vehicle class to know whether it has been connected or not, and have the functions necessary for the VrPerson to interact with it. It also contains the Handle class.

2.1.6 Handle

Allows the VrPerson to tighten and loosen the middle section of the coupling.

2.1.7 AirCoupling

This class functions much like the Coupling class, the major differences being its Inverse Kinematics and having the AirValve class instead of the Handle.

2.1.8 AirValve

The air valve allows the VrPerson to turn on and off the air flow to the VehicleBrakeSystem.

2.1.9 InverseKinematics

This system allows one to move a segmented limb, such as a chain, using inverse kinematics. This is used for the Coupling and AirCoupling, so that they behave in a semi-realistic fashion while in VR.

2.1.10 VehicleBrakeSystem

This system was never completed, but was designed to control the vehicle's brakes based on air pressure, which could be changed based on many different factors. This is mainly controlled by whether the vehicles air valves are open, and whether its air hoses were coupled with another train car.

2.1.11 Guardrail

Allows the VrPerson to board and disembark vehicles that have guardrails. Currently, only the shunting locomotive has this feature.

2.2 VrPerson Interactions

As shown in the diagram, the VrPerson is able to interact with a variety of objects. The VrPerson does so by checking its hands' proximity to the objects it can interact with, and whether it's gripping or not.

The code below shows the basic way to interact with an object. If the conditions mentioned above are met, it simply calls a function on the object itself, and leaves all the details to that object.

```
// Interact with objects through grip
if (grip < 0.5f && prevGrip[index] > 0.5f)
{
    for (Vehicle veh : stVehicleList)
    {
        for (int i = 0; i < 2; i++)
        {
            // Handles
            if
                (veh.getCoupling(i).getHandlePos().distance(rb.getPhysicsLocation())
                < holdRange)
                veh.getCoupling(i).interactMidOffset();

            // Boarding with Guardrails
            if (veh.getGuardrail(i) != null)
                if
                    (veh.getGuardrail(i).getGuardrailPos().distance(rb.getPhysicsLocation())
                    < holdRange)
                    veh.getGuardrail(i).click();

            // Air Valves
            if
                (veh.getAirCoupling(i).getAirValvePivot().getWorldTranslation().distance(
                rb.getPhysicsLocation()) < holdRange)
                veh.getAirCoupling(i).airValveInteract();
        }
    }
}
prevGrip[index] = grip;
```

Couplings work similarly, but require that the person is not already holding anything in the hand it is using to interact with. The code below is the logic used to determine whether a coupling should be held or not.

```
//  Grabbing couplings
if (grip > 0.5f && !holding[index])
{
    //  Coupling
    outerloop:
    for (Vehicle veh : stVehicleList)
    {
        for (int i = 0; i < 2; i++)
        {
            if (veh.getCoupling(i).heldBy() == null)
            {
                // Couplings
                if
                    (veh.getCoupling(i).getEndPivot().getWorldTranslation().distance(
                    rb.getPhysicsLocation()) < holdRange)
                {
                    veh.getCoupling(i).hold(geo);
                    holding[index] = true;
                    break outerloop;
                }
                else if (veh.getCoupling(i).getState() !=
                    Coupling.CouplingState.Connected)
                {
                    veh.getCoupling(i).setState(Coupling.CouplingState.Loose);
                }
            }
        }
    }
}
```

This code shows how the coupling itself happens. When a held coupling is released near another coupling, it will attempt to connect to that coupling.

```
// Releasing and connecting couplings
if (grip <= 0.5f && holding[index])
{
    for (Vehicle veh : stVehicleList)
    {
        for (Vehicle veh2 : stVehicleList)
        {
            if (veh == veh2) // Don't check vehicle with itself
                continue;

            for (int i = 0; i < 2; i++)
            {
                if (veh.getCoupling(i).heldBy() == geo)
                {
                    // Check Coupling on other vehicle
                    for (int j = 0; j < 2; j++)
                    {
                        if
                            (veh.getCoupling(i).getEndPivot().getWorldTranslation().distance(
                               veh2.getCoupling(j).getConnectPivot().getWorldTranslation()
                               ) < holdRange)
                        {
                            // Connect
                            veh.getCoupling(i).connectTo(veh2.getCoupling(j));
                            holding[index] = false;
                        }
                    }
                    if (veh.getCoupling(i).getState() !=
                        Coupling.CouplingState.Connected)
                    {
                        veh.getCoupling(i).release();
                        holding[index] = false;
                    }
                }
            }
        }
    }
}
```

The same logic is applied for AirCoupling, but it still works a bit differently. The AirCoupling requires one to connect the ends of both couplings, rather than the end of one coupling being attached to a hook on the other.

3 Final State of the Project

3.1 Modules and Cuts

We have colour-coded the points below based on how complete or incomplete they may be. Red means it is untouched, green means it is finished, and yellow means it is incomplete.

3.1.1 Module 1: Coupling

- Locomotive is maneuverable.
- When one train car is next to another train car, they can be coupled by lifting a physical coupling that can be tightened. Similarly, there are air hoses that can be coupled, with air valves on each train car.
- When two train cars have been coupled, they are connected and will move together.

The coupling module was fully implemented, and should be working as intended.

3.1.2 Module 2: Physics

What happens to a train, a train car, or a locomotive in various situations given certain conditions:

- Train car starts rolling while on an incline without sufficient brakes active.
- What happens when a locomotive hits a train car that is standing still? Does it get pushed away? Less so if the brakes are active.
- Brakes visibly move onto the wheel while active.
- The coupling of air hoses directly affects the braking system of the train, and determines whether the train will be able to drive, start rolling, etc.
- Sound effects.

Physics have for the most part been implemented, what hasn't been completed is mostly directly related to the advanced braking system. We also did not add any more sound to the project, as we saw this as a very low priority.

3.1.3 Module 3: Network Capabilities

- The program running on DeskSim and in VR is the same.
- Potential differences in the program should be applied through settings or automatically by detecting VR equipment.

The two simulators are synced through an internal WiFi connection.

- The simulators are in the same scenario.
- When the two simulators are in the same scenario, one should be able to perceive the other's presence.
- If the locomotive is being moved in the DeskSim, one should be able to perceive this in

the VR simulator.
- There should be an avatar to represent the person in VR that is visible in the DeskSim. The avatar should look like its wearing appropriate work attire.
- Actions taken in one simulator should be reflected in the other. For example, coupling together train cars, changing tracks, or applying brakes.
- Mirrors on the locomotive will allow the train driver to observe the VR avatar moving around the train, but not while it is between train cars.

Communication between train driver and shunter (switcher), DeskSim and VR.

- Communication usually occurs through a radio (walkie-talkie). One form of simulated radio communication should be available. Either by talking into a microphone, or by sending predefined messages.
- There a series of hand signals used for communication, this should be performed by taking specific actions in the VR simulator.
- Communication must be available for both the person in the VR simulator and the person using the DeskSim.

The networking module is largely incomplete; We managed to implement a rudimentary system that could connect to another computer using a local IP address, but we had difficulties expanding this to a LAN network.
Additionally, we did not fully implement a consistent world across the networked simulators, but this was mostly due to time restraints.
We have written more about the networking under the Networking chapter.

3.1.4 Module 4: 3D Models

Created in Blender, locomotive and train cars resemble their real world counterparts.

- 1 model of a Traxx locomotive.
- 1 model of a shunting locomotive.
- 1 model of a 6-axle container car (jointed).
- 4 models of containers in different colours. Should be placed on previously mentioned train car.
- 1 tank car.
- 1 gravel car.
- 1 closed freight car with sliding doors.

- Components should resemble their real world counterparts.
- Readable labels must be included.
- The VR avatar should have hands that function with Oculus VR controllers.

We focused on completing the shunting locomotive and some basic wagons before the other models, and we have one locomotive and a wagon that is pretty much done, only missing some of the finer details and texture. The modelling has been challenging due to poor or lacking reference.

3.1.5 Module 5: Automated Scenarios

This module has been dropped!

We decided before we started on networking that we should cut automation, and we did so because we thought it would be too much work for the time we had left and too much work compared to how important the module is. We instead chose to cut this module and for the end of the project focus on perfecting modules and bug-fixing instead of adding features.

3.2 Summary

In the project plan we stated that Module 1 and 2 (Coupling and Physics) were the two most important modules, and took priority over everything else. At the end of this project those two modules are mostly complete, and the two most important models have also been completed, while networking only has the most basic implementation.

It is regrettable that we have not been able to fully complete the first four modules, but our progress was slowed due to various issues, including issues caused by the existing project itself, as well as with jMonkey.

4 jMonkeyEngine as an Engine

4.1 Overview

In this section we will discuss jMonkeyEngine, and compare it to Unity, a newer and more popular engine.
jMonkeyEngine is an open source Java engine originally developed in 2003. jMonkey's core development team stepped back from the project in 2008, and core support stopped completely in 2016.
Unity is an engine written in C++ with C# as its scripting API, originally developed in 2005. Unity is arguably the most popular game engine in the world as of this report, and is updated frequently with a major update every year.

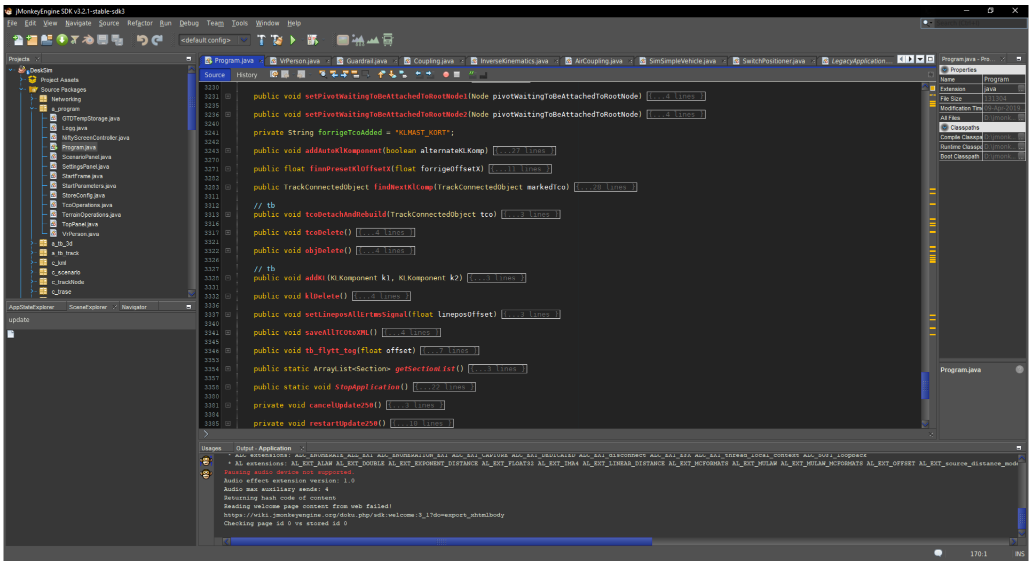
Here is an overview of a basic feature comparison between the two:

Engine:	**jMonkey**	**Unity**
Open Source	Yes	No
Scene Editor	Yes	Yes
Component System	No	Yes
Asset Viewer	No	Yes
Asset Store	No	Yes
Platforms	5	25+

Now before we continue, we feel it is important to clear up some biases. All three group members have extensive experience with Unity; We have used it for several years, and for several major projects. Regardless of how good of an engine jMonkey is, we would have been able to work better and more efficiently in Unity. With that out of the way, it must be said that we do believe that Unity is overall a far better engine for game and simulator development, and we will be exploring why in the following sections.

4.2 Editors

4.2.1 jMonkey



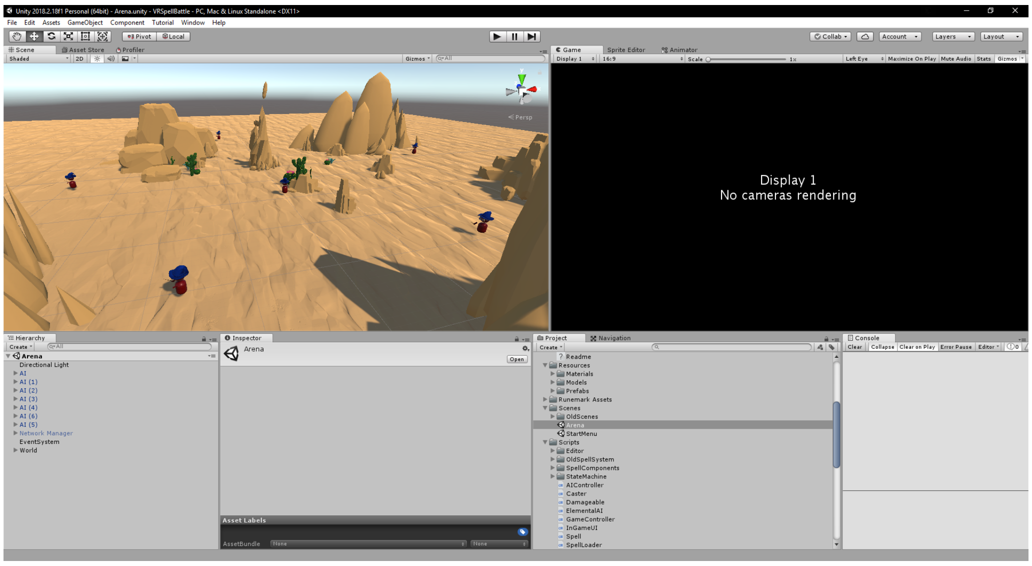
```
public void setPivotWaitingToBeAttachedToRootNode1(Node pivotWaitingToBeAttachedToRootNode) {...4 lines }

public void setPivotWaitingToBeAttachedToRootNode2(Node pivotWaitingToBeAttachedToRootNode) {...4 lines }

private String forrigeTcoAdded = "KLMAST_KORT";

public void addAutoKlKomponent(boolean alternateKLKomp) {...27 lines }

public float finnPresetKlOffsetX(float forrigeOffsetX) {...11 lines }

public TrackConnectedObject findNextKlComp(TrackConnectedObject markedTco) {...28 lines }

// tb
public void tcoDetachAndRebuild(TrackConnectedObject tco) {...3 lines }

public void tcoDelete() {...4 lines }

public void objDelete() {...4 lines }

// tb
public void addKL(KLKomponent k1, KLKomponent k2) {...3 lines }

public void klDelete() {...4 lines }

public void setLineposAllErtmsSignal(float lineposOffset) {...3 lines }

public void saveAllTCOtoXML() {...4 lines }

public void tb_flytt_tog(float offset) {...7 lines }

public static ArrayList<Section> getSectionList() {...3 lines }

public static void StopApplication() {...22 lines }

private void cancelUpdate250() {...3 lines }

private void restartUpdate250() {...10 lines }
```

```
 * ALC extensions: ALC_ENUMERATE_ALL_EXT ALC_ENUMERATION_EXT ALC_EXT_CAPTURE ALC_EXT_DEDICATED ALC_EXT_disconnect ALC_EXT_EFX ALC_EXT_thread_local_context ALC_SOFT_loopback
 * AL extensions: AL_EXT_ALAW AL_EXT_DOUBLE AL_EXT_EXPONENT_DISTANCE AL_EXT_FLOAT32 AL_EXT_IMA4 AL_EXT_LINEAR_DISTANCE AL_EXT_MCFORMATS AL_EXT_MULAW AL_EXT_MULAW_MCFORMATS AL_EXT_OFFSET AL_EXT_source_distance_mode
Pausing audio device not supported.
Audio effect extension version: 1.0
Audio max auxiliary sends: 4
Returning hash code of content
Reading welcome page content from web failed!
https://wiki.jmonkeyengine.org/doku.php/sdk:welcome:3_1?do=export_xhtmlbody
Checking page id 0 vs stored id 0
```

Given how little time we've had to work with and explore jMonkey, there are obviously many aspects of the editor that we have not been familiarized with yet, and as such, there will be notable holes in our knowledge of the engine's full capabilities.

jMonkey's editor consists for the most part of the Java file directory, an IDE, an output log, and a more standard file directory. Overall, the editor is not too different from a multi-purpose IDE, such as Visual Studio.
jMonkey also has one other very important feature, and that is the scene viewer. Scenes can be saved, loaded, viewed, and edited in the editor itself. This is a very useful tool that allows for much faster debugging and fine-tuning of values. However, the existing project we have been given was written as a standalone application, and does not integrate the scene system, rendering this feature unusable.

4.2.2 Unity


Display 1
No cameras rendering

Unity's editor has a scene view, game view, asset manager, inspector, output log, asset store, as well as a lot of extra windows to cover everything from profiling to animation. Additionally, the developer can program their own editor windows. In short, Unity's editor has everything jMonkey has, but with more prebuilt functions, and the ability to create custom windows. Unity does not allow editing source code directly in the editor, but comes with Visual Studio by default, and can be linked with a few other IDEs.

4.3 Component Systems and Scene Graphs

Unity has a component system, which in short, allows one to add an existing functionality onto any object in the scene. This can be extremely useful, and allows one to very easily create variations of the same object, or reduce clutter in the code.
jMonkey on the other hand has no such system, and is bound to the functionality and limitations of Java. Java is of course fully capable of handling such a component system, but we would have to implement that ourselves.

An example of how creating a rigidbody differs:

Unity and C#

```
GameObject go = new GameObject(); // Creates the GameObject
go.AddComponent<Rigidbody>();     //  Adds the Rigidbody
go.AddComponent<BoxCollider>();   //  Adds the BoxCollider
```

jMonkey and Java

```
Node node = new Node(); // Creates the Node or "GameObject"
rootNode.attachChild(node); // Attaches the Node to the SceneGraph
BoxCollisionShape shape = new BoxCollisionShape(); // Creates the
    CollisionShape
RigidBodyControl rb = new RigidBodyControl(shape); // Creates the
    RigidBody using the shape
node.addControl(rb);  //  Make the RigidBody control the node
bulletAppState.getPhysicsSpace().add(rb); // Add the RigidBody to the
    physics space
```

As shown in the code above, jMonkey uses twice as many lines of code to achieve the same thing, because there is no component system, and objects do not add themselves to the world or physics space.
Additionally, jMonkey requires inclusion of 4 libraries for this basic functionality, and would require manual initialization of the application, the BulletAppState, and needs to pass on references of the BulletAppState and the RootNode.

This of course does not mean that jMonkey is necessarily worse, but it shows how much extra work is required to accomplish the same thing in jMonkey. On the other hand, jMonkey allows omitting the physics space entirely, which will add a minimal improvement to performance if the program won't be using it.

4.4 Asset Management

In jMonkey assets are managed using the AssetManager. It allows loading assets from code, and keeping the path to the assets consistent on all platforms. It handles caching and optimization of OpenGL objects. By default assets are bundled into the executable.

In Unity the main way of using assets is by drag and drop in the editor, using the component system. Additionally the resource system allows for loading assets in code.
Unity also has a lot of additional systems for assets in the editor, allowing things like changing format, compression, mip-maps, LOD, color-depth, and more. In general, assets can be previewed in the editor itself, and do not need to be put into the scene to do so.

Unity's prefab system allows one to take any part of the scene graph and save it as an object. The prefab can then be copied into the scene or added through code, and allows reuse of bigger and more complex systems and objects.
The Asset Store is another very useful tool where one can download any kind of asset from an online community for Unity. Assets vary from animated models, to props, to environments, code, shaders, and whatever else needed for a projet. A lot of the more complicated assets cost money, but there are plenty of free assets available as well.

4.5 Networking

Both jMonkey and Unity have built in support for networking, but the system for jMonkey requires a lot more work on the developer's end.
The group did not have a lot of experience with Unity's networking, but the little we have done helped little to nothing when implementing the system in jMonkey, which requires a lot more setup from the developer in code.

The system in Unity also has functionality for match-making which would have helped a lot when developing networking in jMonkey.
We had to learn how to use the system in jMonkey, then how to connect to a server, and lastly how to construct messages, send them and how to deal with them when they are received at the other end.

4.6 Conclusions

Unity and jMonkey have two fairly different approaches to game engines. jMonkey allows one to control more of the engine itself, and at a lower level, and because it is open source, a developer could potentially rewrite the entire engine if needed.

Unity does not allow for the same level of control, and handles a lot more behind the scenes. This could potentially lead to issues the developer can't solve by themselves because it is engine related, or cause the program to be less optimized, but allows for a much faster and more streamlined development process.
Additionally, Unity has a much larger userbase, and it is therefore much easier to get help, solve specific problems, or download existing code or assets for a project.

5 Reorganizing the Existing Project

5.1 Thoughts on the Existing Project

In short, the existing project we were given for this project had a lot of flaws. There was no documentation for the project, the code had a minimal amount of comments, many of which didn't really help us, and both names and comments were a mix of Norwegian and English.

One of the most obvious issues we encountered was the sheer amount of code located in the main file (Program.java). Everything from initializing the project, to creating lighting, and controlling the user's character was handled in the same file, amounting to over 4000 lines of code.

There were also other aspects of the project we would have liked to improve, but the existing project is spread over almost 400 files, and we did not have the time or resources to look into all of the files and figuring out what they all did.
An example of a minor improvement we could have made is adding colliders to more of the environment, as the two only things the user can collide with as of now is the ground itself and the trains.

5.2 Relocating VR Functions and Variables

Since the controllable character and VR are the two most important aspects of this project for us, we had to locate every single related function and variable in the Program.java file. Then we had to decide whether it would be best to access them as public static variables/functions, get functions, or move them to another file entirely.

For the most part, we extracted out as much code as possible from the Program file into the new VrPerson file. This included everything from movement, to collision, to interacting with objects in and out of VR.
However, some variables are intrinsically linked to the Program class, and have to be passed on. This includes the AssetManager, RootNode, Camera, BulletAppState (Physics), VrAppState, Settings, as well as several other variables, such as movement input.

After finally getting all that over to a separate file, the new file is about 750 lines of code long, which is a lot to have directly in a main file.

5.3 Rewriting the Coupling System

The original coupling system we were given with the existing project was a temporary system that needed to be replaced.
The original system allowed aiming at a cube connected to each end of the train, and then

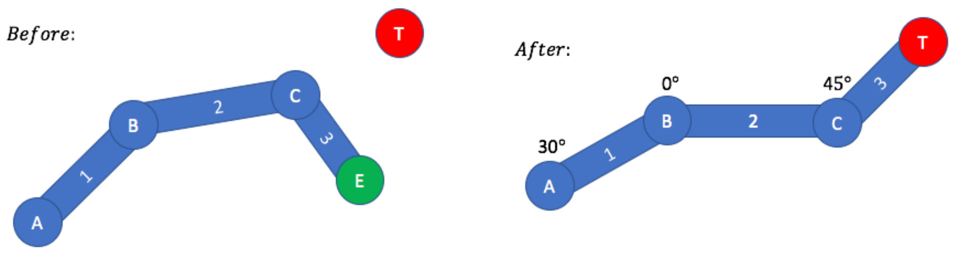
connect the train cars together utilizing a raycast check.
This system had some issues, relying on a lot of code from different files to function, and using many lines of code to move the cubes around properly.

As such, we thought it best to rewrite the entire class. We wanted the coupling to function as closely to real life as possible, as this was one of the most important aspects of the entire project.
When coupling a train in real life, one has to walk up to the end of the train car, pick up the segmented chain, connect it to the hook of a nearby train car's coupling, and tighten it with the handle on the middle-segment.

With this in mind we decided that the chain itself could be in one of three states:

- Coupled or fastened to an object.
- Held by a person.
- Hanging loosely.

Since the chain would have to be very movable in order to be held by a hand and moved around, but also restricted by physical properties, we thought the best solution for this problem would be inverse kinematics.



Source: *medium.com/unity3danimation*

Inverse kinematics is a system which aims to rotate a segmented limb so that the outermost part of it reaches a given position.
In this project we utilized this so that the segmented coupling chain would follow the user's hand around as they carried it, and it would automatically adjust for positional changes in case the hook it was connected to moved around.

Implementing the inverse kinematics was not much of an issue; it's a well known system that is used for all kinds of robotics as well as digital animation, and we even found an implementation for jMonkey (albeit an outdated one). We did however encounter one problem; The chain itself is comprised of several interlocked parts, many of which can only rotate around one local axis. The inverse kinematics system we implemented did not take this into account.

The ideal solution would just be to lock the rotational axis of each segment, but jMonkey does not support this.
We attempted to manually rotate each segment back into place after the inverse kinematics

each frame, but we could not get this to work.
Eventually we found a solution that involved moving the inverse kinematics goal onto a forward line in local space from the coupling's perspective. Essentially causing the inverse kinematics system to be in 2 dimensions rather than 3.

We are quite content with how the coupling system worked out, and we reused the code for the air hoses or "AirCouplings" as they are named in the project. The couplings' movement is generally quite natural, and it feels as if they are actually connecting in VR.

The only big downside with the inverse kinematics system is that we do not account for gravity; When a chain/hose is hanging loosely, the middle segments should hang towards the ground, with enough fine-tuning we could probably have made this work, but we prioritized other systems.

We also had to rewrite how the train handled coupled carts, the way it was when we started the project was that the train set the position of all connected carts. This was the same system as for pushing the other carts when connected. We decided to have those as two different functionalities, as in the train can push other carts whether they are connected or not, and coupling only affects the dragging of carts.

For dragging carts we check each coupling on the train if they are connected to anything, then we check which direction the train is going. If the train is going away from the cart it is connected to and has higher speed than the cart it will set the speed and direction of that cart and then run a smaller update function on that cart that does this again but only for one of the couplings, checking the same coupling that is connected to the one we just checked would be problematic.

Then if the direction is going towards the cart it is connected to and the speed of the cart is higher than the speed of the train it will again set the speed and direction for the cart, which has the effect of braking the cart, before we again run the other smaller update function.

The final thing we do is if a coupling is not connected but the air coupling is connected and the train and cart are going away from each other, then we will disconnect the air couplings because they are not strong enough to stay connected.

The way the couplings work is that they start at the locomotive and then checks connected trains, this should have been more like the system for collision in that every cart checks itself even if it is not connected to the locomotive.
This would have been better but it would be a big change and would likely require a rewrite of the most of the system we made, and it would not give much benefit because at the current state there is not often a moment where a cart is dragging another cart without being connected to the locomotive, but for more physical accuracy and more accuracy in general in unforeseen events.

```
if (coupling1.getConnectedTo() != null) {
    if (direction > 0 && speedKmh >
        coupling1.getConnectedTo().getVehicle().getSpeedKmh()) {
        // If dragging a cart
        coupling1.getConnectedTo().getVehicle().setSpeedKmh(speedKmh);
        coupling1.getConnectedTo().getVehicle().setDirection(direction);
        coupling1.getConnectedTo().getVehicle().tick1();
    } else if (direction <= 0 && speedKmh <
        coupling1.getConnectedTo().getVehicle().getSpeedKmh()) {
        // If braking a cart
        coupling1.getConnectedTo().getVehicle().setSpeedKmh(speedKmh);
        coupling1.getConnectedTo().getVehicle().setDirection(direction);
        coupling1.getConnectedTo().getVehicle().tick1();
    }
} else if (air1.getConnectedTo() != null && this.speedKmh > 0
        && ((direction > 0 && speedKmh >
            air1.getConnectedTo().getVehicle().getSpeedKmh())
        || (direction <= 0 && speedKmh <
            air1.getConnectedTo().getVehicle().getSpeedKmh()))) {
    air1.disconnect();
    // Disconnects air if it is connected to a cart going away from it
}
// Should do something else in the future, maybe disconnecting violently
if (coupling2.getConnectedTo() != null) {
    if (direction <= 0 && speedKmh >
        coupling2.getConnectedTo().getVehicle().getSpeedKmh()) {
        // If dragging a cart
        coupling2.getConnectedTo().getVehicle().setSpeedKmh(speedKmh * 1);
        coupling2.getConnectedTo().getVehicle().setDirection(direction);
        coupling2.getConnectedTo().getVehicle().tick2();
    } else if (direction > 0 && speedKmh <
        coupling2.getConnectedTo().getVehicle().getSpeedKmh()) {
        // If braking a cart
        coupling2.getConnectedTo().getVehicle().setSpeedKmh(speedKmh);
        coupling2.getConnectedTo().getVehicle().setDirection(direction);
        coupling2.getConnectedTo().getVehicle().tick2();
    }
} else if (air2.getConnectedTo() != null && this.speedKmh > 0
        && ((direction <= 0 && speedKmh >
            air2.getConnectedTo().getVehicle().getSpeedKmh())
        || (direction < 0 && speedKmh <
            air2.getConnectedTo().getVehicle().getSpeedKmh()))) {
    air2.disconnect();
    // Disconnects air if it is connected to a cart going away from it
}
```

5.4 Rewriting the Player Collision System

The original collision system was quite rudimentary. When the user wanted to move, the system would make a ray cast collision check starting a short distance above the user's head, it would then move them in the direction they wanted across the plane, and adjust their height based on height differences found with the ray cast.

This system worked without issue at the time, but it only accounted for the static terrain. Since the only other thing we needed the player to collide with was the trains, we decided

to expand on the existing collision system rather than rewrite it entirely.
This however proved to be a challenge. 3D collisions are innately complex, and while we did get a fairly well-functioning system in place, it wasn't good enough. We had issues with inconsistent movement when attempting to move through the trains, as well as inconsistent behaviour when the trains were in motion.

With that in mind we decided to look more closely at jMonkey's existing solutions for physics. With such an old engine, and such a sparse community, it can often be difficult to find the relevant documentation. This was one of the main issues we encountered when applying physics to the player character.

Like most player characters, we used a capsule collider for collisions. However, unlike other engines, the rotational axis of the standard physics objects can't be locked. We tried finding alternate solutions, but only came up with a handful of slightly related posts, many of which contained outdated code.

Eventually we found the BetterCharacterController class, which was automatically locked to only rotate around its own up-vector, but that was not an ideal solution either, as it lacks some of the basic functions in the standard RigidBody class, which caused issues when trying to sync up with the update loop. jMonkey also doesn't have an easily accessible physics update loop, which meant that our player character was now moving with a separate update loop, causing a disconnect between the movement of the player and the trains.

This was an issue when the player boarded the train and the trains started moving. From the player's perspective, they ended up getting pushed farther and farther back as the train went faster, and the train looked to be vibrating.
This was solved by having the player be locked in place during an update loop, and using the guardrails as clickable objects to get on or off.
Since the user no longer needed to actively walk on the train, we replaced the train's more complex collision shape with a standard box collider.

5.5 Rewriting the Train Collision System

The collision system for the trains already in place at the start of this project was pretty basic; it checked the distance between two couplings, and if it was small enough it would just stop the moving train. This system was also divided into three functions, one main one that checked which direction the train was going, and one each for each direction.

The system when connected did not work the same, when the trains were connected they would update the positions of connected carts by setting the position as the position plus the half of the length of the train.

This system was changed to when detecting a collision it would update the speed of the cart at the other end of the collision and the direction, but later we also updated it so it would set the position so we would not have clipping. We also have variables for what coupling the cart is pushed by and for what coupling the train is pushing. These are reset in the update function for the carts when they notice that the coupling they are pushed by is too far away to be able to be pushed by it.

```
if (speedKmh < 0.15f)
       return;

Coupling coupling;
float spacing = vd.trainColliderLength;

if (direction > 0) {
    coupling = getCoupling(1);
} else {
    coupling = getCoupling(0);
}

for (Vehicle v : Program.stVehicleList) {
    if (v != coupling.getVehicle()) {
        for (int i = 0; i < 2; i++) {
            if
                (v.getCoupling(i).getBasePivot().getWorldTranslation().distance(
                   coupling.getBasePivot().getWorldTranslation()) < 0.83f) {
                if (v.getSpeedKmh() < speedKmh) {
                    float totalSpeed = speedKmh + v.getSpeedKmh();
                    float dist = this.odometer - v.getOdometer();
                    System.out.println(dist);
                    spacing += v.vd.trainColliderLength;
                    if (direction <= 0)
                        spacing *= -1;
                    v.setOdometer(odometer + spacing);

                    v.setSpeedKmh(totalSpeed / 2);
                    this.setSpeedKmh(totalSpeed / 2);

                    v.setDirection(direction);

                    this.pushingCoupling = v.getCoupling(i);
                    v.setPushedBy(coupling);
                }
                return;
            }
        }
    }
}
```

We also changed it so the the system for pushing carts was different and independent of the system for dragging carts when they were connected to the train. Unlike the system for couplings dragging carts, this system is ran on every cart on its own, and is not dependant on being pushed by the locomotive.

6 Modelling

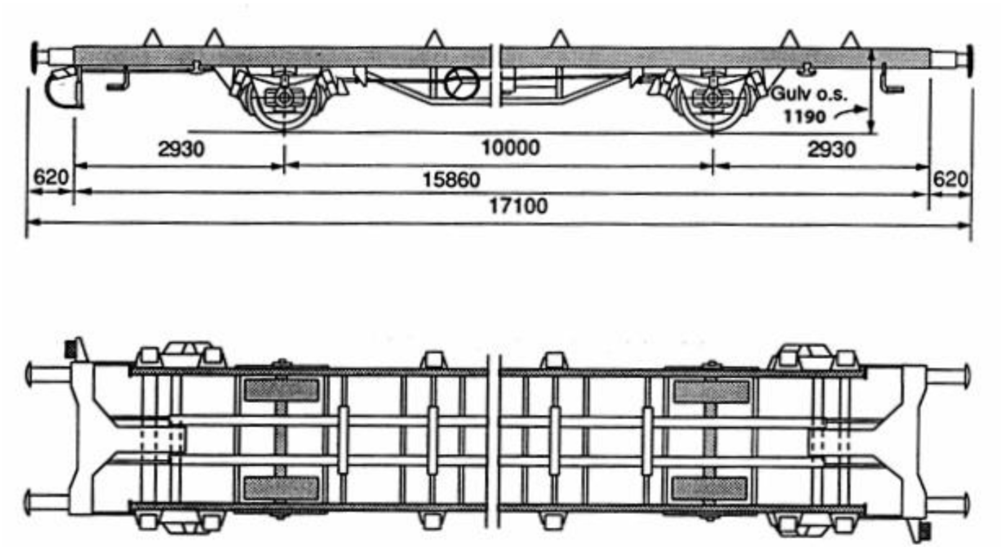
6.1 Tools

For 3D modelling the trains, blender was the tool of choice from the railway school. This is helpful since jMonkey can import .blend files directly, which makes it easy to add and later make changes to the 3D models. Blender is a tool that is relatively easy to start learning, yet it still offers a wide variety of tools to increase modelling efficiency. The modeller had some previous experience with Blender, and got to utilize that for this project.

The textures for the models were mainly created in Substance Painter. Substance is a 3D painting program that has a suite of powerful tools for generating procedural textures. This was one of the first projects the modeller used substance for, a lot of experimenting went into making the textures.

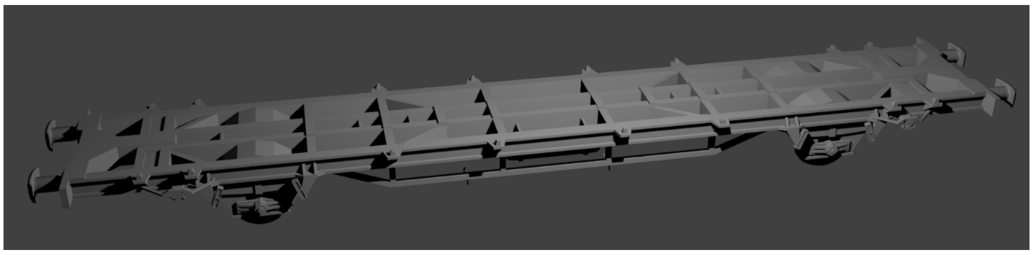
6.2 Process

The process of modelling a piece of the train starts with gathering the necessary reference material. The reference material consisted of images, and plans of the wagons. This proved the most limiting factor in the modelling process since most of the people who were asked for reference material seemed willing to share, but never delivered.



Reference for the Lgns wagon, from a material description provided by the Railway School.

For this particular wagon, and one other, the Railway School also provided a physical model. Although lacking some important details it proved extremely helpful to the modelling process.



The finished model in Blender

Before texturing a model it needs to be UV-mapped, to make sure the faces don't overlap in the texture, this is done within blender itself. It is a tedious process that only gets more time-consuming the more work is put into the model itself.

Inside Substance, the first step is to create the base material, then adding weathering like dirt and rust, and finally going over and adding details by hand.

After texturing

After exporting the textures from Substance, they are put into the Blender project and are imported with the model in jMonkey. Unfortunately jMonkey is too old to support the node materials in Blender, or it's the Cycles rendering engine. The legacy Blender render had to be used for the material instead.

7 Networking

7.1 Development

The networking component of this project was not fully completed, but we did implement a few basic networking components. There was a system in place from before to handle networking built in the jMonkey engine, and there were a few small tutorials on the topic.

The issue we encountered with the tutorials we found was that they were only for localhost servers, so all we got out of the tutorials was how to send messages, and how to set up the connection when we already know the IP address.

This means that the only way we have of connecting a client to a server is if we know the IP of the server before we start the client and put it in the code. We could improve it by finding a way to enter the IP while the program is running but it seems there is no way to automatically find and connect to servers on the local network using the code exmaples we have found.
There also seemed like there was no way to find the IP of a server so that it could display it, meaning we have to find the IP by other means; We used ipconfig in the terminal to find the local IP.

Here is how we started the server and how we connected a client:

```
if(isServer)
{
    try {
        myServer = Network.createServer(6143);
        myServer.start();
        myServer.addMessageListener(new ServerMessageListener());
    } catch (IOException ex) {
        Logger.getLogger(Program.class.getName()).log(Level.SEVERE, null,
            ex);
    }
}
else if(isClient)
{
    try {
        // Works only with exact local ip address
        // TODO: Find ip address automatically
        // TODO: Enter IP here
        myClient = Network.connectToServer("0.0.0.0", 6143);
        myClient.start();
        myClient.addMessageListener(new ClientMessageListener());

    } catch (IOException ex) {
        Logger.getLogger(Program.class.getName()).log(Level.SEVERE, null,
            ex);
    }
}
```

We have only basic networking where we update the position of the train and update the position of the person in VR. This has been achieved by implementing two different types of messages, one for updating position of the VrPerson and one for updating the position of the train.

The message for updating the position of the VrPerson is set up to send the rotation and the position of the VrPerson so that we can update where they are and what direction they are facing, and the client sends this message every time there is a change in either of them, it also sends a message every half second.

```java
public static class VrPersonMoveMessage extends AbstractMessage
{
    private Vector3f pos;
    private Vector3f rot;

    public VrPersonMoveMessage(){}

    public VrPersonMoveMessage(Vector3f p, Vector3f r)
    {
        pos = p;
        rot = r;
    }

    public Vector3f getPos()
    {
        return pos;
    }

    public Vector3f getRot()
    {
        return rot;
    }
}
```

This message also only sends part of the VrPerson's look direction, it does not include where the VrPerson is looking on the vertical axis, meaning one would not be able to tell if the person is looking up or down.

Messaging for updating the position of the train contains its speed, direction, and position on the track. This message is sent whenever there is a change in any of the components or every half second, just like the messages for VrPerson.

```
public static class TrainMoveMessage extends AbstractMessage
{
    private float speedKMH;
    private float odom;
    private int dir;

    public TrainMoveMessage(){}

    public TrainMoveMessage(float sKMH, float o, int d)
    {
        speedKMH = sKMH;
        odom = o;
        dir = d;
    }

    public float getSpeed()
    {
        return speedKMH;
    }

    public float getOdom()
    {
        return odom;
    }

    public int getDir()
    {
        return dir;
    }
}
```

The server has one listener that waits for messages from the client about a movement for the VrPerson, and likewise the client has one listener that waits for messages from the server about movements for the train.

```
public static class ClientMessageListener implements
    MessageListener<Client>
{

    @Override
    public void messageReceived(Client source, Message m)
    {
        if (m instanceof TrainMoveMessage)
        {
            TrainMoveMessage message = (TrainMoveMessage) m;
            Program.trainMoveQueue = message;
        }
    }
}

public static class ServerMessageListener implements
    MessageListener<HostedConnection>
{

    @Override
    public void messageReceived(HostedConnection source, Message m)
    {
        System.out.println("Message Recieved");
        if (m instanceof VrPersonMoveMessage)
        {
            VrPersonMoveMessage message = (VrPersonMoveMessage) m;
            Program.vrPersonMoveQueue = message;
        }
    }
}
```

How the server sends and handles recieved messages:

```
if(myServer.hasConnections())
{
    timeSinceUpdate += tpf;
    float newSpeed = simVeh.getSpeedKmh();
    float newOdom = simVeh.getOdometer();
    int newDir = simVeh.getDirection();
    if(!(newSpeed == lastSpeed && newOdom == lastOdom && newDir == lastDir)
        || timeSinceUpdate > 0.5f)
    {
        timeSinceUpdate = 0;
        myServer.broadcast(new TrainMoveMessage(newSpeed, newOdom, newDir));
        lastSpeed = newSpeed;
        lastOdom = newOdom;
        lastDir = newDir;
    }
}

VrPersonMoveMessage m = vrPersonMoveQueue;
if(m != null)
{
    vrPerson.setPos(m.getPos());
    vrPerson.setRot(m.getRot());
    vrPersonMoveQueue = null;
}
```

How the client sends and handles recieved messages:

```
if(isClient)
{
    if(myClient.isConnected())
    {
        timeSinceUpdate += tpf;
        Vector3f newPos = new Vector3f();
        Vector3f newRot = new Vector3f();

        newPos.set(vrPerson.getPos());
        newRot.set(vrPerson.getRot());
        if(!(newPos == lastPos && newRot == lastRot) || timeSinceUpdate >
            0.5f)
        {
            timeSinceUpdate = 0;
            System.out.println(lastPos);
            myClient.send(new VrPersonMoveMessage(newPos, newRot));
            lastPos.set(newPos);
            lastRot.set(newRot);
            System.out.println(lastPos);
            System.out.println("Message sent");
        }
    }

    TrainMoveMessage m = trainMoveQueue;
    if(m != null)
    {
        simVeh.setSpeedKmh(m.getSpeed());
        simVeh.setOdometer(m.getOdom());
        simVeh.setDirection(m.getDir());
        trainMoveQueue = null;
    }
}
```

7.2 Future Development

In this section we would like to discuss what we would have liked to do with the networking if we had had the time to develop it further. Its current state is clearly not ideal, and needs further enhancements to function according to its original purpose.

First of all, the current solution for connecting server and client is simply not good enough. We would want to implement a way for a client computer to search the LAN to connect to the server computer.

Additionally, we would want to implement synchronization for more objects, so that all object interactions are consistent across all networked computers. This is not in any way difficult, and would only require more time.

And the last thing we would need to implement would be communication between the two users. Part of this would be to give an avatar to the VrPerson, and give mirrors to look through for the person controlling the locomotives. The other and more complex part of this would be the voice communication feature.

None of us have ever implemented anything more than the most basic of networking features, so unless we could find an available library written in Java, this could prove to be quite challenging.

7.3 Conclusions

The networking has been harder than we thought and we still haven't found a great way of setting up the connection. Currently we connect to a specific IP, but we would like to search the local IPs and connect to a server or at least give the user the possibility to enter the IP themselves, but we have not found a way to find the IP on the server, nor a way for the user to input the IP themselves.

This would be a problem when the railway school is going to use it themselves, because they will have different machines that will function as the servers and different machines that will function as clients, so we need a way to allow connecting to different machines without having to hardcode whatever IP address it may have.

8 Conclusions

8.1 Development Model

As described earlier in the report, we have been developing with an iterative process with modules taking focus at different times based on priority. This has worked quite well for our group, and has allowed us to finish the most important parts first, and to go back and iterate on said parts when necessary.

8.2 Meetings and Feedback

We ended up with a lot fewer meetings than initially planned. We have for the most part kept up with meetings with our supervisor, having meetings weekly in the beginning and less frequently when it proved unnecessary to meet that often.

We did however not have nearly as many meetings with our employer. While we didn't need as many meetings as we originally planned due to progress going smoothly, we would have liked to have more towards the end of development to better discuss what was most important to finish for the project.

As planned, we did write reports on the state of development around 4th of March and 8th of April. These reports will be appended.

8.3 Development Process

In short, we have not worked as much nor as hard as we should have. While we initially made good progress and kept up with our planned schedule, as we approached Easter we were no longer up to speed, and issues with both the advanced braking system and networking made us slow down further.

Because of this, we have a fair few incomplete parts related to braking and networking, leaving us with an unfinished product.
This is however not a massive issue, as the project was meant to be something our employer could build upon, rather than a complete entity.

8.4 Final Thoughts

Overall the main hurdles in the project have been rooted in working with the existing project and jMonkey. A lot of time has been wasted on understanding the code and getting used to working with jMonkey. This is, of course, what we expected at the start of the project, and despite those challenges we have for the most part managed to stay reasonably on schedule. We are also fairly content with what we have managed to achieve during the project, as is the Norwegian Railway School, even if we haven't reached our proposed work hours, nor all of our milestones.

Appendix

Appended documents are in Norwegian.

A Project Plan

Prosjektplan, Gruppe 2

Sindre B. Garvik
Haakon B. Aarstein
Eirik Hiis-Hauge

1. MÅL OG RAMMER

1.1. Bakgrunn

Norsk fagskole for lokomotivførere ønsker en utvidelse av et eksisterende prosjekt via en prototype. Det eksisterende prosjektet er en lokfører simulator, og utvidelsen vil gå ut på å koble sammen tog i VR (Virtual Reality).

1.2. Prosjektmål

Vi har fått en rekke moduler vi kan implementere, og har blitt enige om prioriteten for hver av de sammen med oppdragsgiver:

1. Sammenkobling av vogner i VR
2. Fysiske egenskaper på vognene
3. Samkjøring mellom den eksisterende simulatoren og dette prosjektet
4. 3D-modellering av tog/vogner

5. Datastyrte scenarioer for sammenkobling

Målet er å få de to første modulene til å funke godt, modul 3 og 4 er høyst ønskelig å få til, og modul 5 er noe vi kan gjøre om vi får ekstra tid, men er ellers lite prioritert.

1.3. Rammer

Siden vi jobber med et eksisterende prosjekt er vi nødt til å jobbe med samme engine og språk som det originale prosjektet. Dette er da jMonkeyEngine og Java. Tidsrammen vår er bachelor perioden, og vi er tre studenter som jobber på prosjektet.

2. OMFANG ved programvareutvikling

2.1. Fagområde

En av tingene de lærer på jernbaneskolen er skifting, som betyr å flytte og koble sammen lokomotiv og vogner. Opplæringen for dette er både komplisert, tidkrevende og kostbar. De vil derfor flytte den inn i en virtuell verden.

2.2. Avgrensning

Oppgaven vil i stor grad gå ut på kobling av tog spesifikt, andre relaterte problemstillinger som for eksempel innebærer lokfører vil ikke være relevant med mindre det er viktig for samkjøringa. Mange av de utfordringene vi potensielt kunne hatt, som grafikk eller hvordan programmet laster inn miljøet, har allerede blitt gjort i det eksisterende prosjektet.

2.3. Oppgavebeskrivelse

Oppgaven går ut på å utvide / videreprogrammere eksisterende programvare til desksimulatorene ved Norsk Fagskole for lokomotivførere. Det som ønskes utført er delt inn i moduler.

Modul sammenkobling

Lok kjøres inntil vogn. kobling av kobbel på krok, kobling av luft, evt kobling av UIC kabel foretas i VR miljø. Lok og tilkoblet vogn "oppfører seg" i forhold til dette eks. vogn henger med loket. Lok / vogn flytter seg ikke dersom brems er tilsatt. Luft fra lok fyller systemene på vogna og bremser oppfører seg i forhold til dette.

o Lok manøvreres.

o Når lok er inntil vogn (vogn er inntil vogn) kan de kobles sammen. Dette er fysisk kobbel som hektes på krok og skrues / strammes til. Det er luftslange (hovedledning) - en på hver lok/vogn som kobles sammen - luftkraner - en på hver lok/vogn som kan åpnes. En avatar skal ikke være synlig for lokomotivfører i (eventuelle) speil når skifteren er i mellom vognene og ufører dette. Synlig igjen når skifteren går ut av koblingsrommet.

o Når to vogner er koblet henger de sammen.

o Denne sammenkoblingen av luftslangene medfører at luftforbindelsen mellom lok/vogn og neste lok/vogn opprettes. Dette har mye å si for bremsesystemet i toget og dermed hvorvidt toget vil flytte på seg ved kjøreforsøk, begynne å trille osv.

Modul fysiske egenskaper

Hva som skjer med et tog, en vogn, et lok i ulike situasjoner er avhengig av en rekke fysiske forhold.

o Vogn triller når den står i fall uten brems på.
o Vogn triller ikke dersom den står i fall med tilstrekkelig brems på.
o Hva skjer dersom et lok kjører mot en vogn som står stille, dyttes den avgårde? I liten grad dersom bremsen er på, i større grad dersom bremsen ikke er på.
o Bremseklosser går mot hjul når brems blir tilsatt.
o Kobbel kan løftes av og på krok og strammes.
o Luftslanger kan løftes sammen og kobles.
o Luftkraner kan åpnes.
o Lyd.

Modul samkjøring

Programvaren som kjøres på DeskSim og på VR-stasjon er den samme.
o Evt ulik "oppførsel" i programmet fra DeskSim til VR settes gjennom konfigurasjonsparametere eller gjennom detektering av VR utstyr.

Via intern WiFi samkjøres disse slik at de jobber i samme scenarie
o Simulatorene jobber i scenarier.
o Når det samme scenariet lastes på både DeskSim og VR stasjon skal det oppleves at man er i samme "verden".
o Dersom loket flyttes på i DeskSim, ser man at loket flytter seg i VR miljøet dersom det skjer innenfor synsfeltet til VR-personen.
o Dersom man flytter seg eller utfører ting i VR miljøet ses dette fra DeskSim dersom det skjer innefor synsfeltet til DeskSim-føreren. Det bør være i form av en verne-kledd avatar.
o Handlinger som utføres gjenspeiles / utføres begge steder. Eksempelvis kobles lok og vogn henger de sammen begge steder, Endres et signalbilde endres det begge steder, legges en sporveksel over skjer det begge steder.

Kommunikasjon mellom skifter og fører (DeskSim og VR-miljø).
o Kommunikasjon foregår normalt gjennom en skifteradio (walkie talkie).
o Det finnes også håndsignaler. Avataren utfører et utvalg av håndsignaler som et resultat av at den med VR-briller foretar en bevisst handling for å gi riktig signal.
o En form for simulering av skifteradio må være en del av systemet. Det kan være å prate, det kan være å velge fra en meny av meldinger.
o Kommunikasjonen må kunne gå begge veier på en håndterbar måte både for den med VR briller på og den som sitter i DeskSim.
o Simulering av håndsignaler er også ønskelig, men lavere prioritert Avataren utfører et utvalg av håndsignaler som et resultat av at den med VR-briller foretar en bevisst handling for å gi riktig signal.

Modul modeller

Lages i Blender, lok og vogner ligner virkeligheten.
- En modell av Traxx lok.
- En modell av skiftelok.
- En modell av 6 aksla containervogn (leddet).
- 4 modeller av containere, ulike farger. Skal kunne settes på forrige nevnte vogn.
- En tankvogn.
- En pukkvogn.
- En lukka godsvogn med skyvedører.

Komponenter ligner virkeligheten.
- Styreventil.
- Bremsegruppestiller.
- Parkbremsbetjening.
- Manuell lastveksel.

Lesbare påskrifter skal være med.
VR person skal ha hender som oppfører seg i samsvar med betjening av håndkontrollere til Oculus.

Modul automasjon

Programvare kjører med et scenario der "programmet er fører" og flytter toget i samsvar med meldinger fra skifter / student - "trekk fram, kom bak, sakte, stopp osv". Samme løsning som å få avataren til å utføre/gi meldinger, men denne gangen uten synlig avatar.
- Vedkommende som har på seg VR-briller er skifter, går på bakken.
- Programvaren skal i denne modusen "spille" rollen som fører
- Lok (og det som er tilkoblet loket) skal bevege seg i forhold til meldinger fra skifter av type "kom bak", "trekk fram", "stopp", "sakte", "30 meter igjen til butt","15 meter igjen til butt".
- Talegjenkjenning er antagelig i overkant for dette prosjektet så vi ser for oss at dette utføres ved hjelp av en meny av meldinger på en eller annen måte tilgjengelig for den som er i VR miljøet samme løsning som å få avataren til å utføre/gi meldinger, men denne gangen uten synlig avatar.

Programvare kjører med et scenario der "programmet er skifter" og gir meldinger til fører/ student ser helt lik ut som avataren.
- Motsatt modus av modulen over.
- DeskSim opereres av deltager, programvaren utfører skifteoperasjonene.
- Loket manøvreres i henhold til meldinger fra "skifter" som er programvaren i denne modusen.
- Skifteoperasjonen som skal utføres må defineres i scenariet.

3. PROSJEKTORGANISERING

3.1. Ansvarsforhold og roller

Sindre har blitt valgt til å være gruppeleder.
Haakon har hovedansvar for 3D-modellering i prosjektet.

3.2. Rutiner og regler i gruppa

Arbeidsmengde:

- 30 timers arbeidsuke
- Sanksjon om arbeidsmengde ikke er nådd er å bake kake til resten

Arbeidsvaner:

- Ikke push ting som ikke funker til master
- Commit fullstendige enheter
- Kommenter koden godt og ofte

Problemløsning:

- Ved større problemer, for eksempel at vedkommende ikke har bidratt i løpet av 14 dager, får vedkommende advarsel, og gruppa har møte med veileder. Om ingenting blir gjort 7 dager etter møtet må potensielt vedkommende forlate gruppa om ingen annen løsning er nådd.

4. PLANLEGGING, OPPFØLGING OG RAPPORTERING

4.1. Hovedinndeling av prosjektet - Valg av SU-modell/prosessrammeverk med argumentasjon - Valg av Metode og tilnærming (avklare Teori- og Metodebruk)

Gantt-diagrammet senere i planen kan få det til å se ut som om vi bruker fossefallsmodellen. Dette stemmer ikke helt siden diagrammet i stor grad er ment for å vise hvilke moduler vi prioriterer først. Vi har inndelt oppgaven som vist i 1.2. Prosjektmål, og vi kommer til å bruke Trello til å organisere og planlegge oppgaver under hver modul, og hvor de er hen i utviklinga. Dette vil tilsi at vi inndeler oppgaver med en backlog for hver modul, en rekke oppgaver som er i utvikling, en liste over ting som må testes, og en liste med ferdige oppgaver.
Vi jobber derfor etter en mer inkrementell modell.

4.2. Plan for statusmøter og beslutningspunkter i perioden

Vi har avtalt å møte omtrent ukentlig med veileder på universitetet, og tilsvarende med oppdragsgiver over nett. Møtene med veileder er for øyeblikket satt til å være hver Mandag.
Vi har som plan å skrive en statusrapport nær slutten av første modul, da rundt 04. mars, og en ved planlagt slutt av andre modul, da 08. April. Om moduler ikke er ferdig etter planen og en større mengde arbeid gjenstår planlegger vi å kutte mindre prioriterte moduler.

5. ORGANISERING AV KVALITETSSIKRING

5.1. Dokumentasjon, standardbruk og kildekode

Prosjektet vil være tilgjengelig via en git løsning. Utviklingen av prosjektet vil derfor være loggført via commits, som senere kan brukes til å skrive rapporten i mai. Vi tenker å lage et klassediagram for å dokumentere klassene og filene vi lager til prosjektet for å bedre dokumentere det vi har jobbet med, og får at oppdragsgiver skal kunne få bedre oversikt over strukturen til programmet.
Vi kommer til å følge en commit-standard ikke så ulik den her:
https://www.conventionalcommits.org/en/v1.0.0-beta.2/
Der grunntrekkene er at hver commit melding skal begynne med hvilke type endring som er gjort, om det er en liten bug fix eller om det er en helt ny feature, eller andre ting. Det skal og være en liten kort beskrivelse på hva har blitt gjort. Det er også bra hvis et relevant scope er beskrevet, men dette er ikke et krav. Det er i tillegg bra hvis det blir lagt til en lengre beskrivelse under det første, som da forklarer mer i detalj hva som er endret, men dette er heller ikke et krav.

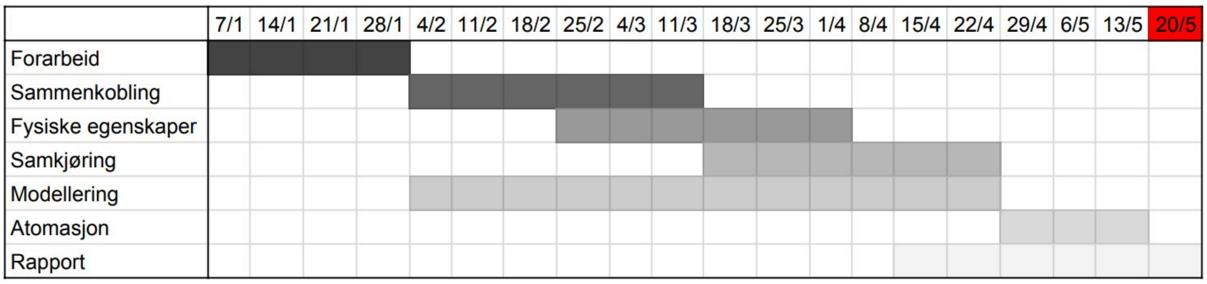
5.2. Risikoanalyse (identifisere, analysere, tiltak, oppfølging) Teknologi, Forretningsmessig, Prosjektgruppemessig

Sikkerhet er veldig lite relevant til oppgaven. Vi produserer en prototype i et eksisterende prosjekt for et produkt som egentlig bare skal brukes av oppdragsgiver internt. Det er ingen personlige data i prosjektet, man får ingenting ut av å krasje programmet, du er alene inne i simulatoren.

Samkjøringa er egentlig det eneste som kan være et sikkerhetsproblem på noe vis, siden det krever at man kommuniserer mellom to datamaskiner.
Av den grunn er det hovedsakelig bare det å få til trygg nettverkstrafikk som blir en prioritet under utvikling. Dette vil inkludere sanitering av input til programmet.

Om noen skulle ønske å stjele prosjektet så må de enten stjele prototypen vår via det private git-repositoriet vårt, eller kopiere over filene fra PCene til oppdragsgiver. Risiko for svikt fra gruppe medlemmer er svært liten men vi har på plass regler og prosedyrer i tilfelle det skal skje. Sykdom er en risiko som er vanskelig å forutse men vi ser på den som veldig liten og i det tilfelle vil resterende gruppe medlemmer måtte i verste fall jobbe litt ekstra i en periode.

6. PLAN FOR GJENNOMFØRING



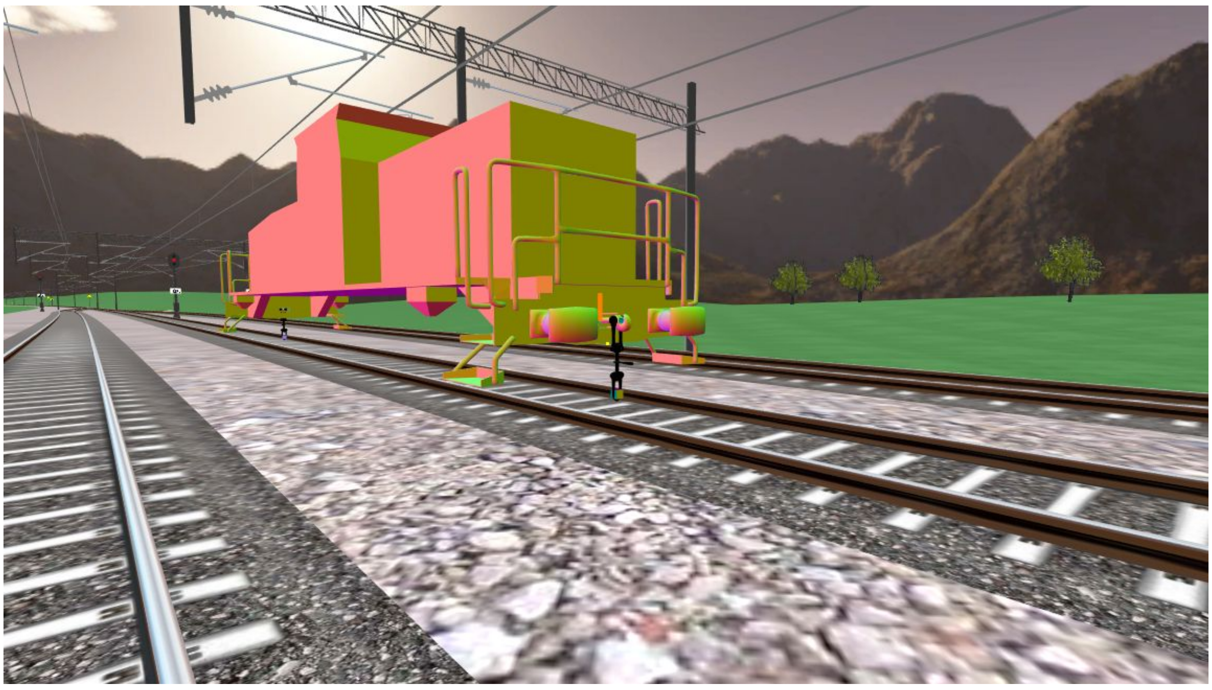
	7/1	14/1	21/1	28/1	4/2	11/2	18/2	25/2	4/3	11/3	18/3	25/3	1/4	8/4	15/4	22/4	29/4	6/5	13/5	20/5
Forarbeid	■	■	■	■																
Sammenkobling					■	■	■	■	■	■										
Fysiske egenskaper								■	■	■	■	■	■							
Samkjøring											■	■	■	■	■	■				
Modellering					■	■	■	■	■	■	■	■	■	■	■	■				
Atomasjon																	■	■	■	
Rapport															■	■	■	■	■	■

B Status Report 1

Status Rapport 1

- Vi har fullført at lok kan manøvreres.
- Vi har nesten fullført at lok kan kobles med vogn, eneste som mangler er luft kraner og synlighet gjennom speil. (mangler mer spesifikk info om hvordan luftkabler og kraner fungerer) Lokfører er ikke enda i VR, dette blir ikke relevant før vi har samkjøring, og derfor er heller ikke speil det.
- Vi har fullført at to vogner som er koblet sammen henger sammen.
- Vi har ikke fullført at luftslanger påvirker bremsing, dette skal ikke mye til, og er noe vi enkelt kan legge til etter at luftslanger virker slik vi vil.

- Vi har ikke begynt at vogn triller i fall uten brems
- Vi har begynt at vogn triller om den står i fall uten nok brems. Samme som forrige?
- Vi har begynt på lok som kolliderer med vogn og dytter tog, vi mangler bremsing.
- Vi har ikke begynt på bremser synlig går inn på hjul, dette gjøres fort så snart hjul og bremse modeller er inne.
- Vi har begynt på kobbel kan løftes av og på krok, Stramming fungerer, men kan ikke interageres med enda.
- Vi har ikke begynt på luftslanger, dette kan gjøres fort, mye kan kopieres fra lenke-logikken.
- Vi har ikke begynt på luft kraner, dette kan gjøres fort og enkelt.
- Vi har ikke begynt på å legge til mer lyd.



C Status Report 2

Status Rapport 2

Vi ligger generelt greit ann, mesteparten har blitt gjort etter planen.
Som nevnt under forrige møte har vi tenkt å droppe modul 5 (automasjon) fullstendig.
Noen punkter i Modul 1 og 2 er ikke helt ferdige, men det er i stor grad detaljarbeid som vi itererer på videre.

Modul 1, sammenkobling

- Vi har fullført at lok kan manøvreres.
- Vi har nesten fullført at lok kan kobles med vogn, eneste som mangler er synlighet gjennom speil. Lokfører er ikke enda i VR, dette blir ikke relevant før vi har samkjøring, og derfor er heller ikke speil det.
- Vi har fullført at to vogner som er koblet sammen henger sammen.
- Vi har begynt på at luftslanger påvirker bremsing, vi har et fungerende system, men vi ønsker å implementere et realistisk system, noe som vil kreve mange kompliserte detaljer.

Modul 2, fysiske egenskaper

- Vi har begynt at vogn triller om den står i fall uten nok brems.
- Vi har begynt på lok som kolliderer med vogn og dytter tog, trenger litt justering.
- Bremser synlig går inn på hjul.
- Kobbel kan løftes av og på krok.
- Vi har begynt på luftslanger, vi ønsker mer detaljer på de.
- Luftkraner er ferdig.
- Vi har ikke begynt på å legge til mer lyd.

Modul 3, samkjøring

Programvaren som kjøres på DeskSim og på VR-stasjon er den samme.

- Påbegynt ulik "oppførsel" i programmet fra DeskSim til VR settes gjennom konfigurasjonsparameter.

Via intern WiFi samkjøres disse slik at de jobber i samme scenarie

- Simulatorene jobber i scenarier.
- Påbegynt at når det samme scenariet lastes på både DeskSim og VR stasjon skal det oppleves at man er i samme "verden", funker for øyeblikket bare hvis lokal ip adresse til server er kjent fra før og lagt i koden.
- Dersom loket flyttes på i DeskSim, flytter det seg i VR miljøet.

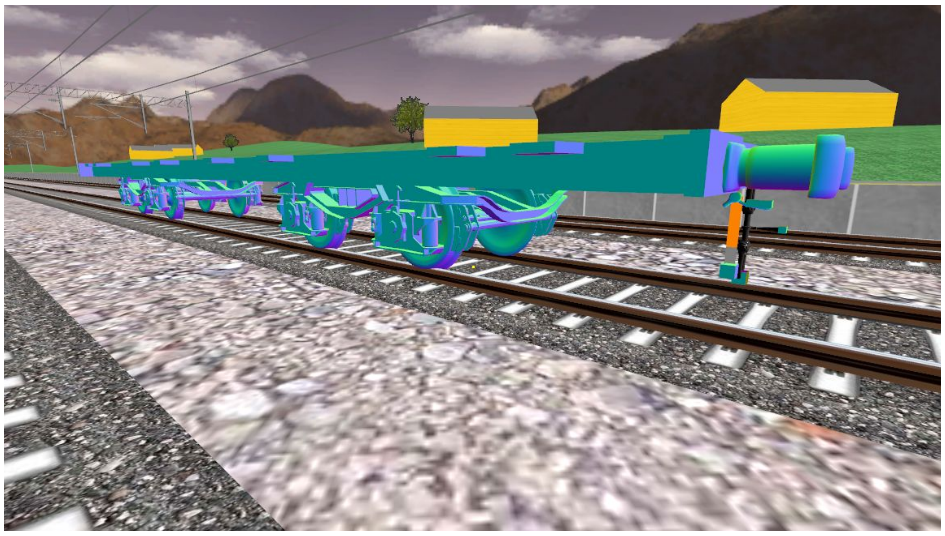
- Påbegynt at dersom man flytter seg eller utfører ting i VR miljøet ses dette fra DeskSim. Ingen avatar enda.
- Ikke begynt på at handlinger som utføres gjenspeiles / utføres begge steder. Eksempelvis kobles lok og vogn henger de sammen begge steder, Endres et signalbilde endres det begge steder, legges en sporveksel over skjer det begge steder.

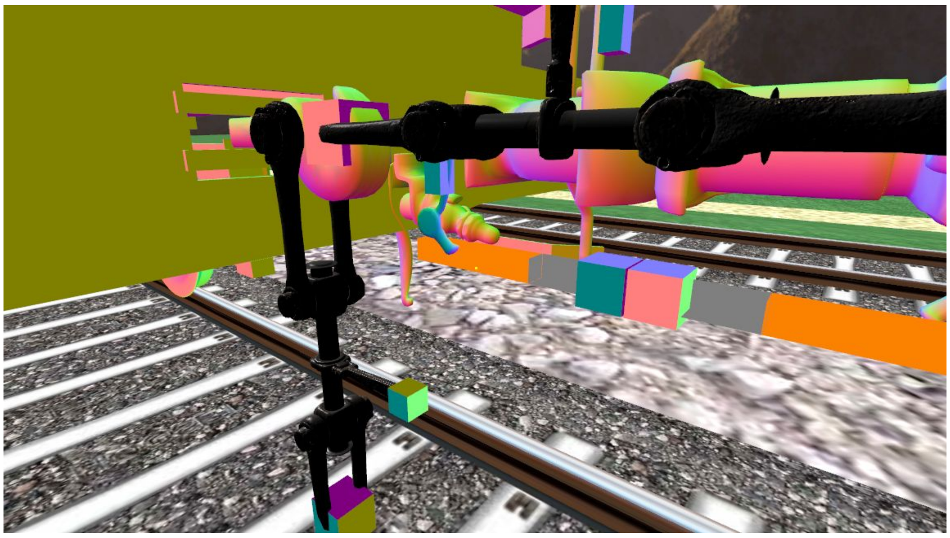
Kommunikasjon mellom skifter og fører (DeskSim og VR-miljø).

- Ikke påbegynt.

Modul 4, modeller

- Et nesten ferdig lok, mangler en del detaljer + tekstur
- En 6-aksla containervong som bare mangler tekstur
- En to-akslet vogn som nesten er ferdig
- Kroken mellom vognene er helt komplett
- Luftkran og -slange er modellert men mangler tekstur og delvis implementasjon.





D Meeting 1

Norsk fagskole for lokomotivførere

Referat Prosjektmøtemøte 6 Digiskift

Sted	Debriefrommet
Dato	10.01.2019 kl 11:00 – 13:00
Tilstede	Eirik Hiis-Hauge, Haakon Bjørnås Aarstein, Sindre Blomberg Garvik, André Gustavsen, Hallgeir Olsen, Atle Schaathun, Ottar Haslestad.
Fraværende	
Sider (inkludert vedlegg)	2

- Veileder fra NTNU? Hvor mye og hvordan involveres?
  Fått kontakt-info. Lokførerskolen tar kontakt

- Definering av oppgaven(e) + presentasjon av programmering (Hallgeir og André)

  Modul «sammenkobling» som basis. Eventuelt utvide med «fysiske egenskaper» Hallgeir presenterer litt om koder osv i programmet. Noe i dag og så mer etter hvert. Programmet er overlevert og Hallgeir presenterte kort om programmets struktur (på kontoret)
  Lokførerskolen skaffer tekniske tegninger på vogner.

- Lokasjon underveis.
  Mest mulig fra Gjøvik. På lokførerskolen ved behov.

- Hvor ofte «møtes» vi?
  En-to ganger i uka
  Definere milepæler etter hvert. Lokførerskolen vil få tilgang til Git og Trello.
  Kontrakt og plan med mål skal leveres NTNU 01.02.2019
  Hele Bachelor-oppgaven(rapporten) skal være ferdig 20.mai. Presenteres i juni (4.,5. eller 6.)

- Skype (eller annen måte)? «Appear.in» prøves først. Haakon sender Link. Lokførerskolen kan dekke betalingsversjon ved behov.

- Eventuelt
  Reiseutgifter: Haakon har alle kvitteringer (hittil)
  Lokførerskolen gir en kort «undervisningsøkt» i grunnleggende bremser godsvogner

Restanseliste

Tiltak	Ansvarlig	Frist
Kopiere programmet og gi veiledning (memory-stick)	Hallgeir	
Avklare vogntyper som skal modelleres	André	
Klargjøre program for «møter»	Atle og Haakon	
Undersøke/skaffe tekniske tegninger	Ottar	
Kontakte veileder NTNU	Ottar	
Sette seg inn i programmet	Eirik, Haakon og Sindre	
Undervisningsøkt: Grunnleggende forståelse av bremsesystemet	Atle	

E Meeting 2

Norsk fagskole for lokomotivførere

Referat Møte med studentene Digiskift 18.03.2019

Sted	Kjeller i A-bygg NTNU-Gjøvik
Dato	18.03.2019 kl 11:00 – 15:00
Tilstede	Eirik Hiis-Hauge, Haakon Bjørnås Aarstein, Sindre Blomberg Garvik, Hallgeir Olsen, Atle Schaathun, Ottar Haslestad.
Fraværende	Øivind Kolloen
Sider (inkludert vedlegg)	2

Formålet med møtet var å kvalitetsikre forståelsen av programmet til simulator.
I tillegg inngå en avtale mellom Jernbanedirektoratet og Bachelor-oppgave studentene ved NTNU. Denne avtalen gir rett til et stipend etter levert og godkjent oppgave samt rett til refusjon av utgifter ved reiser osv i forbindelse med oppgaven.

Hallgeir tok tog til Gjøvik og ankom ca kl 11:00. Demo av progresjon og forståelse samt eventuelle avklaringer rundt programmet var hovedmålet.
Hallgeir er godt fornøyd med nivået på leveransen dette stadiet i oppgaven. Han returnerte til Oslo med et tog som gikk fra Gjøvik ca kl 15:30.

Atle og Ottar ankom i bil ca kl 12:30. Det var avtalt møte med veileder Øivind Kolloen for en kort samtale og en underskrift på avtalen, men han var dessverre syk denne dagen.
Studentene fikk modelltogvognene som har ankommet lokførerskolen, men fortsatt har vi ikke tilgjengelig en RPS, tankvogn og Traxx-lok. Ottar har etterlyst kontaktperson hos Railpool for tegninger av lokomotivet.

Den nevnte avtalen mellom Jernbanedirektoratet og Bachelor-oppgave studentene ved NTNU ble underskrevet av studentene, en person med myndighet fra NTNU og Ottar Haslestad.
Studentene hadde noen spørsmål rundt detaljer om bremsesystem. Hallgeir og Ottar skisserte på et ark og forklarte prinsipper. Vi ble enige om at studentene formulerer spørsmål i en mail og Ottar skaffer svar på disse. Idéen om en kort video som et hjelpemiddel for å forstå bremsesystemet ble revitalisert.

Atle og Ottar returnerte fra NTNU ca kl 13:30

Restanseliste

Tiltak	Ansvarlig	Frist
Klargjøre program for «møter»	Atle og Haakon	
Undersøke/skaffe tekniske tegninger	Ottar og André	
Sette seg inn i programmet	Eirik, Haakon og Sindre	
Undervisningsøkt: Grunnleggende forståelse av bremsesystemet	Atle	

F Project Agreement

NTNU

Vår dato Vår referanse

Norges teknisk-naturvitenskapelige universitet

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

Norsk fagskole for lokomotivførere

_______________ (oppdragsgiver), og

Eirik Hiis-Hauge

Haakon Bjørnås Aarstein

Sindre Blomberg Garvik (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 1/2 - 19 til 20/5 - 19 .

   Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon/fax, reiser og nødvendig overnatting på steder langt fra NTNU på Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv hvis de har skriftlig karakter A, B eller C.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn): Øivind Kolloen

Oppdragsgivers kontaktperson (navn): Ottar Arne Haslestad

Student(er) (signatur): [illegible] dato 25/1-2019

[illegible] dato 25/1-2019

Eirik Hiis-Hauge dato 25/1-2019

_______________ dato _______

Oppdragsgiver (signatur): Ottar Arne Haslestad dato 25.01.2019

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): _______________ dato _______

NTNU
Kunnskap for en bedre verden