NTNU
Norwegian University of
Science and Technology

Software Application for Verification of Equipment Access

Christian Tverås

Master of Science in Engineering and ICT
Submission date: June 2016
Supervisor: Bjørn Haugen, IPM

Norwegian University of Science and Technology
Department of Engineering Design and Materials

Abstract

The goal of this project is to create a software application. The application will use dynamic relaxation to check the feasibility of transporting equipment along a predefined path. It is going to validate and optimize the predefined path and visualize the movement. When engineers checks for feasibility of a path, he or she does a manual check by moving and rotating the transportation object for every position. This application will simulate the movement and do the check. As an engineer has to use a lot of time with manual check, this application is more effective. A master thesis from 2014 worked on the same problem, but used an expensive software and used a time-consuming collision detection. The goal for this thesis is to develop a software that is open source, user friendly and free to use.

This application is a .NET windows application, programmed in C# and XAML. ParaView is a visualization software that visualizes the results of the verification process. Inputs are a 3D geometry model of the area, a path and a transportation object. It is normal to design a model of the workspace (like an offshore platform) in a 3D program. That is why the application has to read 3D model files and understand the file format to solve the problem. It also reads a path file that user has to create in ParaView. The user set the properties of the transportation object in the application. It simulates the movement through the geometry model, along the path, using dynamic relaxation. The results is an animation of the transportation object that moves along the path. It detect collisions with the geometry model and optimizes the path if it is possible.

A lot of work went to gathering theory about the dynamic relaxation algorithm and collision detection. These two algorithm was crucial for developing a functioning application with meaningful results. It was also a lot of work to understand how to represent the results for visualizing it in ParaView. ParaView is a software I have never used and I am quite new to data visualization. Gathering theory was therefore important to develop a functioning application.

Sammendrag

Målet med denne masteroppgaven er å lage et program som bruker dynamisk relaksjon for å verifisere og visualisere frakt av tungt utstyr gjennom trange områder. Det skal være mulig å validere og optimalisere en gitt sti og visualisere trallens bevegelse langs denne stien. I dag gjøres det som en manuell test, hvor ingeniøren må flytte og rotere trallen for å se om den kolliderer. Denne applikasjonen automatiserer denne oppgaven sånn at ingeniøren slipper å gjøre dette. Manuell verifisering er en lang prosess, hvor en feil eller endring i designet gjør at ingeniøren må starte på nytt. Ingeniøren sparer masse tid ved automatisere denne prosessen. En master fra 2014 jobbet med det samme problemet, men det programmet var dyrt og brukte en kollisjonssjekk som tok en del tid. Hovedmålet for denne oppgaven var å lage et program som er gratis, lett å bruke og tilgjengelig for alle.

Programmet er en Windows applikasjon skrevet i programmeringsspråket C# og XAML. For å visualisere resultatene brukes visualiseringsprogrammet ParaView. Inputdataene er en 3D-modell av et område, en sti og et transportkjøretøy. Det er veldig normal å designe en modell av et område før det lages (som for eksempelen en oljeplattform) ved hjelp av et 3D-program. Det er grunnen til at denne applikasjonen leser inn 3D-modellen fra fil hvor programmet må forstå filformatet for å løse problemet. Den leser også stien fra fil, som bruker selv må lage i ParaView. Bruker velger attributtene (lengde, bredde og høyde) i applikasjonen. Dynamisk relaksjon brukes for å simulere trallens bevegelse langs stien. Resultatene blir visualisert som en animasjon av trallen som følger stien. Kollisjoner mellom trallen og 3D-modellen blir registrert og en ny optimalisert sti blir definert om det er mulig.

Mye tid gikk til innhenting av teori om dynamisk relaksjon og om kollisjonsalgoritmen. Disse to algoritmene er viktige for å utvikle en funksjonell applikasjon som lager resultater som gir mening. Det gikk også mye tid til å forstå ParaView og hvordan dataen måtte representeres for å visualisere resultatene. Jeg har aldri brukt ParaView før og har lite erfaring med datavisualisering. Derfor var det viktig å innhente teori innenfor dette feltet for å skape visualisering som ga mening.

Preface

This master thesis is a part of the study program Engineering and ICT at the Department of Engineer Design and Materials (IPM) at Norwegian University of Science and Technology (NTNU), written spring 2016. Associate Professor Bjørn Haugen at NTNU is the academic supervisor during this project.

The master thesis will continue on the project work from previous semester. It is a new approach on a master thesis from spring 2014. Most of the techniques and calculations are the same, but nothing except the theory from the thesis have been accessible.

The goal of this project is to develop a software application that will make it easier to verify transport routes, for bringing equipment on a work site where access is tight.

Acknowledgment

I would like to thank associate Professor Bjørn Haugen for his support and help in guiding me through this master thesis. He has been a great help in giving me advices, ideas and sharing knowledge that might be interesting for my work. He followed my work weekly and kept me en route.

I also need to acknowledge Marius Hansen Røed who worked on the same problem earlier and put a lot of effort into it. His master thesis was the motivation for continuing on this problem. It is also important to mention my friends and family for being around, helping me out if necessary and supporting me.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter is an introduction to this Master thesis. It will describe the problem, objectives, motivation and the background of this project work. The introduction will explain the task, its current problems and how this thesis might solve the problem.

1.1 Background

Many jobs involves handling and transportation of heavy equipment. Work area is often cramped and verification of equipment access is important. Knowing in advance if it is possible to transport the equipment to the required destination, is both cost and time efficient.

The goal of this project is to create a software application in .NET. It uses dynamic relaxation to check the feasibility of transporting the equipment. Dynamic relaxation is a numerical method that consist of a set of equations to find equilibrium of all forces in a pseudo-dynamic system. It validates the path during the movement and checks for collisions. The application changes the path if the transportation object collides. A visualization tool shows the results as an animation of the transportation movement. This could save a lot of time and be valuable by allowing the engineer to run several iterations with a low cost.

A master thesis from fall 2014 [15] worked on the same problem. Some of the theory is similar like using dynamic relaxation to solve the problem. Only the research from that master thesis was accessible, and some of it will contribute to the software application. The application will use dynamic relaxation as well, but focuses on not using any expensive external software. The project work previous semester was about the same problem as this thesis. It focused more on research, understanding the problem and finding different ways to solve the different implementations.

1.2 Objectives

A set of list with objectives involved in this project are

1. Document theoretical basis for the application.
2. Define the input parameters for the core numerical algorithm of the application.
3. Implement quasi-static movement of the equipment along a predefined path.
4. Presentation of closest possible path and possible failed path.
5. Define a workflow for the modeling process geared towards use of the developed algorithm.
6. Develop a user-friendly application, which visualizes and verifies access of equipment and access ways.

Gathering research has been important to create this application. This document consist of research on how to represent input data and how to use ParaView to visualize the movement of the trolley. It also explains dynamic relaxation and the calculations and theory behind. There are also some explanations of the most important implementations for the application.

1.3 Continuation on previous work

Marius Hansen Røed wrote the master thesis ”Verification and visualization of equipment access on offshore platforms” during fall 2014 [15]. I also worked on this task during my project work. My focus was then on gathering theory and testing out different technologies. Some time went to testing out ways to solve some of the implementations as well. Marius made a plugin application in Navisworks during his work. Navisworks is a project review software that review integrated models to gain better control over project outcomes [1]. Some of the problems with Navisworks though is that it is quite expensive. It costs an annual fee of 2.255 for the version including the necessary functions for this plugin. One of the reasons he used Navisworks for the plugin was because of a collision detection feature. A problem was though, that the collision feature was time consuming and slowed down the plugin [15]. This is some of the reasons for the new take on this software application. It also means that some of the theory behind is the same, as well as the calculations.

1.4 Manual verification

Engineers has to do a manual verification, to check for equipment access along the given path. They first design the model given some criterias based on the NORSOK (the Norwegian shelfs competitive position) standards. Then they import the model from PDMS

(Plant Design Management System), or some other 3D modelling software, to a software to review the design. They use this software to validate the transportation path from start point to destination. The engineers have to move and rotate the transportation vehicle and checks if it collides with the environment by using their own eyes only. He or she changes the rotation and translation for each step of the movement. They have to repeat the whole verification in some cases. Either if the transportation object collides, if there are some design changes or if the size of the transportation object changes. This makes the whole process a time consuming and tiring job.

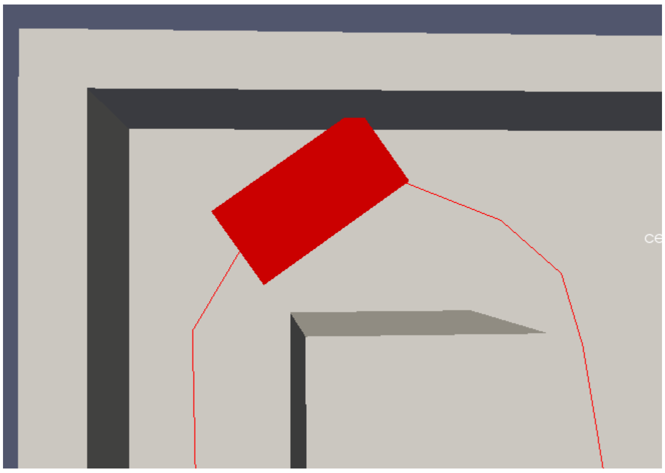


Figure 1.1: Vehicle collides with a wall

Figure 1.1 shows an example of a transportation vehicle colliding with a wall. This is just one of many steps an engineer have to check and validate. The engineer has to either redesign the model or try another movement and rotation to get past the corner when it collides.

1.5 Structure

This is a small explanation about what each of the chapters is about:

Chapter 1 - Introduction:

> Consist of the background of the problem, the motivation and previous work of the same type of work.

Chapter 2 - Input parameters:

> What type of parameters there are and how the application initialize and uses them.

Chapter 3 - Visualization:

Different visualization parts of this application from file formats visualizing different models to actual visualization ways.

Chapter 4 - Dynamic relaxation:

Calculations and theory of dynamic relaxation. How to solve the verification problem with dynamic relaxation.

Chapter 5 - Implementations of application:

Different implementation parts of the application.

Chapter 6 - Conclusion:

Conclusion with a summary and ideas for further work.

Chapter 2

Input parameters

This chapter will look at the input parameters of this project and how they are initialize and used.

2.1 Input Parameters

The transportation object should follow a path through a work environment. This application are going to create a visualization of the movement and create an automatic verification check. The environment consist of a transportation object that follows a path, trying to avoid the obstacles of the 3D geometry. There are three input parameters: A model of the work area, the transportation path and a transportation object. This chapter will look more at these three parameters.

This is a standalone application and will not use any applications to do the calculations, reading and writing the input parameters. It uses ParaView to visualize the results, but it calculates the results before opening it in the visualization tool.

2.1.1 Geometry model

The work area is a 3D model of the surroundings. It consist of walls and other objects that can work as boundary restrictions for the transportation object. The goal of the whole application is to verify that the model design is suitable for the given transportation path. That is why this is maybe one of the most important input parameters. Managing collision detection depends on the representation of the geometry model in the application.

The engineer usually creates the 3D models with PDMS, or some other 3D CAD (Computer-aided design) application. STL files (STereoLithography) has a common file structure, and is a fundamental and much used file format. It present the data as vertices sorted in vectors of triangles. This file format is possible to visualize in ParaView, as well as reading the data in the application. Figure 2.1 shows an example of a simple model opened in ParaView. It shows the surface made of triangles that the STL file consist of. Later on in the thesis is more information about STL files and the usage of it in the application.

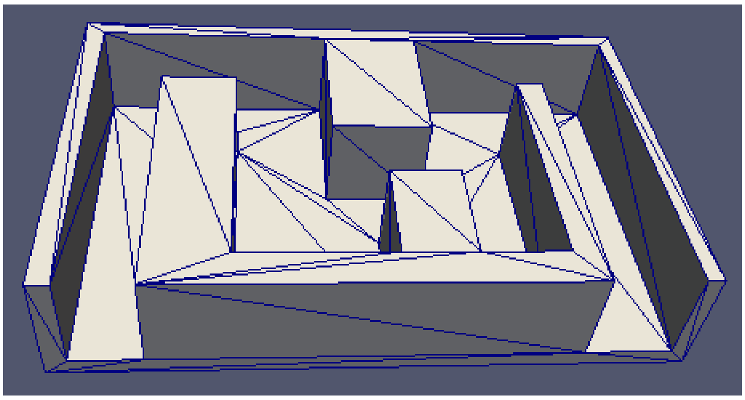


Figure 2.1: A simplified test example of a geometry model

2.1.2 Path

A path is necessary to validate the transportation. The path is crucial when using dynamic relaxation as validation method, as the trolley has to follow a continuous path. The path should be easy to create, so that validating a geometry model is effective.

Some of the restrictions is that the path has to be continuous and consists of multiple nodes. It needs a fixed start and end position. Corners should consist of arcs, or nodes close together, to get a more realistic movement, but this is up to the engineer to choose. It is important that creating a path is easy, with as few clicks as possible and easy to open in the application. The user should have the possibility to choose the number of nodes in the path and the distance between them.

This application reads the path of points as a VTK file (Visualization ToolKit), which is a file format that is easy to read. It is possible to use ParaView to create the VTK file. A Poly Line Source creates a line of nodes. It is possible to click and drag each node to change the position or write in the coordinates. Figure 2.2 shows a path line created in ParaView. The points is not visible when opening vtk files with the path in ParaVew as seen at the figure.

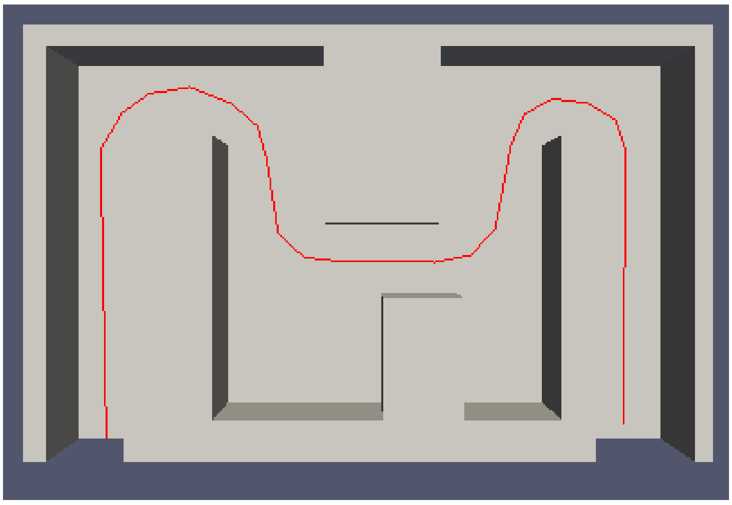


Figure 2.2: Example of a path made out of multiple nodes

2.1.3 Object

The transportation object is usually a trolley or some other type of transportation vehicle. A transportation object might have different paths, as the geometry of the trolley changes. The geometry is during this case simplified. The object is a rectangular object consisting of vectors as sides and nodes as edges. This makes it easier to check for collision between the geometry model and transportation vehicle. It also is easier to initialize and create the visualization of its movement. It needs a center that should follow the nodes of the path. This center could change depending on the transportation vehicle, but will in this case just be in the middle of the rectangular object. It should be possible to customize the object by choosing size and mass. Another possibility is that the user chooses from a selection of different objects.

2.1.4 Workflow

Workflow of the path verification shows a detailed process flow in the application. It starts with opening the different input data. The idea is that it will not matter what order the application opens and reads the input data. There are two options: verifying or optimizing the path. The user has to pick at least one of these. The workflow show how optimization of the path has to run if not running verification. It writes and saves the files after finishing the simulation.

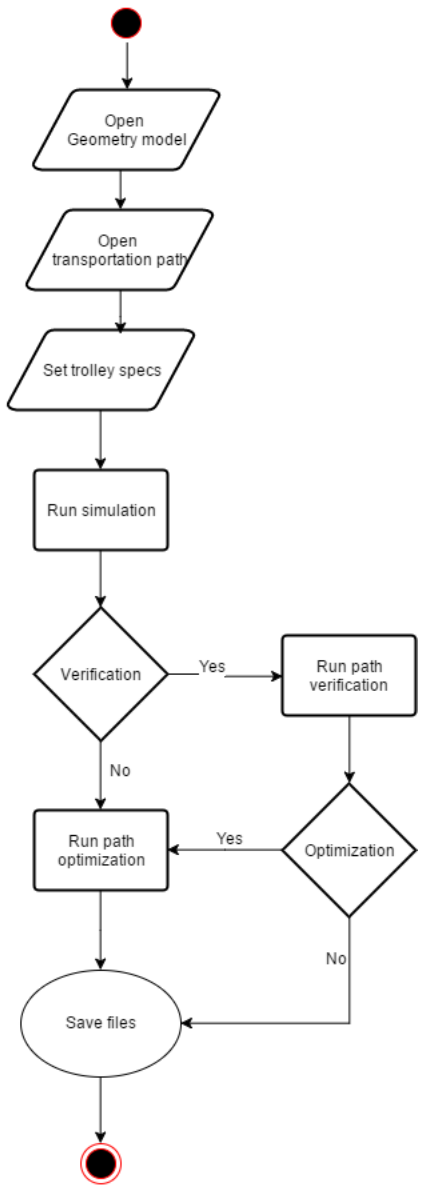


Figure 2.3: Workflow of the path verification

Chapter 3

Visualization

This chapter will look at the visualization part of the project. This will go in depth of the STL file format, VTK format and ParaView.

3.1 STL file format

An STL file is a triangular representation of a 3D surface geometry [11]. The file format is a common CAD-related file, with a mesh of many triangles. Each triangle consist of three vertices and a normal vector representing the direction. These triangles represent the geometry, by following a set of rules.

3.1.1 Format specifiactions

Each of the vertices in a triangle consist of x-, y- and z-coordinates. Each triangle consist of 12 values. Three vertices, with coordinates for each, and a normal vector (a vector perpendicular to the surface with length one).

Two things decides the orientation for each of the triangles. The first thing is the unit normal vector, which is always facing outward of the surface. The other thing is the order of the vertices listed in the file format. This order follows a counterclockwise order, outward of the surface.

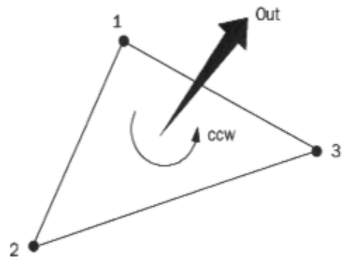


Figure 3.1: A triangle showing direction of the unit normal and order of vertices [11]

There are some rules on how the triangles form the geometry. Every triangle joined together has to share two vertices. This means that one vertex from a triangle cannot be between of two other vertices of another triangle. Figure 3.2 show examples on how violating and following this rule looks like.

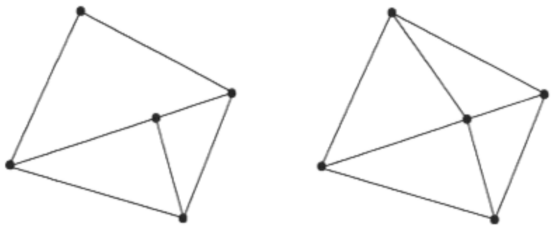
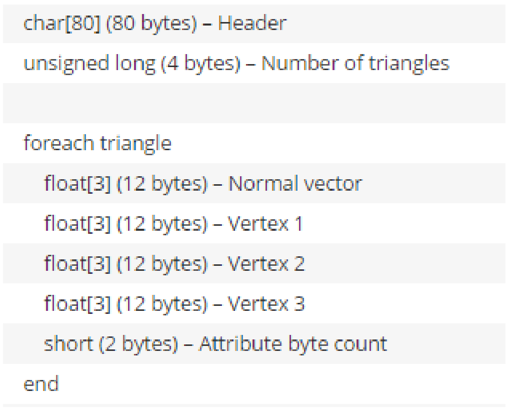


Figure 3.2: Figure to the left is a violation while picture to the right show a correct mesh representation [11]

3.1.2 File Formats

Every file is either an ASCII file (American Standard Code for Information Interchange) or binary file. ASCII files are not common for STL files because of the size of the file, but gives a clear overview of the format of the file. The representation of the file format is clear and just as explained before. Binary files stores the values as binary numbers while ASCII files consist of normal characters.



```
char[80] (80 bytes) – Header
unsigned long (4 bytes) – Number of triangles

foreach triangle
  float[3] (12 bytes) – Normal vector
  float[3] (12 bytes) – Vertex 1
  float[3] (12 bytes) – Vertex 2
  float[3] (12 bytes) – Vertex 3
  short (2 bytes) – Attribute byte count
end
```

Figure 3.3: An overview of the binary file format [10]

Figure 3.3 shows the structure of an STL file. It start with a header of 80 bytes, and then 4 bytes of the number of triangles in the file. Each triangle follows, consisting of 50 bytes each. Every value has a size of 4 bytes, so each vertex, with x-, y- and z-coordinates is 12 bytes. It is not normal to use the last 2 bytes of these 50 bytes.

3.2 VTK (Visualization Toolkit)

VTK [13] is a software system for 3D computer graphics, modeling, image processing, volume rendering, scientific visualization and information visualization. It is a C++ toolkit, but also supports wrappers in Python, C# and Java among others.

The origin of VTK was the textbook ”The Visualization Toolkis An Object-Oriented Approach to 3D Graphics” in 1993. Three graphics and visualization researchers worked on it on their own time. The motivation of the book was to collaborate with other researchers to create an open framework for creating leading-edge visualization and graphics applications. Users and developers around the world improved and applied the system to other problems after they finished the core of VTK. The three founders left their jobs and founded Kitware Inc. in 1998, to support the huge VTK community.

3.2.1 Data model

The fundamental data structure in VTK is data objects. These data objects is either datasets, such as different grids and meshes, or graphs and trees. Each data objects consist of mesh and attributes. The main datasets is image data, rectilinear grid, structured grid, polydata and unstructured grid.

Building blocks

Some abstractions are the same for everyone, even though the data structure of a mesh depends of the type of datasets. A mesh usually consist of vertices and cells. A cell consist of multiple vertices and maps a region or surface in different ways.

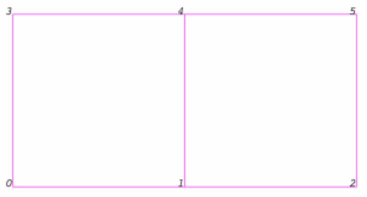


Figure 3.4: Mesh structure of 2 cells from 6 vertices [14]

Attributes are discrete values that are not a part of the mesh. These attributes are for example pressure, temperature, velocity or mesh color. Data arrays with a number of components stores the attributes.

Datasets

In the beginning of this chapter is the names of the different types of datasets and grids. Uniform rectilinear grids (image data) consist of multiple cells of the same type, dimension and size. This is the most normal type of dataset and takes the least size of the datasets. This is why most algorithms in VTK takes advantage of this kind of grid. Adaptive Mesh Refinement (AMR) datasets are almost the same, except that it consist of a collection of multiple image datas. Rectilinear grids consist of cells with the same geometry but with different size. Curvilinear grids (structured grids) consist of different geometry and size. These are the structured grids and uses the least memory.

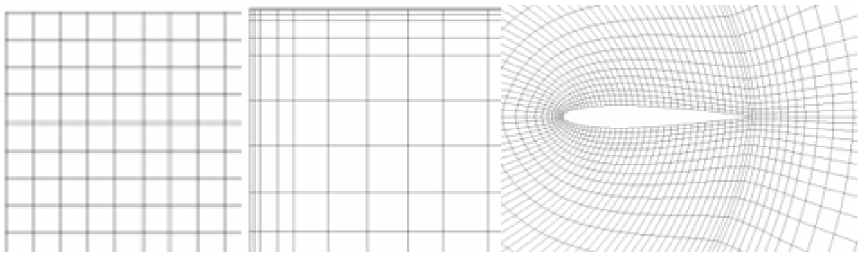


Figure 3.5: Uniform rectilinear grid, Rectilinear grid and Curvilinear grid [14]

An unstructured grid is the most primitive dataset type. This kind of dataset uses more memory, which is why other datasets should represent the data if possible. It may consist of all the cell types VTK supports, in different dimensions and sizes. Polygonal grid (polydata) is a more specialized version of the unstructured grid. Unstructured grids works as a more efficient rendering option and consist of all cell dimensions except 3D cells.

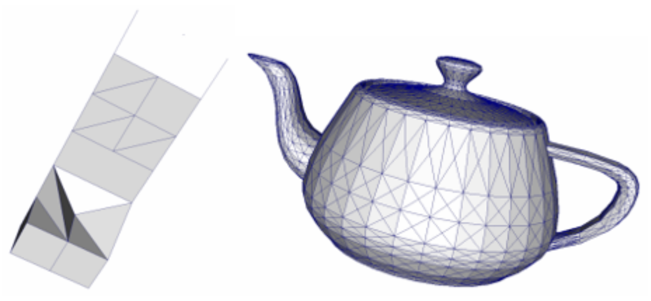


Figure 3.6: Unstructured grid and polygonal grid [14]

3.3 ParaView

ParaView [7] is an open-source data analysis and visualization application. Users can analyze large datasets through visualizations by using different types of techniques. It is a general-purpose, end-user application you can run on your computer or on remote parallel computing resources. It is also a framework with different tools and libraries for applications, such as scripting, web visualization and on site analysis (in-situ analysis).

ParaView was a collaborative effort between Kitware Inc. (who also made VTK) and Los Alamos National Laboratory in 2000. Kitware Inc. also started working on its own independent web based visualization system, which contributed to the development of ParaView. By collaborating with many other governmental and academic institutions. One of the goals was to create a user-friendly, open source, flexible and intuitive user interface. It is possible to open in a lot of different visualization files (STL and VTK is just a few), which makes paraView flexible to use in almost any setting. This has been one of the thing focused on during development of new versions. Almost 100 000 users download ParaView every year. It won the HPCwire Readers choice award in 2010 and 2012, and editors Choice award for best High-performance computer visualization product of technology in 2010.

This chapter will look at how ParaView works and give an intro to how to use it and its tools. It is important for understanding how the application will use paraView to visualize the given data. This part also gives an idea on how some of its technologies is possible to use in further development of this application.

3.3.1 Basics

Paraview has a tutorial that looks at most of the basic functionalities of the software [8]. ParaView uses VTK for the basic visualization and rendering algorithms. The graphical user interface (GUI) is just a small part of the functionalities in paraView, with many libraries available. This makes it possible to create your own GUI. It also comes with a Python interpreter (pvpython) where you can automate visualization and post-processing data with python scripts. Pvbatch is just the same as pvpython, except that it is not using paraView. Pvpython takes commands from input scripts and run it in parallel if it uses the MPI (message passing interface). Pvpython and pvbatch takes exactly the same inputs. There is also other executables for running on server and doing remote visualization. ParaView is flexible for customization.

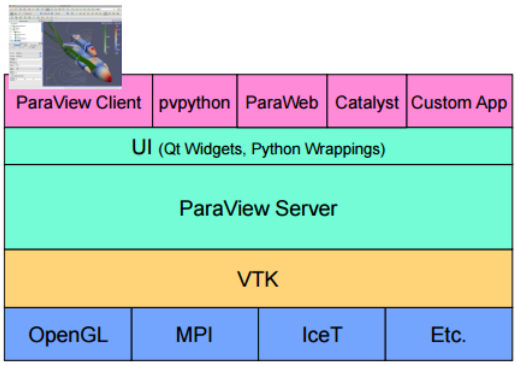


Figure 3.7: The ParaView system [8]

It uses a data-flow paradigm (a pipeline structure), which means that data flows through different modules of algorithms, where the output of each element is the input of the next. There are three different types of algorithms. First type is sources, which create data into the system such as readers that read data from files. Filters is algorithms that process datasets and generate, extract or derive features from the data. The last is sinks that saves the data with writers. ParaView includes many readers and writers that makes is possible to use all kind of file formats.

The GUI consist of a menu bar and toolbars as almost every other programs, with access to almost every features in ParaView with shortcuts to the most used features. It also consist of a 3D view to visualize the data, a pipeline browser and a property panel. The pipeline browser show the structure and makes it possible to select each pipeline object. The property panel allow you to view and change the parameters of the current pipeline object. It is also possible to add and remove various GUI elements.

Paraview visualize and analyze big datasets. It will not render anything in the 3D view when creating an instance. Since sources (and filters) has parameters, it waits to apply the data on the 3D view so that the user may change any of the parameters. It is an apply button to render the view, as ingesting big data into the system can be a time-consuming processes.

3.3.2 Scripting in ParaView

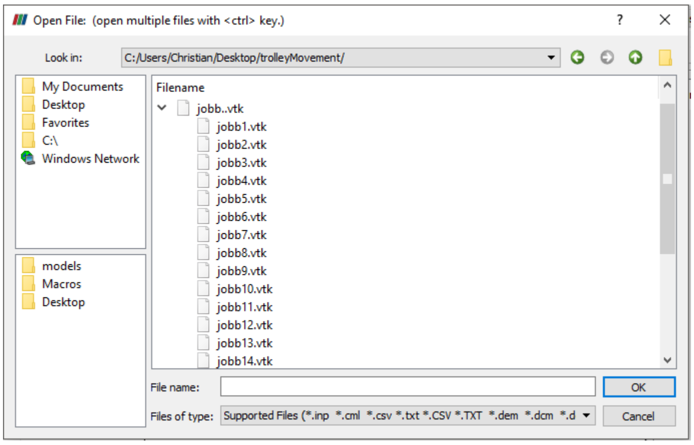
ParaView comes with a python interpreter through pvpython, as mentioned before. It comes with python packages that provide functionalities in ParaView. This makes it possible to create python scripts to visualize and analyze what you need without using the GUI. Yet, the GUI also comes with a python shell (Tools - Python Shell) to run scripts and python commands there as well. The GUI interface will update after every commands when writing commands in the shell.

Tracing

ParaView has an option to trace all your actions in the GUI (Tools - Start Trace), and creates python scripts that do the same actions. When you stop tracing (Tools - Stop Trace), ParaView opens an editor window with the generated python code. This is a great way of learning the basics of the python packages, which comes with ParaView. It is also a nice way of automating actions you do in ParaView many times to be more efficient.

3.3.3 Temporal file series

ParaView handles time in different ways depending on the file format [14]. Several file formats makes it possible to save the time steps in one file, while others save each step in a file series (with each time step as a new file). ParaView recognizes time series in the Open File dialog and shows them as a grouped element with the filename ending with "..vtk" (see figure 3.8). It is also possible to open one individual file from the file serie if necessary.



Look in: C:/Users/Christian/Desktop/trolleyMovement/

Filename
jobb..vtk
jobb1.vtk
jobb2.vtk
jobb3.vtk
jobb4.vtk
jobb5.vtk
jobb6.vtk
jobb7.vtk
jobb8.vtk
jobb9.vtk
jobb10.vtk
jobb11.vtk
jobb12.vtk
jobb13.vtk
jobb14.vtk

File name:

Files of type: Supported Files (*.inp *.cml *.csv *.txt *.CSV *.TXT *.dem *.dcm *.d

Figure 3.8: An example of a temporal file series in ParaView

Chapter 4

Dynamic Relaxation

The calculations is almost the same as in the master thesis of Marius Hansen Red. This makes it unnecessary to start all over again with this topic. That means that this chapter will cover some theory from the master thesis Verification and visualization of equipment access on offshore platforms [15]. The chapter will explain the method, as well as giving a brief explanation on how to solve the verification problem.

Dynamic relaxation is a numerical method that consist of a set of ordinary differential equations to find equilibrium of all forces in a dynamic system. Form-finding is one of many ways of using dynamic relaxation [3], which finds a geometry where all forces are in equilibrium. The geometry consist of many nodes, each with a mass. An applied force will lead to oscillation in the system, which will always try to maintain equilibrium. Simulations of this dynamic process checks through all the iterations and looks at the geometry in all time steps during the external loads.

4.1 Method

Dynamic relaxation originated from the equation of motion, which for a system like in figure 4.1 is

$$\mathbf{P}_{ij} = [\sum K\delta]_{ij} + \mathbf{M}_{ij}\ddot{\delta}_{ij} + C\dot{\delta}_{ij} \tag{4.1}$$

$\mathbf{P}_{ij}$ is the external load vector while $[\sum K\delta]_{ij}$ and $C\dot{\delta}^n_{ij}$ is the internal load vectors. K is the node stiffness while δ is the displacement. C is the damping coefficient, $\ddot{\delta}$ is the acceleration and $\dot{\delta}$ is the velocity. ij refers to node number i in direction j.

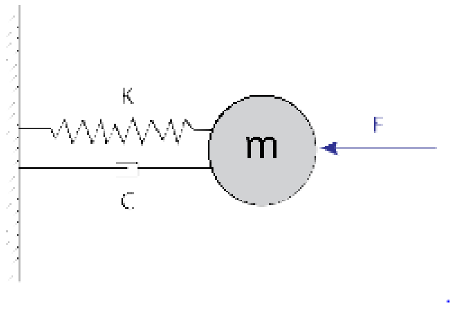


Figure 4.1: A dynamic system with a mass m, connected to a spring with a stiffness K, a damper with a coefficient C and an external force F [15]

By introducing the residual force, which is the difference between the external ($\mathbf{P}_{ij}$) and the internal forces ($[\sum K\delta]_{ij}$ and $C\dot{\delta}^n_{ij}$), we get that

$$\mathbf{R}_{ij} = \mathbf{P}_{ij} - [\sum K\delta]_{ij} - C\dot{\delta}^n_{ij} \tag{4.2}$$

This gives us a residual force equation after substituting equation 4.1 into equation 4.2.

$$\mathbf{R}_{ij} = \mathbf{M}_{ij}\ddot{\delta}_{ij} \tag{4.3}$$

The equation above results in equations for acceleration and velocity for each iteration. these equations calculates the displacement at next time step (n+1).

$$\ddot{\delta}^{n+1}_{ij} = \frac{\mathbf{R}^n_{ij}}{\mathbf{M}_{ij}} \tag{4.4}$$

$$\dot{\delta}^{n+1}_{ij} = \dot{\delta}^n_{ij} + \ddot{\delta}^n_{ij}\Delta t \tag{4.5}$$

$$\delta^{n+1}_{ij} = \delta^n_{ij} + \dot{\delta}^n_{ij}\Delta t \tag{4.6}$$

The system is approximately in equilibrium when these equations leads to $\mathbf{R}_{ij} \rightarrow 0$.

Mass, stiffness and time increment has to be set so that the method is realistic, as these assumptions is important. The mass needs to be realistic compared to the stiffness. The damping coefficient is set to be critical damping, which makes the system return to equilibrium the fastest possible way.

$$C_{critical} = 2\sqrt{mK} \tag{4.7}$$

4.2 Solving the Verification Problem

When the transportation object is following the path, the object moves from one path node to the other (see figure 4.2). This movement is one iteration. The dynamic relaxation method verifies every step of the path. Modification of the path will occur when a collision occurs if possible, so that the new path do not involve any collisions.

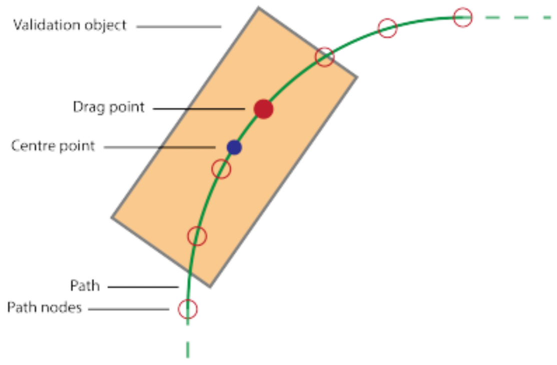


Figure 4.2: Representing how the transportation object follows the path [15]

Between the center point of the transportation object and the drag node, is a damper (see figure 4.3). The distance between them is the displacement δ. Velocity $\dot{\delta}$ and acceleration $\ddot{\delta}$ is a result from δ and the damping coefficient C. The transportation object moves closer to the drag node during each time step. This method will continue to the next iteration (drag node will move to the next path node), when the system reaches equilibrium (Residual force $\mathbf{R}_{ij} \to 0$).

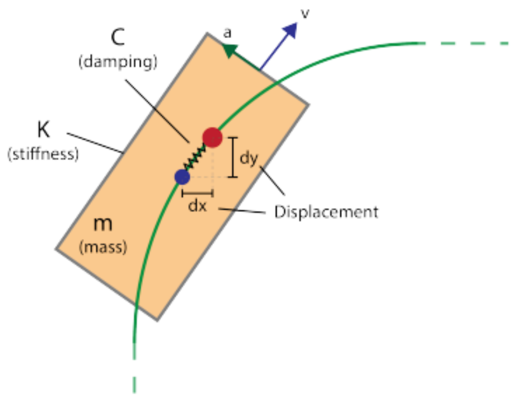


Figure 4.3: A damper between the center point and the drag node [15]

Collision with the geometry model adds an external force that pushes the transportation object back out of the model. The force creates unbalance in the system, where the dynamic relaxation method tries to push the system back into balance. It will return false if the system does not reach equilibrium after a given number of iterations (largest number of iterations to reach equilibrium). This means that it is not enough space for the object to move through this area of the geometry model. The system adds the new node position and then drags the node to the next path node, if it reaches equilibrium ($\mathbf{R}_{ij} \rightarrow 0$).

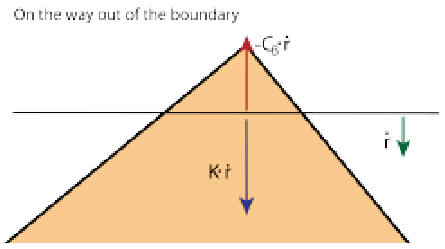


Figure 4.4: Transportation object while colliding, with the velocity vector $\dot{r}$. Damping opposite direction of the applied force. Applied force and damping has same direction in the moment it collides (velocity direction into the geometry model).[15]

Chapter 5

Collision detection

This chapter will look at collisions between the transportation object and geometry model. Detecting collision depends on how the application handles the input parameters. Chapter 3 contain theory about the STL format as representation of a CAD model. This chapter looks at collisions between these triangles and the trolley, and how different techniques creates an effective collision detection for larger datasets.

5.1 Line-Plane intersection

There are different ways to calculate the intersection between a plane and a line. Some of the reason for this is because it is possible to represent lines and planes in different ways. It also depends on the constraints and the inputs available. In this case, a plane consist of three point and a normal, and the line of two points (corners of the trolley). Here is two ways to solve the line-plane intersection problem.

This first example [4] only need one point in the plane V_0 and its normal vector $\mathbf{n}$, and the vector line of two points. The parametric equation of the line is

$$P(s) = P_o + s(P_1 - P_0) = P_0 + s \cdot \mathbf{u} \tag{5.1}$$

Where $s \cdot \mathbf{u}$ is the vector from P_o to a point on the line. There are three relations between the line and plane. These are that the line is parallel to the plane, the line is in the plane or that it intersects. The first check is if the line is perpendicular to the normal vector which it is if $\mathbf{n} = 0$. If this is not true then the line is not in the plane either. By adding vectors to find a vector from the point V_0 to a point on the line the equation is

$$P(s) - V_0 = \mathbf{PV} = \mathbf{w} + s \cdot \mathbf{u} \tag{5.2}$$

$\mathbf{w}$ is the vector from point on the plane to point on the line $(P_0 - V_0)$. Figure 5.1 shows an illustration on how the vectors added together gives another vector to the line. Direction of the line depends on s.

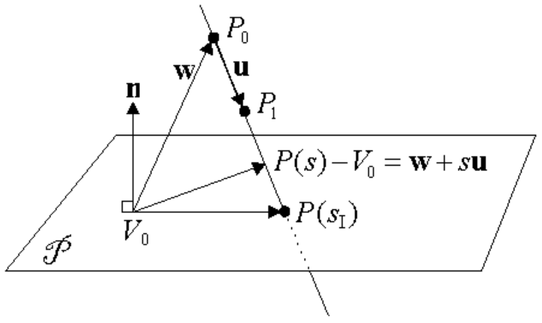


Figure 5.1: Vector representation of the intersection problem [4]

The line intersects on the plane when the vector $\mathbf{PV}$ is perpendicular with the normal vector $\mathbf{n}$

$$\mathbf{n} \cdot (\mathbf{w} + s \cdot \mathbf{u}) = 0 \tag{5.3}$$

Solving this on s show that the Line-Plane intersection happens when

$$s = -\frac{\mathbf{n} \cdot \mathbf{w}}{\mathbf{n} \cdot \mathbf{u}} = -\frac{\mathbf{n} \cdot (V_0 - P_0)}{\mathbf{n} \cdot (P_1 - P_0)} \tag{5.4}$$

If it is a finite line between P_0 and P_1, then it intersects only if $0 \leq s \leq 1$.

The second example [17] to check for line-plane intersection, uses three points in the plane and two points forming the line. The parametric equation 5.1 still represent the line. It is possible to look at this equation as three equations, one for each axis (x, y and z). There is also a fourth equation as a matrix representation of the plane

$$0 = \begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} \tag{5.5}$$

The intersection point is possible to determine by solving these four equations for x, y, z and t. Solving them gives the equation

$$s = -\frac{\begin{vmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 0 \\ x_1 & x_2 & x_3 & x_5 - x_4 \\ y_1 & y_2 & y_3 & y_5 - y_4 \\ z_1 & z_2 & z_3 & z_5 - z_4 \end{vmatrix}} \tag{5.6}$$

P_4 and P_5 is the points forming the line. The line intersects if $0 \leq s \leq 1$, as in the previous example. Plot s into equation 5.1 to find the intersection point for each of these examples.

5.2 Barycentric coordinates

Note that the line-plane intersection examples above only checks for intersection, but not if it intersects inside the given surface. There are a few ways to do this, but using barycentric coordinates is effective [9]. Barycentric coordinates are three numbers that are masses placed at each of the vertices in the triangle. These three numbers are the coordinate of the geometric centroid point [16]. If the triangle coordinates is A, B and C, the sum of the masses is

$$P = At + Bu + Cv \tag{5.7}$$

The sum of masses is equal to one when normalizing this equation. By substituting this normalized equation into 5.7 for t, the plane equation with two unknown is

$$P = A + (B - A)u + (C - A)v \tag{5.8}$$

This makes every other points in the triangle relative to point A. Every coordinate on the line $\mathbf{AB}$ give $0 \leq u \leq 1$ and $v = 0$, which represent every point on the line. The same apply to the line $\mathbf{AC}$, where $0 \leq v \leq 1$ and $u = 0$ does the same. Negative u and v shows that the point is on the wrong side of point A and is outside of the triangle surface. When $u + v = 1$, the point is on the line segment $\mathbf{BC}$. Finally this means that a point P is on the surface inside the triangle when $0 \leq u \leq 1$, $0 \leq v \leq 1$ and $u + v \leq 1$.

Since the line-plane intersection finds the intersection point from the value t, finding u and v makes it possible to figure out if this point is on the surface. It is possible to create two equation of 5.8 and solve for the two unknowns u and v. Creating a variable for each of the vectors simplifies the equation 5.8.

$$\mathbf{AB} = B - A \tag{5.9}$$

$$\mathbf{AC} = C - A \tag{5.10}$$

$$\mathbf{AP} = P - A \tag{5.11}$$

By solving the two equations for u and v the equation is as following

$$u = \frac{(\mathbf{AB} \cdot \mathbf{AB})(\mathbf{AC} \cdot \mathbf{AP}) - (\mathbf{AC} \cdot \mathbf{AB})(\mathbf{AB} \cdot \mathbf{AP})}{(\mathbf{AC} \cdot (\mathbf{AC})(\mathbf{AB} \cdot \mathbf{AB}) - (\mathbf{AC} \cdot \mathbf{AB})(\mathbf{AC} \cdot \mathbf{AB})} \tag{5.12}$$

$$v = \frac{(\mathbf{AC} \cdot \mathbf{AC})(\mathbf{AB} \cdot \mathbf{AP}) - (\mathbf{AC} \cdot \mathbf{AB})(\mathbf{AC} \cdot \mathbf{AP})}{(\mathbf{AC} \cdot (\mathbf{AC})(\mathbf{AB} \cdot \mathbf{AB}) - (\mathbf{AC} \cdot \mathbf{AB})(\mathbf{AC} \cdot \mathbf{AB})} \tag{5.13}$$

The point P is on the line if $0 \leq u \leq 1$, $0 \leq v \leq 1$ and $u + v \leq 1$, as mentioned earlier.

5.3 Clipping

Clipping is a typical technique in computer graphics. It removes objects, lines or line segments that are outside of a rectangle [12], such as a canvas view. It can also determine if it is possible to avoid intersection calculations [2]. This is necessary for bigger datasets like collision detection with many walls, to create effective algorithms.

5.3.1 Cohen-Sutherland algorithm

Cohen-Sutherland Line-clipping algorithm divides a 2D area into a grid of 9 rectangles. The middle rectangle is the geometry that needs a check for intersection. Every region get a four-bit code representing true (1) or false (0) depending on the position of the regions around the middle rectangle. Each bit shows if the rectangle is above, underneath, to the left and to the right of the middle rectangle.

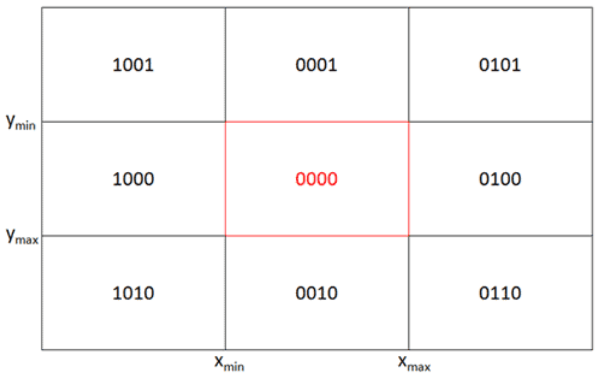


Figure 5.2: Nine segment grid each with a four-bit code [5]

This test makes it possible to check if there will be any line intersection with the middle rectangle. The line will not intersect if the whole line is either above, underneath, to the left or right of the middle rectangle. Figure 5.3 shows an example where line CD will not intersect as both C and D lay above the rectangle. An intersection test will check if the other lines intersect, as the Cohen-Sutherland algorithm does not rule out these lines.

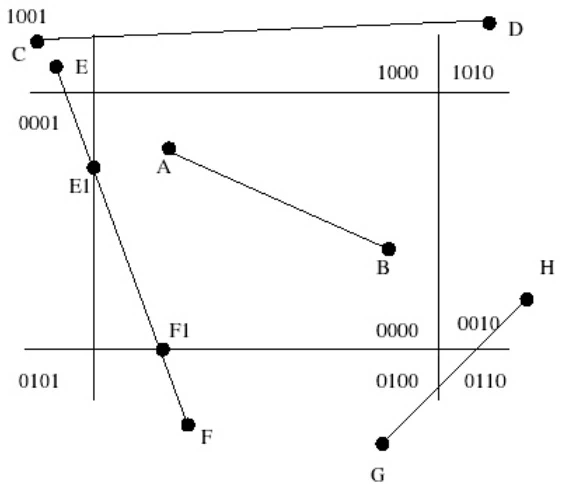


Figure 5.3: Example with different lines where some intersects with the middle rectangle [5]

Clipping polygons

Cohen-sutherland polygon-clipping algorithm [2] is finding every line or parts of a polygon that intersects inside a rectangle. The algorithm checks for four different cases of intersection between the vertices.

- Case 1: Whole line inside the rectangle, where it saves the endpoint
- Case 2: Exit rectangle, and save the intersection point
- Case 3: Everything outside, and saves nothing
- Case 4: Enter the rectangle, and saves the intersection point and endpoint

Following these cases gives a full picture of everything that intersects and are inside the rectangle.

Chapter 6

Implementation of application

This chapter will explain how some of the main implementations of this application uses the theory found. There are also some explanations on how the application deals with movement of the transportation object. The application is a .NET application written in C# and XAML (Extensible Application Markup Language). Everything except of the GUI components is in C#.

6.1 Reading STL files

Figure 3.3 shows the structure of a binary STL file and is how the implementation looks like. A binary reader opens the file in C#. The header is not important and skipped, while the number of triangles helps to iterate through every triangle. A list of vectors stores the normal vector and all the vertices of the triangles. The list consist of a tuple with the normal vector and a list of the triangle points.

6.2 Reading Path

The application reads VTK files of the path in ASCII format only. You should use ParaView when creating the path as it is the easiest option and since you have to use ParaView to visualize the job anyway. Corners of the path should be arcs so that the movement is as realistic as possible, but this is not possible when using dynamic relaxation. Nodes close together is thus the only way to make movement like an arc.

The headers is pre-defined, the structure has the polydata section with all the points first and then it defines the order of the points. As the order of the point is from zero and up the last part is not important for reading the path. Reading the points from the file, when everything else is pre-defined, is easy itself.

6.3 Transportation object

A new instance of the transportation object class defines the object and all its movement. It initializes the parameters from width, length, height, mass, a start position and the point it faces in the beginning. Center point of the trolley is set as the start position, while calculations from the pre-defined stiffness variable and mass finds the damping coefficient of the trolley. It sets the position of all the box edges with the width, length and height from the center point, before rotating them facing the right direction.

There are two types of movement for the trolley, and that is rotation and translation. The trolley faces the next node in the path, where a function consisting of a matrix transformation rotates the edges of the box around the center. This transformation only rotates the trolley around the z-axis only, as it did not work for rotations around more than one axis. This means that it completely works for cases where the trolley moves in the xy-plane, but not in other planes. To implement rotations among many planes, quaternions is the best way to go. For every step of the object movement, a function updates the position of the center point and the points forming the box. It updates the position with the distance it moves during one step of the dynamic relaxation algorithm.

6.4 Dynamic Relaxation

To iterate from one position to the next, every internal variable in the algorithm is set to zero, while finding the displacement from the current position of the transportation object to the next node in the path. A while loop runs while the residual force is not close to zero. The residual force (equation 4.2) is equal to collision force minus internal force minus the damping force (this depends on the directions of the forces). This force increases as the internal force is big and the damping force is small. Velocity inceases as the residual force increases, which increases the damping force as well. The internal force decreases as it closing in on the next path node. This decreases the residual force, the velocity and the damping force. The new acceleration (equation 4.4), velocity (equation 4.5) and displacement (equation 4.6) updates trolley position with the change of displacement before doing the same calculations over again in the loop. When the collision detection discovers a collision, it finds a force to push the trolley back out of the wall and adds it to the residual force. It adds the current position to the new path and continues to the next node, when the residual force is close to zero.

6.5 Collision detection

This part is maybe one of the more difficult implementations, because of the difficulties to test and debug the collision detection. A lot of time went to figuring out how to debug it when there was something wrong with the results. A geometry model could consist of more than 100 triangles. It is necessary to test for collision between the 12 lines of the transportation object and every triangle. This means that there are thousands of values to look at to find collision or not. It is difficult to debug if just a few of these numbers is wrong, and it is impossible to know which of them that are wrong. Since the dynamic relaxation algorithm does the collision test for every movement, the application does the intersection check for hundreds of thousands and maybe even millions of times.

The first attempt on the collision test used another line-plane intersection equation than mentioned in chapter 5. This equation missed some collisions as well as detecting collision when there was none. To figure out the reason it did not detect the crashes, few test cases just of these walls made the problem much smaller. These tests showed that there was something wrong with the line-plane intersection equation. It got wrong values to check if the line was inside of the triangle or not for some reason (same check as for barycentric coordinates in chapter 5). This is why the collision detection uses another intersection equation, as it is still unclear why the other equation did not work.

The implementation of collision detection first uses Cohen-Sutherland to rule out triangles that are completely away from the trolley. Even though Cohen-Sutherland is a 2D representation of clipping to a canvas view, it is possible to use it on a 3D object for every axis of a cube. Checking if every point in the triangle wall surface is higher or lower than the maximum and minimum x, y and z value of the trolley, rules out the surface.

Next step is to check if the surface and the trolley edge is parallel, which means that they will not intersect. They are parallel if an edge of the trolley is perpendicular to the normal vector of the triangle (normal · line = 0). If the trolley edge is not parallel to the triangle, the collision detection performs an intersection check by using equation 5.6. The collision detection is true if intersection is on the line and inside the triangle. Barycentric coordinates checks that the intersection is inside the triangle. If $0 \leq$ equation 5.6 ≤ 1, intersection happens on the line. There are no good libraries in .NET that allows matrix operations. MathNet.Numerics package [6] is a great option though, which is possible to install from the NuGet Package Manager in visual studio.

This collision detection will work two different ways. The first way when just validating the path. If the test detect a collision, it adds the position of the trolley to the path with a boolean for collision to be true. It return the external crash force to be zero for the dynamic relaxation algorithm. The second way the collision works is for optimizing the path. To do this, it has to return an external force that pushes the transportation object back out of the geometry model. A form of Cohen-Sutherland polygon clipping is a way of doing this. This algorithm is not easy to implement, as its purpose is not 3D clipping. It is not easy to sort out if there are more than one collision either. If it finds a collision, it adds this force to the total clash force where the force from collision is

$$F_{external} += d \cdot K_{wall} \tag{6.1}$$

d is distance the transportation object is outside of the wall at the given position, while K_{wall} is the stiffness of the wall. K_{wall} is a made up variable found through some attempts of trying and failing. The only important thing with K_{wall} is that it is not to big that the force pushes the position to far away from the actual position.

This is the most bugged part of the application, since sorting out which triangle collision that belong together is difficult. It is right now working only for crashes happening with just one wall. If there is a crash with a wall, it looks after if two of the trolley edges that crashes shares a corner. The distance outside of the wall is then distance between triangle plane and the trolley edge [18] that crashed with the wall. This distance is

$$d = \mathbf{n} \cdot (P - x) \tag{6.2}$$

$\mathbf{n}$ is the triangle normal vector, while P is the trolley edge and x is any of the vertices of the triangle plane. It also checks if the transportation object is moving perpendicular into the triangle surface by checking if the dot product between triangle normal vector and trolley edge is zero. First, it finds which side of the intersection point on the line that are the shortest. The shortest line segment is the one crashing with the triangle surface. If the simulation collide with multiple triangles, like in a corner of a wall, this test will not know how to find the force of the collision. It still manages to make a optimization that is not too bad, since the most normal collision is when one of the corners of the trolley is colliding.

6.6 Writing trolley visualization

A temporal file series with different positions of the trolley creates the animation in ParaView. These files are VTK files, so the structure of them is like the path file. To create the trolley, the same structure with polydata is on the top. The polydata consist of the 8 points forming the box and then the structure of how the points forms the cells. This is the only difference from the path file, since the structure of the path only showed the order of the nodes in a line. The file structure then look like this:

```
# vtk DataFile Version 4.0
vtk output
ASCII
DATASET POLYDATA
POINTS N float
x(0) y(0) z(0)
x(1) y(1) z(1)
...
x(N-1) y(N-1) z(N-1)

POLYGONS M K
4 0 1 3 2
4 2 3 5 4
...
```

The number of points N is in this case 8 where the points have x-, y- and z-coordinates. The polygons has M number of surfaces, or 6 for a cube, where K is the total of numbers the polygon part consist of. Each surface has a first number that explains how many point the polygon consist of and then which of the points following.

Last part of this file is the color attributes of the box. The box is green when not colliding and red when it does. This is how to give each of the surfaces a color:

```
CELL_DATA M
SCALARS cell_scalarsint1
LOOKUP_TABLE default
0 1 2 3 4 5

LOOKUP_TABLE default M
red green blue alpha
...
```

First up is defining each of the surfaces (cells) in the lookup table, and then let the lookup table define the color of each surface in the lookup table with RGBA (red, green, blue and alpha) values between 0 and 1. Alpha defines the opacity of the color.

6.7 Visualization result

There are two possible visualization results, which is validation and optimization. The user have a choice between running either validation check, optimization check or both checks, in this application. The validation is an animation where the color of the trolley object depends on if it collides or not. It is green if it does not collide and red if it does. The trolley is red from the first position it detects a collision until the last. Figure 6.1

shows an example of the first step it collides with the wall and the first position it does not collide anymore.

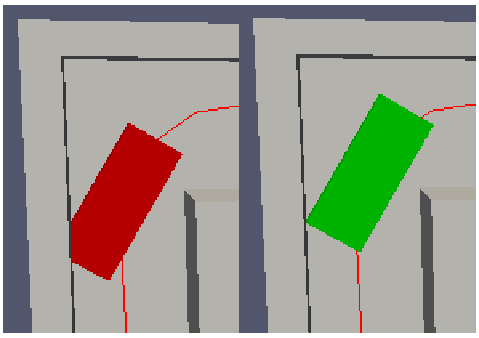


Figure 6.1: Result from two positions of the validation animation

The optimization check creates an animation and a new path. The new path shows the optimized path that involves no collision. This path might be impossible though, so the user still needs to check if it is a reasonable path. If it is impossible to optimize a new path without a collision, it will get to a tolerance and will not finish the optimized path. The animation of the optimized path moves along the new path. This animation should not contain any collision. Figure 6.2 shows the geometry model with the old path in red and the optimized path in green.

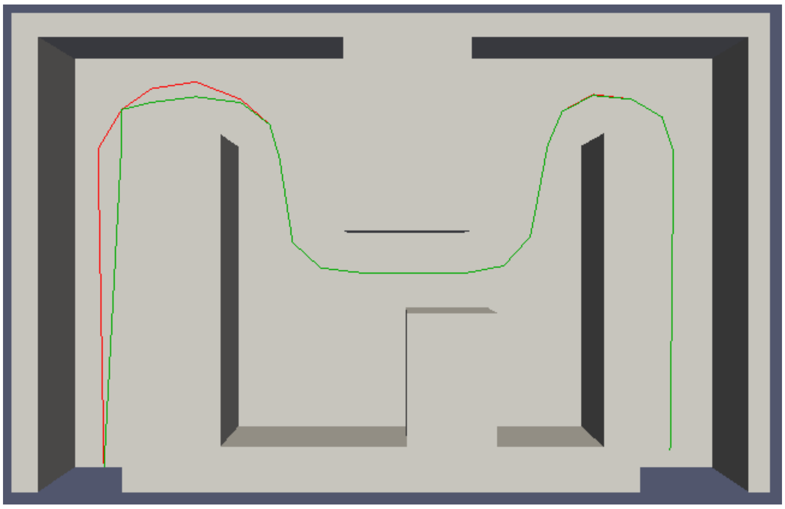


Figure 6.2: Geometry model with the old path in red and the new optimized path in green

Chapter 7

Conclusion

This chapter will summarize my work and give a conclusion to what I have done. It will discuss the findings and some of their problems and strengths.

7.1 Summary

The goal of this project was to create a software application. The application uses dynamic relaxation to check the feasibility of transporting the equipment along a predefined path. It was going to validate and optimize the predefined path and visualize the movement. This master thesis is a continuation on my project work in fall 2015 and some of the theory and work led to the results of this thesis. That also apply to the master thesis ”Verification and visualization of equipment access on offshore platforms” [15] from fall of 2014, as this work is a new take on the same problem. That means that some of the theory is similar.

I know .NET framework a little better from the project work last semester. So it feels natural to develop a .NET windows application, programming in C# and XAML. C# also contained the functionalities needed to develop this application. ParaView is a visualization GUI client that uses VTK libraries and will visualize the results of the verifications. A lot of time went to figuring out how ParaView works, as it is a powerful tool with many functionalities. Even though there is a lot of documentation online and a big ParaView community, it seemed like there was no good tutorial to get started except ParaViews own tutorial [8] and the documentation [14] from installing ParaView. These gave a good view on basic usage of ParaView but still did not explain how to create visualization data. A lot of trying and failing ended up in figuring out how to create visualization files in VTK format.

To test this application, I had to develop test cases. They consisted of a 3D geometry model and path. The geometry model is a solidworks model of passages to move through (see figure 2.1) and saved as an STL file. I also created a path in ParaView (see figure 2.2) and saved it as a VTK file. These test datas was important for every stage of developing the application. From reading the input parameters, simulating movement along the path to visualizing the movement.

Reading the input parameters was the first priority. Using dynamic relaxation to simulate the movement was next up after the application read the correct inputs. Some testing went to figuring out the size of different constants. The increment of the movement step and type of the system oscillation depends on the trolley stiffness, trolley mass and the damping coefficient. It did not change the output results at all though. Visualizing the movement was next, after the dynamic relaxation created a correct movement for the trolley. Last step was the collision detection. I implemented this in the end, since it was easier to test when the application visualized the results.

The code went through a consistent refactoring process. Especially in the end, since it was a little unclear what result to visualize and how to do it. The user got a choice to verify and optimize the path. Verification shows every step of the old path, with the trolley colored red when colliding. The optimization created a new optimized path that did not collide, if that was possible. Verification was impossible to do at the same time since the optimization changed the trolley position during the simulation. This meant that the simulations had to run separate. This led to a new code structure, as it was not planned to run the simulation more than one time.

7.2 Discussion

The plan was in the beginning to create an effective, user-friendly application that was open source, free to use and independent to any other application. It was important, as this work is a continuation of a previous master thesis [15] that created a plugin using Navisworks. The goal was to create an application that was possible to use for everybody, as Navisworks was an expensive software for corporate use. I figured out quite early that it would be impossible to fulfill these expectations. The application needed a view to create the path as well as for visualizing the results, which would be a lot of work. I would not have time to gain the necessary knowledge to develop the view with the functionalities needed. This is why I chose to use ParaView. This change was not the worst as ParaView is an open-source software and quite simple to use for this purpose. It is not a lot of extra interactions for the user either, as ParaView is already open when creating the path. Users only need to run the simulation and open the files in Paraview to see the results.

Gaming communities use collision detection a lot. This means that there are a lot of different algorithms and ways to test for collisions. The choice landed on line-plane intersection, but there was a lot of others to pick as well. Line-plane intersection is easy to implement, as the input parameters is just points, vector lines and planes. The problem with line-plane intersection was that it was difficult to figure out the directions of the collision between plane and line. It was no easy way to sort the different collisions as the transportation object might collide with many wall surfaces at the same time. It was an idea of implementing a plane-to-plane intersection instead. Direction of the collision might be possible to figure out by checking intersection between the planes, without being certain. The line-plane intersection will not work in some cases either. Like when there is an obstacle intersecting on the surface side of the trolley without intersecting with any of the edges. A plane-to-plane intersection algorithm seemed complex and would take more time to implement. As it was not a lot of time left, it felt safer to use line-plane intersection. It would be interesting to check out some of the more used algorithm in game developing. This is algorithms such as the separating axis theorem, Minkowski Sums and the Expanding Polytope Algorithm just to mention a few.

There was a major change in the code in the end of the project to develop a simulation for verifying the simulation and optimizing the path. The user got the choice between verifying or optimizing the path as mentioned earlier. Some of the reason of this choice is so that the user got the possibility to see if it is necessary to optimize the path at all. The idea was that the user might see from the verification if it is necessary or even possible to create an optimization. It should not be any reason for the user to optimize the path if there are no collision. Visualization result from verification looks quite good, even though there was no specification to create a verification simulation that showed every parts it collided.

Some of the specification was to use dynamic relaxation to create movement along the path. This worked well and was an interesting way to solve the problem. It is other ways to solve this problem though. One way would be to use a pathfinding algorithm. Pathfinding checks for the shortest route between two points.

7.3 Further Work

The application is still in need of some improvements. Some of the implementation does not work as they should and contains flaws. There are also possible to add some functionalities and change some the implementations, as it is possible to solve some of the problems in another way that might be better. The most important thing to improve is the collision detection where a force should push the trolley back out of the wall. This implementation does not work as intended as it only finds the force if it is a corner of the trolley that collide with a single triangle of the wall. Nothing happens if it collides with more than one triangle surface at the time. There are many ways of fixing this part. One way is to keep using the same implementation, which I think would be difficult. Another way is to find a new technique that uses Cohen-Sutherland polygon clipping or a similar algorithm to sort out the different type of collision. What I would recommend is to implement a new collision detection. It might take the most time to do, but it will solve some other problems with the current collision detection. Expanding Polytope Algorithm is my recommendation for another collision detection, since it allows getting the depth of intersection.

This application ended up working pretty well for many different cases and scenarios. It is a few problems and scenarios it does not work for though. The application does not read binary VTK files when reading the path. This is not important for developing a functioning application, but makes it easier for users to create the path, as it does not have anything to say if the file is in ASCII or binary. Some types of collision will not register, such as when an object of the geometry model collides on the surface of the transportation object, and not with the edges. This is why I did recommend using another collision algorithm earlier, as it will solve these problems. The implementation of collision is much slower than I hoped for, but I am uncertain if changing collision algorithm will improve runtime. One possible solution is to use octree, which is a tree data structure that divides a 3D space into smaller parts. It rules out parts of the model that is not interesting for collision check. One other case that the application does not work for is, when the path moves through more than one axis. I did mention this in section 6.3, as the transportation object will rotate around the z-axis only. Quaternions are the best ways of fixing this, and should not be hard to fix either.

Developing a view is also something that can be interesting. It is possible to develop a view using VTK. Creating a view will help, as the user will not need to create a path in an external software and save any files for the results. The application will visualize everything in the view, with functionalities to create the path as well as visualizing results straight from the calculations. Implementing a view might make the application more user friendly and easy to use.

7.4 Conclusions

Working on this application have been interesting and challenging in some ways. I have never worked with visualization tools before, and had to use a lot of time to understand how to represent visualization data. I did not have time on the more challenging problems and made some bad choices in the end, since I had to use a lot of time on what I now think is basic. Especially the collision detection is one of these bad choices. There are many well-documented collision algorithms and libraries online. The problem is that these algorithms are complex and takes some time to understand.

There are choices I am not satisfied with, even though the results is not too bad. The test cases ended up giving result that gave sense for checking feasibility of the transportation. I do know that output of the optimized path for some of the cases was wrong. It did not manage to deal with some types of collision. User still have the possibility to figure this out through the verification results. I feel like this report answers most of the objectives. It was a little difficult to show the different implementations without sharing the code. But there are some explanations on how different implementations uses the theory behind.

References

[1] Autodesk naviswork software. `http://www.autodesk.com/products/navisworks/overview/`. [Online; accessed 17-December-2015].

[2] Computer graphics - clipping. `http://www.cc.gatech.edu/grads/h/Hao-wei.Hsieh/Haowei.Hsieh/mm.html#sec3`. [Online; accessed June-2016].

[3] Dynamic relaxation. `https://en.wikipedia.org/wiki/Dynamic_relaxation`. [Online; accessed May-2016, last modified 16-January-2016].

[4] Intersctions of lines and planes. `http://geomalgorithms.com/a05-_intersect-1.html`. [Online; accessed June-2016].

[5] Line intersection with aabb rectangle. `http://stackoverflow.com/questions/3746274/line-intersection-with-aabb-rectangle`. [Online; accessed June-2016 last modified 16-November-2011].

[6] Matrices and vectors. `http://numerics.mathdotnet.com/Matrix.html`. [Online; accessed May-2016].

[7] ParaView - overview about paraview by kitware. `http://www.paraview.org/overview/`. [Online; accessed April-2016].

[8] The paraview tutorial. `http://www.paraview.org/Wiki/images/5/56/ParaViewTutorial44.pdf`. [Online; accessed April-2016].

[9] Point in triangle test. `http://www.blackpawn.com/texts/pointinpoly/`. [Online; accessed May-2016].

[10] Reading an stl file with c++. `http://www.sgh1.net/b4/cpp-read-stl-file`. [Online; accessed October-2015].

[11] The stl format - standard data format for fabbers. `http://www.fabbers.com/tech/STL_Format`. [Online; accessed 17-December-2015].

[12] Viewing clipping. http://www.tutorialspoint.com/computer_graphics/viewing_and_clipping.htm. [Online; accessed June-2016].

[13] VTK - overview about visualization toolkit by kitware. http://www.vtk.org/overview/. [Online; accessed 11-May-2016].

[14] Utkarsh Ayachit. *The ParaView Guide - Community Edition, Updated for ParaView version 5.0.* Kitware Inc., 2015.

[15] Marius Hansen Røed. Verification and visualization of equipment access on offshore platforms. June 2014.

[16] Eric W. Weisstein. Barycentric coordinates. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/BarycentricCoordinates.html. [Online; accesed June-2016].

[17] Eric W. Weisstein. Line-plane intersection. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/Line-PlaneIntersection.html. [Online; accesed May-2016].

[18] Eric W. Weisstein. Point-plane distance. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/Point-PlaneDistance.html. [Online; accesed May-2016].