Visualization and comparison of geospatial data

Using modern game development tools

Tom Kristian Tjemsland

University of Stavanger

ABSTRACT

Every second, a mind-numbing amount of data concerning the real world is gathered worldwide. Everything from the speed of cars through a road junction, to the geographical position of African lions, are registered and stored on enormous servers. This data have been collected to serve many different needs and purposes and so, the possible formats for the representation of the data vary greatly. Some data represents a single point on a road section, such as accidents and locations of zebra crossings, while another set of data could represent directional geospatial data, such as force and direction of wind or ocean currents. Often, a simple data registration also contains multiple properties. Weather data collected from a single location could for example include properties such as humidity, temperature, precipitation and dew point temperature.

This paper will explore how to reliably visualize and compare arbitrary properties from different data collections belonging to the municipality of Stavanger. Easy integration of new sources will be possible, but for the scope of this paper, the source will exclusively be open data made available by *Statens Vegvesen*. How to properly visualize different data sets using virtual reality will also be discussed, meaning that both user experience and GUI will be considered. The goal will be to make a program that is both user-friendly and delivers intuitive functionality.

KEYWORDS

Data Visualization, Virtual Reality, Game Engine, Unity

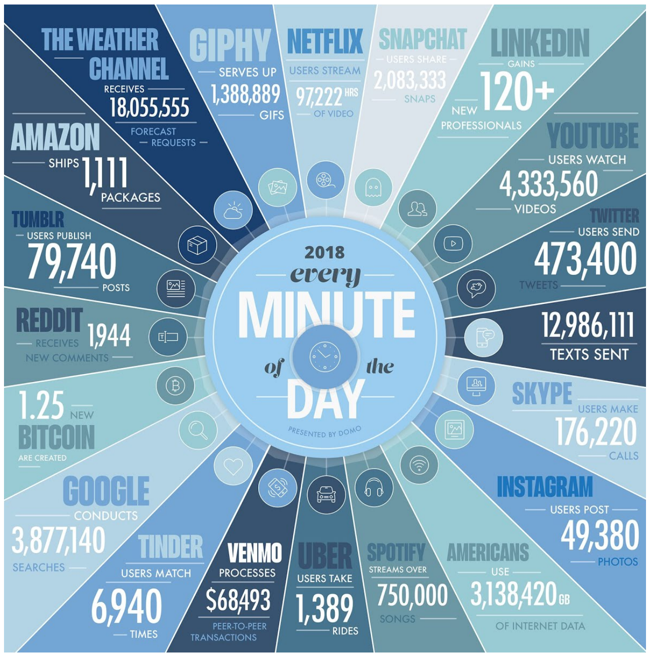


Figure 1: Data generated per minute in 2018 as presented by the report *data never sleeps 6.0.*

1 INTRODUCTION

Human population, internet speeds, and storage capacities are all rapidly increasing, and naturally, the amount of data generated follows this trend tightly. An industry insight released by IBM in 2013 estimated that 2.5 million terabytes of data were generated every day. They also estimated that 90% of the world's data had been generated in the last two years [3]. It's clear that the amount of data being produced yearly is growing exponentially and it shows no sign of slowing down. For more modern statistics, the American computer software company DOMO releases a yearly report on data generation called *data never sleeps* [1]. The 6th edition covers the year 2018 and an excerpt from the presented results is shown in Figure 1. We can see that data comes from widely different sources and in many different shapes, with data pertaining to both physical and virtual events.

Scientia potentia est, or knowledge is power, is a famous Latin phrase often attributed Sir Francis Bacon and it absolutely holds true for today's world. As more data is being generated, it becomes much more important to collect and research this data. For privately owned companies such as Lyse, Ebay or Amazon, this research could provide invaluable results that could help steer the company towards a more profitable path. This could be achieved by increasing profits or by decreasing losses and expenses. But because this research contain so much value, it's often kept private and impossible to access. With the increase in data gathering, there is also an increase in publicly available data. Norwegian Public Roads Administration, or *Statens Vegvesen*, is an example of a source that has recently made huge amounts of data publicly available. Open data like this carries significant value by itself and enables us to read and process all kinds of information. However, instead of treating the data sets strictly as independently collections, it would be extremely valuable to be able to compare them across sources. Data

Supervisor: Erlend Tøssebro.
External Supervisors: Morten Forsberg Aasbak and Sindre Tøsse.

that can be pinpointed to some specific location, be it country, city or geographical coordinates, are especially interesting since data with overlapping locations can easily be compared. By taking it one step further and allowing users to explore the data in Virtual Reality, the value of the data sets would be enhanced as distribution, outliers and potential patterns would be much simpler to observe.

Related Work. Despite the value it could provide, there is little information to find about generically comparing geospatial data across different sources. There has been far more research regarding geospatial data in general. For instance, an article released in the journal *Big Data Research* discusses the challenges and opportunities that are linked to geospatial big data [5]. Another article explains a tool developed for visualising simple data by using the Quake 3 game engine [4]. While the latter shares some similarities with this paper it does not explore the possibilities for comparing data.

- We visualize a map over Stavanger, Norway in the game engine Unity.
- We visualize different data types as a three dimensional overlay to the map.
- We construct a graphical user interface allowing the comparison of arbitrary properties.

2 BACKGROUND

In this section, we will be introducing both *Unity* and *Mapbox*, as well as giving details on the source that we will be using throughout this paper, *Statens Vegvesen*. We will also briefly explain the content of the data made available.

The game industry is a multi-million dollar business, and the U.S. alone has more than 2400 companies operating within this sector [8]. A handful of commercially available game engines clearly dominate the marked, with some of the more popular alternatives being Unity, Unreal Engine and Game Maker. Game Maker focuses primarily on the development of two-dimensional games, whereas both Unreal and Unity have a clear focus on the development of three-dimensional games. Unity was initially released in 2005 and the intention was to offer a more affordable game development tool to the public. A few years later, in 2009, at a conference in San Francisco, it was announced that Unity would become freely available [10]. This resulted in a massive surge in popularity that can be seen in Figure 2. Coupled with well documented functionalities, this have resulted in Unity becoming the most wide-spread and popular tool for creating games.

Mapbox is a company providing online maps that was founded in 2010. The available data comes from open data sources, such as *OpenStreetMap*. Although many map providers exist, Mapbox is of interest as it can easily be integrated into Unity through a free SDK. Beside providing standard map visuals, Mapbox also supports visualization of three-dimensional map geometry and buildings. The visualization of the various objects is divided into smaller sections, known as chunks. By providing Mapbox with a geographical coordinate, we can reliably visualize all chunks within some specified radius. The SDK also comes with caching as an *out-of-the-box* feature. This means that the program will save time whenever the user tries to load a chunk that has been visited previously.

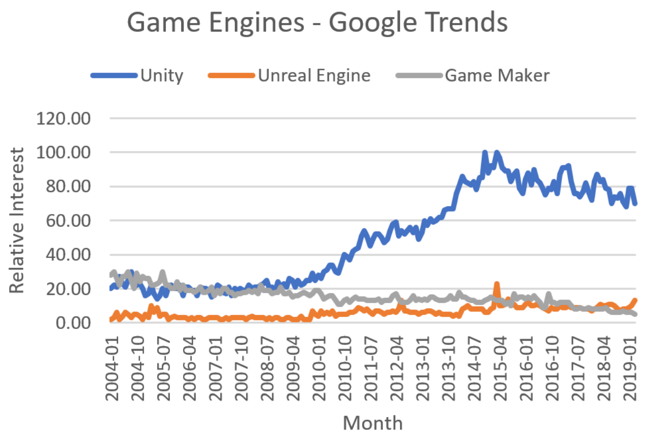


Figure 2: Relative interest in the search terms "Unity", "Unreal Engine" and "Game Maker" since 2004. Data found using Google Trends [9] and plotted in excel.

All the data that we will be working with is distributed by *Statens Vegvesen*. The various data collections can be retrieved as either XML or JSON formatted lists. This source supports complex queries, which means that very specific collections of data can be requested. Being a public roads administration, all the data that can be accessed is related to Norwegian roads or events occurring on the roads. An example of available data collections provided by *Statens Vegvesen* can be seen below.

- **Manholes**
- **Speed Limits**
- **Accidents**
- **Tunnel Sections**
- **Bridges**
- **Speed Bumps**
- **Traffic Amount**

Although this source introduces a fair amount of data, they always fall into one out of two categories. The real-world location of data can be given as either a single point or a linear section of some road. *Accidents* and *Speed Bumps* are examples of the former, while *Tunnel Sections* and *Bridges* are examples of linear data.

Each entry in a data collection contains a great amount of information. Among this information there is a list of *properties*. As an example, entries of *Manholes* contains properties such as *Depth*, *Diameter* and *Material Type*. It is primarily these properties that will be the focus of all comparisons. There is also information regarding which road the data belongs to, and the geometry of the data. For point data, this geometry is simply given as a single geographical point. Line data is instead given as a list of geographical points. This list is usually extremely long due to the density of the points.

Lastly, it would be absolutely meaningless to compare the properties of data collections at one specific instance of time. This is because the data provided by *Statens Vegvesen* is updated infrequently and irregularly. Some of the recorded values are months old, while others are older than a decade. However, this does not mean that comparisons will be irrelevant altogether. This simply means that all the properties will have to be treated as either relative or constant. There is a clear overweight of constant properties, with some examples being the *diameter of manholes* and the *length of bridges*. *Traffic Amount* is an example of a data set with some relative properties. It cannot tell us how much traffic there is on some road at a given time, but it can tell us which road has the highest traffic load relatively.

3 METHOD

In this section, we will elaborate on how the program was made using the game engine, Unity. Because we need the support for both visualization and comparison from the same software, we will have to design a more complex structure.

Point Data:	Data covering a single point on a road.
Line Data:	Data covering a section of some road.

For simplicity, we will from here on, refer to the two data categories as *Point Data*, and *Line Data*. As seen in the table above, they refer to *points on a road* and *sections of a road*, respectively.

3.1 Defining comparable data

Before looking into the design of the program, we have to assign a definition to the term, *comparable data*. With two different categories of data, there are three unique types of comparisons that would be technically possible. These comparisons consists of comparing *point* and *point* data, *point* and *line* data, and *line* and *line* data. However, in our specific situation, only one of these comparisons would actually provide valuable results. Considering two different collections of point data, we would immediately encounter an issue with overlapping data entries or rather lacks thereof. Regardless of the amount of data entries, very few will share the same identical geographical space. Road data can only be used to describe their exact location, unlike other types of data such as temperature, wind and precipitation, which also describes their surroundings. This introduce complications when performing generic comparisons, as it would be difficult to determine the direct relationship between the data collections. As a result, the only cross-data comparison that we will be considering is between *point* and *line* data. These two data types can easily be compared as long as the section covered by the line data also contains the point. The results of these comparisons can then be used to create co-occurrence matrices or scatter plots, depending on the specific property types involved.

As mentioned before, comparing properties across different data collections does not produce meaningful results. However, we are still interested in allowing users to compare two different properties belonging to the **same** data collection. Since comparisons like this only occur internally in a data set, property overlapping can be ignored as it's no longer of concern. An example of a comparison like this would be to compare the *depth of manholes* with the *diameter of manholes*. The results of an internal comparison can be presented in the exact same manner as a cross-data comparison.

Because Unity operates with three-dimensional *scenes*, it would also be of interest to plot three different properties against one-another in three-dimensional plots. Not only could this be used to show pattern between more than two properties, but it could also be used to evaluate the value of a 3D environment.

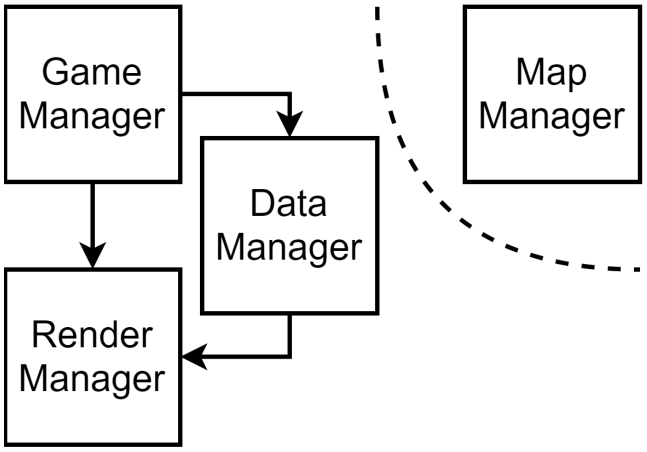


Figure 3: General structure of the finished program.

3.2 Property types

The number of available comparisons rises exponentially with the number of allowed property types. We have selected two properties, numerical and text-based, to maintain focus on the design of new features, rather than implementing support for an increasing number of comparisons. Other properties such as date and geometry will be ignored, which means that we need to consider three different possible comparisons, as seen below.

- *Numerical* and *Numerical* comparison.
- *Numerical* and *Text* comparison.
- *Text* and *Text* comparison.

When both properties are numerical, we will show the results of the comparisons as a scatter plot. The X and Y coordinates of each point will correspond to the numerical values on overlapping properties. Similarly, when comparing numerical and text properties, the X coordinate will correspond to the various text values. The result will be a plot with scattered "lines" over each text value, which gives a visual insight on data density. If both properties are text-based, the result will be shown in a co-occurrence matrix. The rows and columns will correspond to the possible text values of the two properties. The number within each cell will be the result of how many times the different text values occur together. The produced co-occurrence matrix shows density, as well as which text values are more likely to overlap.

3.3 Design

The final program needs to fulfill many requirements. It needs to be able to render multiple data types simultaneously and compare arbitrary properties. These operations need to occur so that the program does not slow down or stutter significantly. Furthermore, the graphical user interface, or GUI, has to be intuitive enough to require minimal explanation. Meeting these requirements have been the main focus when designing the program. This has led to a general structure as shown in Figure 3.

Game Manager denotes the primary script and follows the standard Unity naming convention. This is the component responsible for start-up procedures and general management of the program.

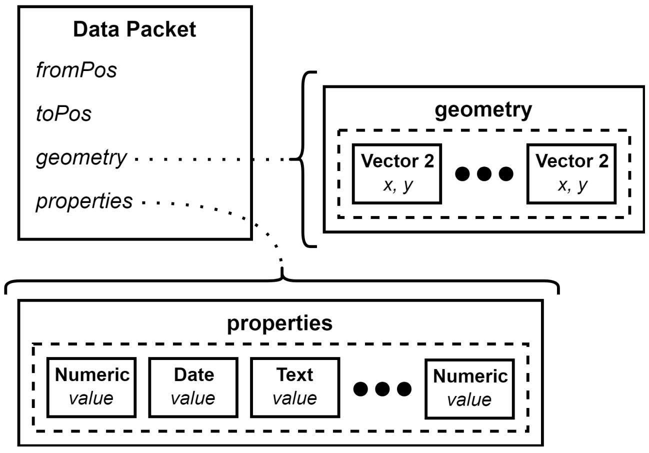


Figure 4: The structure of each individual data packet.

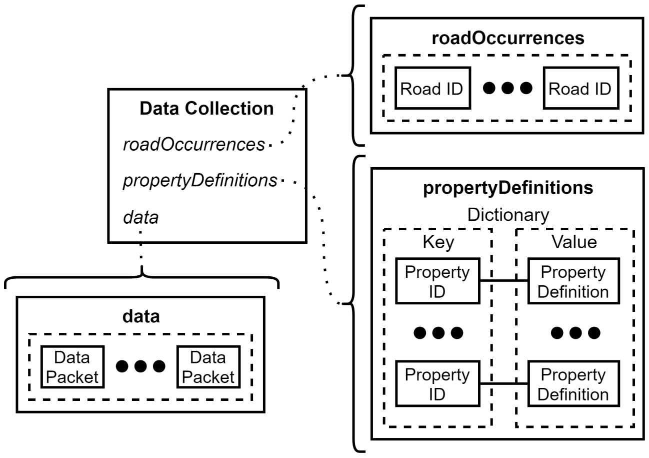


Figure 5: The structure of each data collection.

For this particular project, its primary function is to handle user input and relay information to the rest of the system. Both button presses and drop-down menu selections within the GUI main menu will be handled by this component. Other functions of the *Game Manager* is to inform the *Data Manager* whenever new data needs to be cached and update the content of the *Render Manager.*

The *Map Manager* provide a static instance of itself and is accessible from any script. This component mainly functions as a connection to the functions and variables belonging to Mapbox. Many important variables have been set within the map object even before run time, such as tile-size and radius of the rendered region. One of the most important variables is the current position of the player. The *Map Manager* makes it easy to retrieve this position clamped to the center of the closest tile. It also functions as a translator between geographical coordinates and world-space coordinates. This is important because all data returned from *Statens Vegvesen* has its location expressed as latitude and longitude.

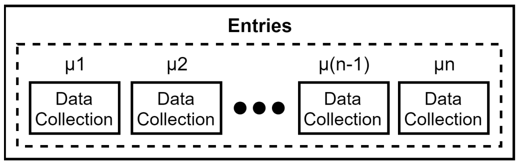
The *Render Manager* handles everything related to data visualization and comparison. Whenever new tiles should be rendered, the *Render Manager* queries the *Data Manager* and uses the results to draw data within a bounded region around the player. It can simultaneously render two different data types, primary and secondary. Primary and secondary has to be set by the *Game Manager* before the respective data can be rendered. Different data types also have to be considered, with *line data* rendering as lines, and *point data* rendering as pointers. Because this component has control over all the data visuals, it is also responsible for generating the various plots. Having a strong connection between rendered and visualized data makes it possible to give the user feedback regarding the relations between them.

Data Manager. Most of the design requirements rely on well-structured data management. This makes the *Data Manager* the most important component of the general structure. It also makes this component the most complex of them all, with many internal classes and structures. To better explain the *Data Manager*, we will be elaborating on the design in a bottom-up fashion. Every single data point from *Statens Vegvesen* is represented as a data packet as seen in Figure 4.

fromPos and *toPos* is where each measurement starts and ends relative to some road. They are normalized, and so they both range between zero and one. For *Point Data*, these fields have been assigned the same value. The geometry is given as an array of vectors. Each 2-dimensional vector has a latitude and longitude pair stored in their x and y variables. For *Line Data* this array could potentially become very long, but for *Point Data* there will always be only one element. Each *Data Packet* also contains a set of properties. The properties within this set can be either numeric, textual or related to dates. If we know the property-type at each index, the original values can be retrieved by type casting.

Data Collection, as the name indicates, represents an entire data collection from the municipality of Stavanger. These are complete collections such as *Manholes* and *Speed Limits*. The structure of a *Data Collection* can be seen in Figure 5. The field *roadOccurrences* contains a list of unique road IDs. The *data* field is a collection of all individual data packets. *propertyDefinitions* holds a dictionary, which connects property IDs with single property definitions. Each definition contains data regarding the type and range of its respective data. It also provides an index, which expresses the position this property type has in the *properties* collection of individual data packets.

All the different data collections come together to create a big set of entries. The structure of this set can be seen below.



In the figure we are referring to the various entries with indices ranging from $\mu 1$ to μn. This is because the key to these indices will be used within the *Data Manager* on multiple occasions. We will be referring to these as the *data indices.*

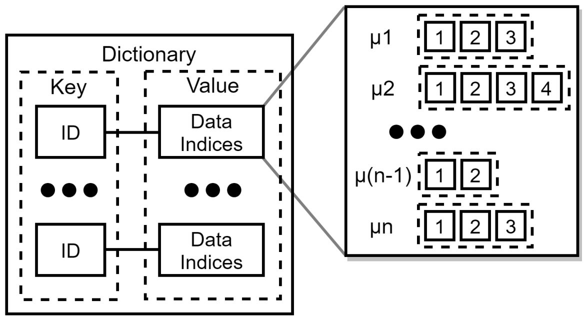


Figure 6: The structure of a look-up dictionary.

As mentioned earlier, *Data Collection* contains an array with all data belonging to a certain category. The problem with this structure is that it's not optimized for either data visualization or data comparison. To visualize data around the player, the program would have to iterate over the full collection of available data and calculate distances. This is not a viable solution, as it is slow and provides insufficient scalability. Our solution is to create two look-up dictionaries that can be used to find all data sources belonging to specific roads and tiles. *Line Data* is assigned to tiles based on the center of their geometry. Both dictionaries follows a similar structure, as seen in Figure 6.

For visualization the keys are clamped tile coordinates calculated by the *Map Manager*. For comparisons the keys are IDs of different sections of road. Similar for both occasions, is that the values consist of jagged arrays. The *data indices*, used when finding data entries, will also be used here to extract appropriate sub-arrays. Each of these arrays contains a collection of measurement indices located on the given road or tile. The indices point to a specific *Data Packet* within the *data* field of a *Data Collection*.

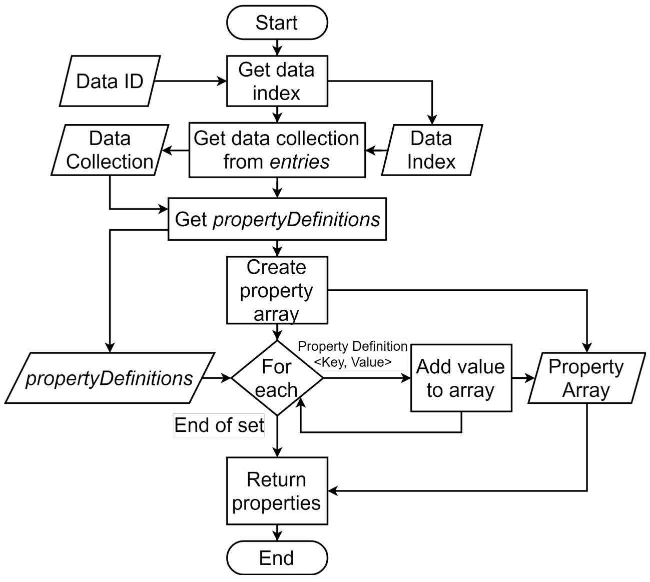


Figure 7: Returning properties for a given data type using the *Data Manager*.

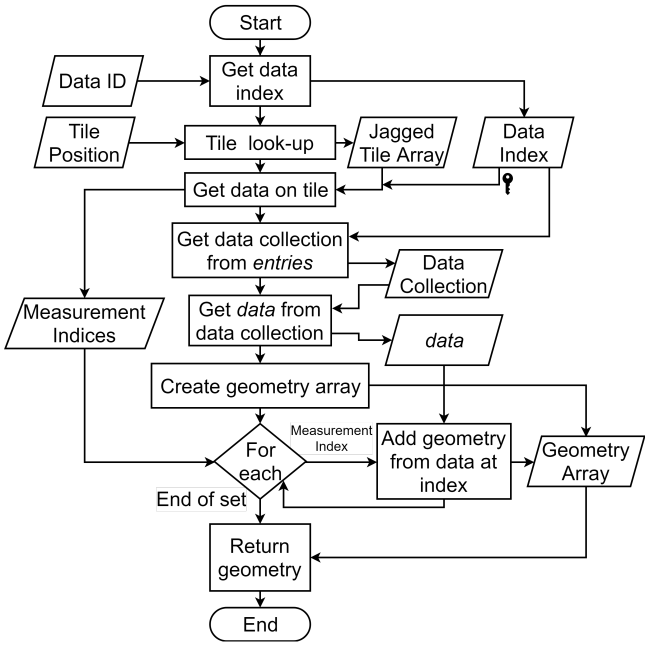


Figure 8: Returning all geometry for a specified data type on a given tile using the *Data Manager*.

The last important component of the *Data Manager* is the entry dictionary. This dictionary takes in a data ID and returns a *data index*, starting from zero. It also contains information regarding the data category, and can be used to evaluate whether a given data type is *Point Data* or *Line Data*.

Flowchart. In order to better explain the design of the *Data Manager*, we will now look at flowcharts for two possible situations. This will help show how all the different sub-components are interconnected, and how they can be used to produce certain output data. The situations are as given below.

- *Game Manager* requests full list of properties for a data type.
- *Render Manager* requests all data geometry for a specific data type on a given tile.

The first situation can be seen in Figure 7. As long as the data is already cached, this system is mostly closed, with only one input variable being needed, Data ID. The dictionary *propertyDefinitions* is often short since it only contains one entry per unique property available on a given data type. As a result, getting all properties belonging to a certain data type is a very simple operation.

Getting all data from a given tile is slightly more complex, given all the sub-components required for performing the operation. The flowchart for this particular situation can be seen in Figure 8. This operation starts similar to the previous one, but after retrieving the data index, it also needs to refer to a look-up dictionary. The data index works as a key to find the correct sequence of measurement indices belonging to the tile. By using these indices, the geometry can be accessed on all relevant measurements in a data collection.

3.4 JSON structure

This section explains the general structures of the JSON responses received from *Statens Vegvesen*. Because there are different data categories and property types, many variations exists within the original structure. All significant variables on the various levels of the JSON will be briefly explained. The top-level structure of the JSON can be seen below.

id:	ID of the data entry.
href:	Link to request all data regarding entry.
egenskaper:	A set of properties.
segmentering:	A set of municipalities that the data belong to.
geometri:	Information regarding physical geometry.
vegsegmenter:	Information regarding road segmentation.

These names have been given in Norwegian and are the same as the ones encountered within the JSON. The structure of a single property vary slightly depending on whether it is *Numeric* or *Textual*, as seen below.

	Common
id:	ID of the property.
navn:	Name of the property.
datatype:	An integer denoting type of data.
datatype_tekst:	The name of the data type.
verdi:	The value of the measurement.
	Numeric Exclusive
enhet:	Information regarding unit.

As presented in the table, the field *enhet* is exclusively given numeric values. This field is very important for understanding the context of the correlated value. It is, for example, absolutely vital that users can see whether a value regards meters or kilometers. The *enhet* layer can be seen below.

id:	ID of the unit.
navn:	Name of the unit.
kortnavn:	Short version of unit name.

Most of the properties encountered have a datatype of either 2 or 30. These are numeric and textual properties, respectively. A real property entry of each datatype can be seen below. These properties belong to a manhole situated at *Øvre Stokka, Stavanger*.

id:	1586	**id**:	1411
navn:	Dybde	**navn**:	Materialtype
datatype:	2	**datatype**:	30
datatype_tekst:	Tall	**datatype_tekst**:	Tekst*
verdi:	0.6	**verdi**:	Betong
enhet→id:	1		
enhet→navn:	Meter		
enhet→kortnavn:	m		

* Truncated to avoid overflow. Real value is *FlerverdiAttributt, Tekst.*

The *geometry* field of the upper layer only have a single field of significant interest for this paper, *wkt*. This is an abbreviation for *well-known text*, which is a standardized way of representing geometry through text. The geometry returned by *Statens Vegvesen* is mostly on the format *POINT Z* or *LINESTRING Z*. These represent point geometry and line geometry, respectively. The *Z* denotes that there is also height information available within the position vector. The structure of these geometry strings can be seen below.

POINT Z:	POINT Z ($X\ Y\ Z$)
LINESTRING Z:	LINESTRING Z ($X_1\ Y_1\ Z_1, \cdots, X_N\ Y_N\ Z_N$)

Depending on the resolution of the geometry, *LINESTRING Z* can potentially become very large. Realistically, there are often more than 20 positions making up every individual line. Although every position includes a Z-coordinate, we will be completely ignoring this value throughout the project. We can do this because *Mapbox* already translates any latitude and longitude to its correlated height on the provided map.

The last field, *vegsementer*, contains information regarding the road where the data entry is located. This field is very important when performing cross-data comparisons, as it can be used to evaluate whether data overlap or not. If *Point Data* and *Line Data* overlap on a section of road, it becomes more meaningful to compare properties between these data entries. The *vegsementer* layer found within the JSON can be seen below.

stedfesting:	Location relative to road.
kommune:	Municipality of road section.
fylke:	County of road section.
region:	Region of road section.
strekningslengde:	Length of road section.

In this particular case: municipality, county, and the region are predetermined by the data request, so the only field of interest is *stedfestning*. The content of this field vary depending on the category of the data, as seen below.

	Common
veglenkeid:	ID of road.
kortform:	Compressed road info.
retning:	Direction with or against road vector.
felt:	Amount of lanes on road.
	Line Data Exclusive
fra_posisjon:	Start position of data entry.
til_posisjon:	End position of data entry.
	Point Data Exclusive
posisjon:	Position of data entry.
sideposisjon:	Side of road of data entry.

The fields *fra_posisjon*, *til_posisjon* and *posisjon* all regards relative positioning. This means that the correlated values span the unit interval, or $[0, 1]$. Despite being a common field, the structure of *kortform* varies between the data categories. For *Point Data* its on the format *posisjon@veglenkeid*, while for *Line Data* its on the format *fra_posisjon-til_posisjon@veglenkeid.*

The content of a *stedfestning* field from both categories can be seen below.

Point Data		Line Data	
veglenkeid:	319606	**veglenkeid**:	320581
posisjon:	0.64	**fra_posisjon**:	0.93
—		**til_posisjon**:	0.95
kortform:	0.64@319606	**kortform**:	0.93-0.95@320581
retning:	MED	**retning**:	MED
sideposisjon:	H	**felt**:	2

Position values have been rounded off to nearest two decimals to avoid overflow. The left side of the table is the same *Manhole* entry that was used to present real properties, while the right side is from a *Speed Limit* entry covering a small section of the motorway passing through *Forus*, *Stavanger*.

3.5 Analysis

There is absolutely no lack of tools that already visualize data. The sheer amount of data that can be visualized through online sources is staggering. This data can vary immensely, with some examples shown below.

LightningMaps.org:	Displays real-time lighting occurrences.
Murdermap.co.uk:	Tracks homicides in London, UK.
MarineTraffic.com:	Displays positions of marine vessels.
Kolumbus.no:	Real-time tracking of Norwegian buses.
Worldometers.info:	Displays worldwide changes in population.
BirdCast.info:	Tracks migration of birds over America.

Despite being very different in how they operate and what data is provided, all sources have a very limited scope. *BirdCast.info*, for instance, only provides data that is considered relevant for the migration of birds. This is usually the case with online data visualisation services. They are often tailored to certain uses, which makes it difficult to adopt the data for other purposes. For example, if an individual would like to analyze the connection between population growth and the increase of marine traffic, they would have to perform manual cross-checking.

The program being developed as a part of this project has a very different approach to data visualization, with no predefined properties. As a result, it is possible to integrate new data provides to meet different demands. This makes it possible for users to perform **any** desirable comparison, regardless of the originally intended scope.

The use of game development tools is another reason why this project is very different from the more common web-based approach. Through the use of virtual reality, it becomes possible to observe the data in new and interesting ways. For example, instead of *zooming*, the users can instead lean closer physically. This is a more intuitive way of exploring data. Additionally, the use of 3D to visualize data opens up for the possibility of producing more complex plots. However, plots of higher dimensions will not be relevant for this project as we are only comparing two properties at a time.

3.6 Optimization

The majority of the optimizations have been aimed at increasing the speed of visualizations and comparisons. The most central component for performing these operations is the *Data Manager*. As a result, this component has been the major focus for optimizations. While the design was elaborated in Chapter 3.3, there are also many optimizations that does not concern structural aspects. With one of the more important ones being the caching logic.

Caching of data. When the user first requests a new data type, the response from the server will be stored locally. This is a process known as caching and makes successive requests much faster. Since the data we will be operating with belongs to the municipality of Stavanger, the program caches all data of a given type in a single operation. For larger cities, such as Oslo, it could have been necessary to divide the region into smaller subsections to maintain acceptable operation speeds. However, this could create other problems such as *Line Data* entries crossing boundaries, which again would result in data *existing* on multiple tiles. Besides increasing cache size, this could cause multiple instances of a data entry to be rendered simultaneously, which would result in unnecessary draw calls.

For certain operations, the program needs to know the minimum and maximum values found within a set of *numeric* properties. These values are stored together in each properties' respective *Property Definition*. Getting the minimum and maximum values for a single property requires an iteration through all data entries in a set. This is of concern as the number of entries and properties could cause a slowdown if the program were to calculate everything while caching data. The way the program handle this issue is by calculating these values on-demand. Just like for entire data sets, these values are also cached to operate faster on successive requests. This is a much more logical approach, as most properties will not be of interest to a single user. At most occasions only a small number of properties will actually be visualized and compared.

There will always be a trade-off between storage requirements and performance. By introducing caching and look-up dictionaries, we effectively increase the speed of operations, at the cost of needing to store more data locally. The best way to combat this issue is by limiting the size of the locally cached data. As seen previously in Figure 4, this program stores numeric properties by their actual type, rather than as pure text. This help reduce the size of the final cache as *float* variables only require 4 bytes, rather than 1 byte per character. The different storage requirements for a small section of *pi* can be seen below.

π:	3.1415927
Float:	$4\ bytes$
String:	$9\ characters = 9\ bytes$

As the program receives new data from *Statens Vegvesen*, it has to perform some heavy processing before the result can eventually be stored locally. Part of this processing involves parsing pure text into the enormous JSON structure described earlier. By default, this causes the entire program to stop for multiple seconds, which negatively impacts the VR experience. To avoid this issue, an optimization was made where most of the heavy processing occurs in a separate thread. This works well as long as the user is given visual feedback regarding the current progress.

3.7 Implementation

The program was in its entirety programmed in C# using the powerful game engine, Unity. In this section, we will highlight central concepts of the various components, how they function, and how they operate together. We will also elaborate set-up and operation of the *Graphical User Interface*.

One of the features that makes Unity so useful is a feature called the *Inspector*. The *Inspector* makes it possible to view and interact with the different components of a game object directly in the Unity IDE. This feature is especially useful when operating with custom scripts. Any variable that have been declared *public* will be exposed in the *Inspector*. This makes it possible to see and adjust variables at run-time. It also makes it simple to change initial values without having to open the script and locate correct variables.

Mapbox is an important external framework used throughout this project. Besides providing map visuals and building geometry, it also comes bundled with a series of other frameworks. The most significant of them being Json.NET [6]. This framework simplifies working with JSON files, and is particularly useful when handling the response from *Statens Vegvesen*.

Before looking into the core modules of the program, we will be taking a closer look at how *Mapbox* is set-up to function as desired. After being successfully imported to the project, *Mapbox* supplies a pre-made game object, called *Map*, that can be included in the scene. This game object has an *Abstract Map* script component that handles all general functionality provided by *Mapbox*. Figure 9 shows this component as it appears exposed in the *Inspector*.

The *General* settings are used to define the extent, scale and location of the map. *Longitude Latitude* is used to position the center of the map, with given coordinates corresponding to the center of Stavanger, as seen in Figure 10. *Zoom* defines the zoom-level of the map view and will for all purposes be kept constant throughout the program. A zoom level of 14 was chosen as it significantly reduces the amount of building geometry that has to be rendered. How detailed this specific zoom-level is can be seen in Chapter 4.2, where a set of sample scenarios are explained. The *Extent Options* have been set to only render the map around a specific *Transform*. By setting the player as the target transform, we can easily render map geometry exclusively around the player within some specified radius. Having a *Visual Buffer* of 3 results in a visible region of 3 by 3 tiles, centered around the player. A *Disposable Buffer* of 4 denotes that a maximum of $4 \cdot 4 = 16$ tiles can be rendered simultaneously. When the current amount of rendered tiles exceed this amount, old tiles will be unloaded. When *Snap Map To Zero* is toggled, it causes the center of the map to align with Unity's coordinate system. This means that the geographical coordinate with latitude 58.968 and longitude 5.7325 corresponds to the position $(0, 0, 0)$ in Unity. *Unity Tile Size* determines the scale of each tile in Unity. With a value of 5, this means that the side of each tile have a length of 5 units within Unity's coordinate system.

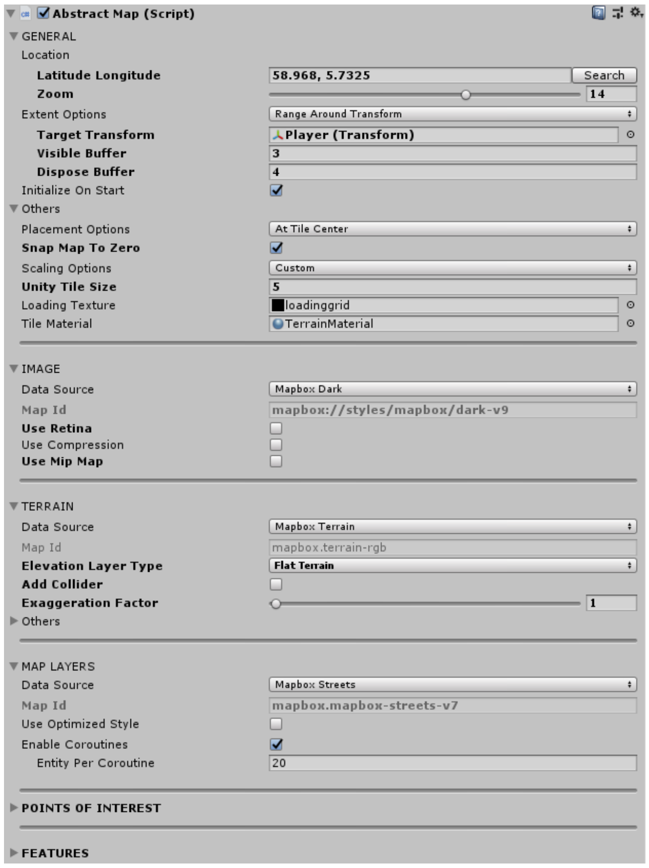

```
Abstract Map (Script)
GENERAL
  Location
    Latitude Longitude        58.968, 5.7325        Search
    Zoom                      14
  Extent Options              Range Around Transform
    Target Transform          Player (Transform)
    Visible Buffer            3
    Dispose Buffer            4
  Initialize On Start         ✓
Others
  Placement Options           At Tile Center
  Snap Map To Zero            ✓
  Scaling Options             Custom
  Unity Tile Size             5
  Loading Texture             loadinggrid
  Tile Material               TerrainMaterial

IMAGE
  Data Source                 Mapbox Dark
  Map Id                      mapbox://styles/mapbox/dark-v9
  Use Retina                  ☐
  Use Compression             ☐
  Use Mip Map                 ☐

TERRAIN
  Data Source                 Mapbox Terrain
  Map Id                      mapbox.terrain-rgb
  Elevation Layer Type        Flat Terrain
  Add Collider                ☐
  Exaggeration Factor         1
Others

MAP LAYERS
  Data Source                 Mapbox Streets
  Map Id                      mapbox.mapbox-streets-v7
  Use Optimized Style         ☐
  Enable Coroutines           ✓
    Entity Per Coroutine      20

POINTS OF INTEREST

FEATURES
```

Figure 9: *Abstract Map* as it appears exposed in the *Inspector*.

The *Image* settings controls the visual style of the rendered map. Satellite images are available, but for this project, we have decided to use a minimalist dark theme, which results in fewer distractions when observing the data.

The *Terrain* settings can be used to add 3-dimensional geometry to the terrain of the map. However, flat terrain will be used, as 3D terrain occasionally causes *clipping* issues when placing *Line Data* geometry.

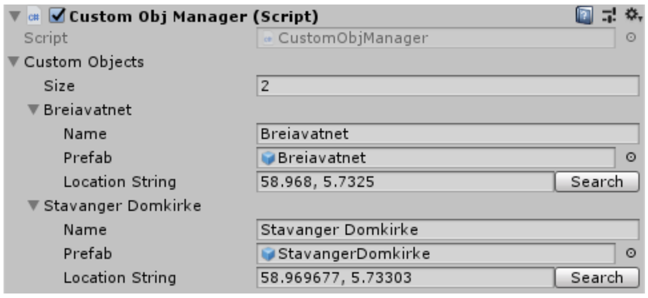
The *Map Layers* settings determines the data sources that will be used when rendering the map. Preferably the *Data Source* should be set to *Mapbox Streets With Building Ids.* This would have assigned each building a unique ID, which would have made it possible to replace and remove specific buildings. The problem with this data source is that very few building in Stavanger actually have an ID assigned. As a result, data source have to be set to *Mapbox Terrain* in order to get expected building geometries.

The *Features* settings are used to render special objects beside standard map visuals. This is where we can define how buildings should be visualized. Any building of type *cathedral* have been filtered out. For Stavanger, this only affects *Stavanger Domkirke*, which will be replaced by a custom 3D model.

A weird problem encountered with *Mapbox* was that the central lake in Stavanger, *Breiavatnet*, was missing. This could cause confusion among users, as it is a well-known landmark. A custom

Figure 10: Center of map as shown in *Google Maps*.

script was made to handle this issue, with exposed parameters in the *Inspector* as seen below.



```
Custom Obj Manager (Script)
Script                  CustomObjManager
▼ Custom Objects
  Size                  2
  ▼ Breiavatnet
    Name                Breiavatnet
    Prefab              Breiavatnet
    Location String     58.968, 5.7325          Search
  ▼ Stavanger Domkirke
    Name                Stavanger Domkirke
    Prefab              StavangerDomkirke
    Location String     58.969677, 5.73303      Search
```

This script spawns and maintains the position for a set of custom objects according to some geographical coordinates. Both *Breiavatnet* and *Stavanger Domkirke* was made in the 3D modeling tool known as *Blender* [2].

Map Manager. The *Map Manager* works as a bridge between the *Abstract Map* script, provided by *Mapbox*, and the rest of the system. It also houses a custom class called *Vector2Int*, which represents a 2-dimensional vector where both coordinates have integer values. Unity and *Mapbox* already provides *Vector2* and *Vector2d*, which operates with float and double values, respectively. However, integer vectors are necessary since floating-point numbers are unsafe as dictionary and hashmap keys.

Retrieving clamped player position

```
static Vector2d clampVal = new Vector2d(100, 50);

public Vector2Int GetTargetPositionClamped()
{
    Vector3 pos = targetTransform.position;
    Vector2d geoPos = map.WorldToGeoPosition(pos);
    int x = Mathd.RoundToInt(
        100 * Mathd.Round(geoPos.x * clampVal.x) /
        clampVal.x
    );
    int y = Mathd.RoundToInt(
        100 * Mathd.Round(geoPos.y * clampVal.y) /
        clampVal.y
    );
    return new Vector2Int(x, y);
}
```

The most important function of the *Map Manager* is to calculate the center of the tile the player currently occupies. This makes it possible to divide the data into sections based on tile location. The function that performs this operation can be seen above. This function should return latitude rounded off to the closest 0.01 and longitude rounded off to the closest 0.02. Latitude and longitude need to have different values for the data regions to appear as perfect squares. This is caused by *Mapbox* having different scaling on its latitude/longitude axes. The clamping calculation performed for the coordinates of Stavanger can be seen below.

$$\begin{aligned} \textbf{Latitude}: &\quad 58.968 \cdot 100 \approx 5897 \implies 5897/100 = 58.97 \\ \textbf{Longitude}: &\quad 5.7325 \cdot 50 \approx 287 \implies 287/50 = 5.74 \end{aligned}$$

For all possible situations, these two values will only have two significant decimals. By multiplying both values with 100, we can avoid floating-point numbers, and work with integer values instead. The position of surrounding tiles can then easily be calculated by using the equation below.

$$(lat, long) = (baseLat, baseLong) + [x, 2y]$$

Where x and y is the distance in tiles along the latitude and longitude axis, respectively. This makes it easy to calculate the keys for tiles surrounding the player, and only render a small subset of the data at any given time. Keys for the tiles adjacent to the center of Stavanger can be seen below.

⋱	⋮	⋮	⋮	⋰
⋯	(5896, 576)	(5897, 576)	(5898, 576)	⋯
⋯	(5896, 574)	**(5897, 574)**	(5898, 574)	⋯
⋯	(5896, 572)	(5897, 572)	(5898, 572)	⋯
⋰	⋮	⋮	⋮	⋱

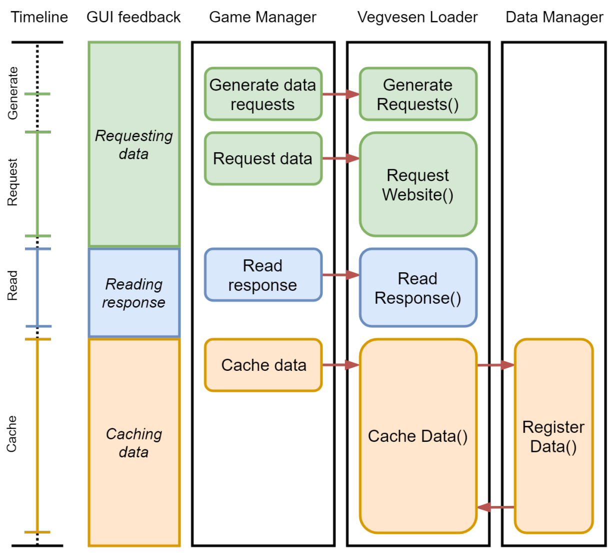


Figure 11: The coordination between the *Game Manager*, *Vegvesen Loader* and *Data Manager* during a data caching event.

Data Manager. The implementation of this component revolves around its unique design, which has been thoroughly described in a previous chapter. Most of its functions are for retrieving specific subsets of the stored data. There is also a function for caching data that takes a JSON object as an input parameter. However, retrieving and caching the data from *Statens Vegvesen* takes too much time for it to be a single operation. It would be difficult for users to determine whether something went wrong, or if the data is still being loaded. The script *VegvesenLoader* was written for this specific purpose. It defines a series of asynchronous methods for caching the data and functions as an intermediary between the *Game Manager* and the *Data Manager*. The coordination between these components during the caching of a single data set can be seen in Figure 11.

As seen in the figure, the *Game Manager* sequentially calls the methods defined within the *Vegvesen Loader*. Since these methods are asynchronous, the *Game Manager* await for each method to complete before continuing. This way, the *Game Manager* can simultaneously present the progress of the ongoing caching operation in the GUI. Feedback provided the user during the caching operation can be seen in the column *GUI feedback*. This entire operation can be divided into three sections, as shown by the colors: requesting data, reading the response and caching the data.

When caching a data set, two separate data requests have to be sent. One for receiving information regarding the data collection in general, and one for receiving all data of the given type. The response from the former request is, among other things, used when determining data category, *Line Data* or *Point Data*. Both requests can be seen below, in the order they were mentioned.

- https://www.vegvesen.no/nvdb/api/v2/vegobjekttyper/{ID}
- https://www.vegvesen.no/nvdb/api/v2/vegobjekter/{ID}?kommune=1103&inkluder=egenskaper,vegsegmenter,geometri&srid=4326&antall=5000

The second request is slightly more complex, because more specifications are needed to receive the expected data. *kommune* is the municipality we want to get data from, with 1103 being the municipality of Stavanger. *inkluder* is the additional data we want to have returned from the server. If this field is left empty: properties, road segmentation and geometry will not be contained within the response. An *srid* value of 4326 makes sure the coordinates are given on the format of WGS 84, or *World Geodetic System*. This is the expected coordinate system where locations are defined using latitude and longitude. *antall* decides how many entries should be contained within the returned response. By setting a significantly large value, we can be certain to always collect every entry belonging to Stavanger. The common variable for both requests, {ID}, denotes the ID of the desired data collection. This ID differ for each instance of the *Vegvesen Loader*. The IDs for some of the data collections can be seen below.

Data Collection	ID
Manholes	83
Speed Limits	105
Accidents	570
Tunnel Sections	60
Bridges	67

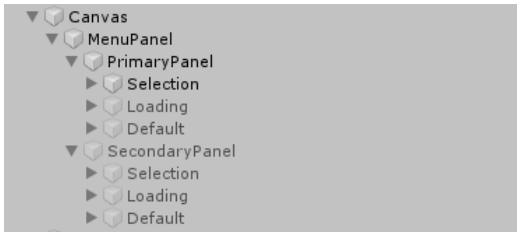
The responses from *Statens Vegvesen* are received as long sequences of bytes. On this format, the data is not very valuable. To make use of the data, the bytes first has to be interpreted as UTF8 strings. Depending on the size of the response, this operation could take a significant amount of time. However, it is always the fastest of the three sections.

Caching data

```
public IEnumerator CacheData()
{
    bool done = false;
    Thread _thread = new Thread(() => {
        JObject info = JObject.Parse(infoString);
        JObject dataRoot = JObject.Parse(dataString);
        roadDataManager.RegisterData(
            ID, info, dataRoot
        );
        done = true;
    });
    _thread.Start();

    // Wait for thread to finish
    while (!done) {
        yield return new WaitForSeconds(.1f);
    }
}
```

Before caching the data, we have to transform the UTF8 strings over to JSON objects. The problem is that the methods defined by the JSON library are not asynchronous, and so the entire game freezes until the operation finishes. This is unacceptable as it impairs the feeling of immersion using VR. To avoid this issue, most of the slow operations occur on a separate thread, as seen in the code excerpt above. A Boolean, *done*, handed over by the main thread ensures that progress does not continue until the data have been cached successfully.

Game Manager. The *Game Manager* operates on the programs presentation layer, or front end. User input, as well as GUI updates, are handled within this component. Unity handles GUI presentations and interactions through a *Canvas* game object. Children of this object assign screen boundaries to GUI exclusive text or graphics components. For the *Game Manager* to be able to locate all GUI elements, it have been given a reference to the root of this *Canvas*. The *Canvas* hierarchy, from layer four and up, can be seen below.



```
Canvas
  MenuPanel
    PrimaryPanel
      Selection
      Loading
      Default
    SecondaryPanel
      Selection
      Loading
      Default
```

The grayed out entries are currently disabled, but can be enabled through code whenever needed. This canvas can be divided into two sections: a primary panel and a secondary panel. The primary panel is more important, as it handles the selection and rendering of the primary data collection. It's also the only panel needed when performing comparisons between properties of a single data collection. The secondary panel is only active while performing cross-data comparisons. From the root of this hierarchy, the *Game Manager* can easily locate either of these panels. Two important functions provided by Unity, can be seen below.

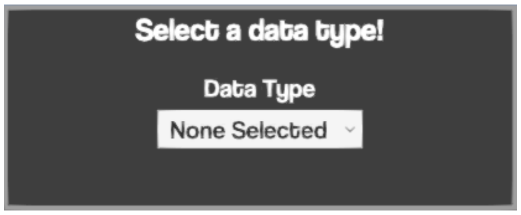
transform.GetChild(N)
transform.Find("...")

The former function allows us to get any child-transform at a given index, while the latter can be used to find a child-transform by name. And so, to get the primary panel from the *Canvas* root, we can use the code below.

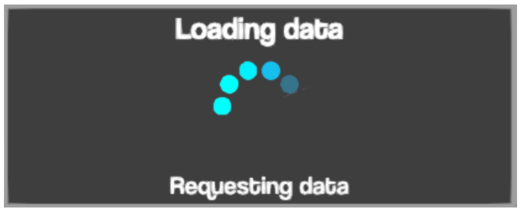
root.GetChild(0).Find("PrimaryPanel");

Only one of the sub-panels *Selection*, *Loading* and *Default* are active, at any given time, depending on the current situation. The *Selection* panel is the first GUI component presented. This is where the player can choose which data collection they want to be working with. If the collection have yet to be cached, the *Loading* panel will be displayed temporarily. The most important sub-panel is *Default*, where rendering and comparisons can be initiated. All sub-panels can be seen in Figure 13, as they appear in the GUI.

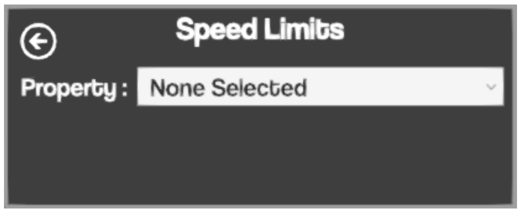
Both the selection and default panel presents a drop-down list to the user. For the *Selection* panel, the content of the drop-down is a list of available data collections. While for the *Property* panel, this list contains all properties available on the currently selected data collection. The text entries for the drop-down options come from widely different data structures. In the case of the *Selection* panel, the options comes from the variable *name* found within the custom class *DataDefinition*. The property drop-down options are, on the other hand, retrieved from a string array. A generic option



(a) Selection



(b) Loading



(c) Default

Figure 12: Primary sub-panels as displayed in the GUI.

generator function was made to support both situations, as seen below.

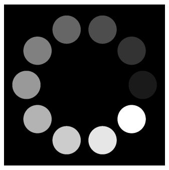
Generate options for dropdown

```
public List<Dropdown.OptionData>
CreateDropDownOptions<T>(
    T[] array, Func<T, string> GetVariable
)
{
    /* Create a set of option */
    List<Dropdown.OptionData> options =
    new List<Dropdown.OptionData>
    {
        new Dropdown.OptionData("None␣Selected")
    };

    foreach (T temp in array) options.Add(
        new Dropdown.OptionData(GetVariable(temp))
    );
    return options;
}
```

The function *CreateDropDownOptions* defines a generic variable type locally named *T*. As input variables it takes an array of *T* elements, but also another function. The supplied function is expected to accept a single *T* element and return a string. Both situations mentioned earlier can now be supported as seen below.

CreateDropDownOptions(*dataCollections*, c => c.name);
CreateDropDownOptions(*properties*, c => c);

The *Loading* panel is an intermediate panel that is shown between the *Selection* and *Default* panel. Besides showing info regarding the current status, it also displays a loading icon to ensure the user that progress is being made. This icon does not actually rotate. It's instead animated using custom graphics and shaders. The most important texture can be seen below.

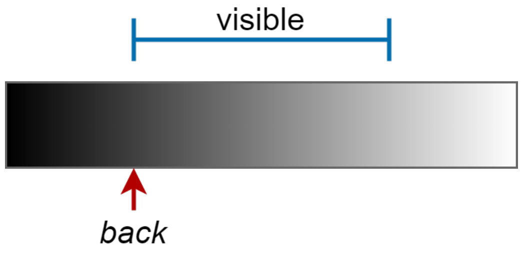


This image consists of ten smaller circles following the outline of one larger circle. The gray scale values in the image represents the alpha-values in the original texture. Each small circle in the sequence slowly increase in visibility. If we imagine this alpha value to start over at 0 after passing 1, there would be an equal difference in alpha values between lateral circles. In this particular case where there are ten circles, the difference can be calculated to be $1/10 = 0.1$.

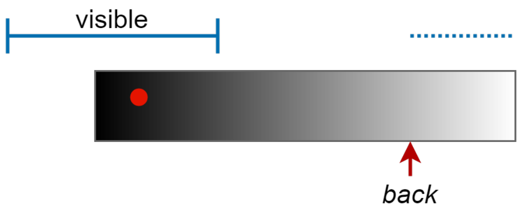
Shader excerpt for loading texture

```
fixed4 frag(v2f V) : SV_Target
{
  fixed4 c = SampleSpriteTexture(V.texcoord);
  c.rgb *= c.a;

  fixed back = _Time.y % 1;

  if (back > .5 && c.a < .5) back -= 1;

  if (c.a < back || c.a > back + .5) discard;

  fixed rel = (c.a - back) * 2; // 0 - 1

  return V.color * 3 * rel;
}
```

An excerpt from the shader used to animate the loading texture can be seen above. As a product of the current time, we define a *back* variable. This makes up the posterior of the visible alpha-space. If any pixel from the texture have an alpha value within the range $back < alpha < back + 0.5$, then it should be rendered.



The figure above shows the visible alpha range with *back* located in the lower half. If the alpha value of a pixel resides in the lower half of the scale, while *back* is currently located the upper half, potential issues could arise. For these situations, the location of *back* is recalculated as if it has wrapped around, as seen below.

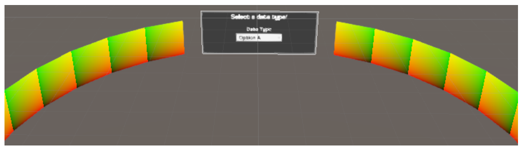


Any pixel outside the visible range is discard, i.e. not rendered to the loading icon. If a pixel have been determined to lie within the visible range, its relative visibility is calculated. This is a unit alpha value, where pixels closer to the location of *back* is less visible. As a result, the icon seems to be rotating, while in reality, we're only altering the alpha values of the individual circles. This gives a crisp animation with the need for only a single texture.

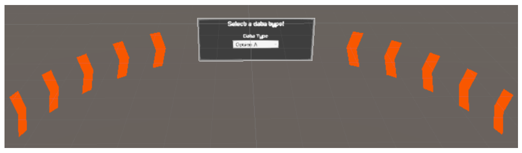
Another job handled by the *Game Manager* is making sure the user does not have problems locating the GUI. Unlike traditional computer programs, Virtual Reality does not go well with screen-space menus and interfaces. In fact, menu-panels with fixed positions within the user's viewport could be felt as uncomfortable, or even claustrophobic. The result is that GUI elements have to be placed in world-space instead. Moving the GUI around would only confuse the user, so the best option is to guide the user towards the intended view.

The approach for this project is to guide the user through arrows moving along the inside of a cylinder. Texture for a single arrow can be seen in Figure 13a. To get multiple horizontal replications of the arrow texture, the UV mapping is as seen in Figure 13b. The red and green values of the UV points towards a relative x and y

(a) Texture



(b) UV Mapping



(c) Result

Figure 13: Arrows guiding the user's view.

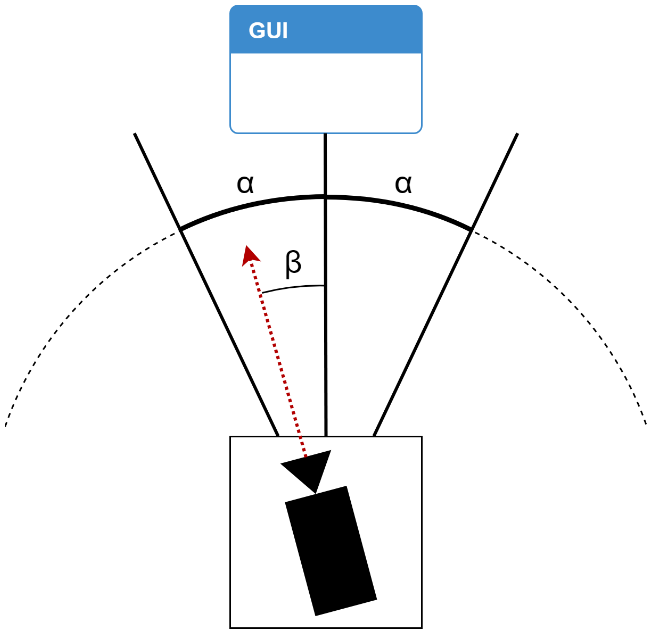


Figure 14: Calculating the alpha of arrows using camera rotation.

coordinate within the given texture, respectively. For both halves of the cylinder, the red values increase towards the GUI, but jump back to zero after passing a value of one. Figure 13c shows how the arrows render in world-space. To better guide the user, the arrows have been animated to slowly move towards the direction they are pointing. This has been accomplished by making the shader read texture coordinates with an increasingly negative displacement along the x-axis.

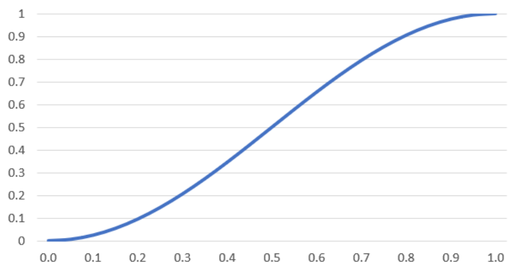
It is in our interest to only display the arrows whenever the player loses track of the GUI. Having the arrows animate while users maneuvers the menu would be distracting. To solve this issue, we have to calculate where the player is currently looking, as seen in Figure 14. The symbol β denotes the angle between the camera's forward vector, and a vector pointing directly towards the intended view. While β is smaller than a certain threshold, angle α, the arrows remains invisible. This gives the user some degrees of freedom, where they can look around without being guided towards the correct view. For any angle greater than α, we calculate an alpha value as seen below.

$$alpha = \frac{\beta - \alpha}{180 - \alpha}$$

For this calculation, we want to work with vectors that exclusively belongs to the XZ-plane. The player should be able to look up or down without influencing the visibility of the guiding arrows. We can achieve this by ignoring any y-component of the vectors before calculating the β-angle, as seen below.

$$v_2 = [v_x, 0, v_z]$$

By default, the increase of the *alpha* value is linear. This means that the user has to look more than 90 degrees away from the GUI before the arrows can reach half visibility. It would be much better if the arrows became visible quickly after passing the α-angle threshold. The constant speed of which alpha increase is also not desirable, as it makes the critical angles, *alpha* and 180, very apparent. We want the alpha to change less in value close to these critical angles. A much smoother function can be achieved by using properties of the cosine function, as seen below.



$$y = \frac{cos(x * \pi + \pi) + 1}{2}$$

For the specified region of the x-axis, this function closely resembles the sigmoid function. As we can see in this figure, the change in alpha values is much less significant towards the extremities. A more smooth transition has been achieved, but alpha value still stays below 0.5 until halfway through. This can be solved by using properties of exponentiation. When operating within the space $\in [0, 1]$, an exponent below one will "expand" the duration of larger values. The full transformation can be seen below.

$$y = \left(\frac{cos(x * \pi + \pi) + 1}{2}\right)^{0.6}$$

Figure 15 shows how the transformation looks like compared to the original linear increase in alpha. When the β-angle is now $\alpha + (180 - \alpha)/2$, the alpha value is at 66 percent, instead of 50 percent. The curving at the ends of the function has also been retained.

3.8 Render Manager Implementation

The *Render Manager* is the most complex of all the components. It handles rendering, plotting and the display of legends belonging to the data collections. As a result, it will be explained in its own separate subsection.

To minimize the number of re-calculations required by the *Render Manager*, it needs to keep track of its currently rendered region and player position. As long as the player remains within a single tile, there is no need to perform an update. By default, we render a 3x3 tile region around the user, and should the user move to another tile, data need to be re-drawn. However, as long as the player only move two tiles or less, there will be no need to perform a full update. Among the new and old tiles to render there will be overlapping entries, as seen in Figure 16.

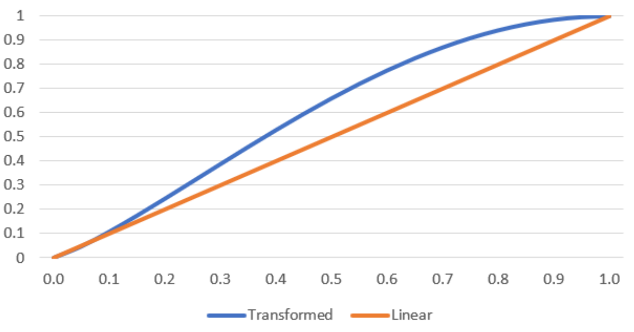


Figure 15: Increase in alpha value before and after transformation.

From the figure, we can see that the previous region was centered around (5897, 574). All currently rendered tiles have their position stored in a dictionary, which should never have more than nine elements. When rendering a new region, any old tiles that do not overlap are first unloaded. For this particular case, this means that the remaining tiles are (5897, 574), (5898, 574), (5897, 572) and (5898, 572). The new tiles can then be rendered and have their position added to the dictionary. In best case scenarios, only a single new row/column need to be rendered.

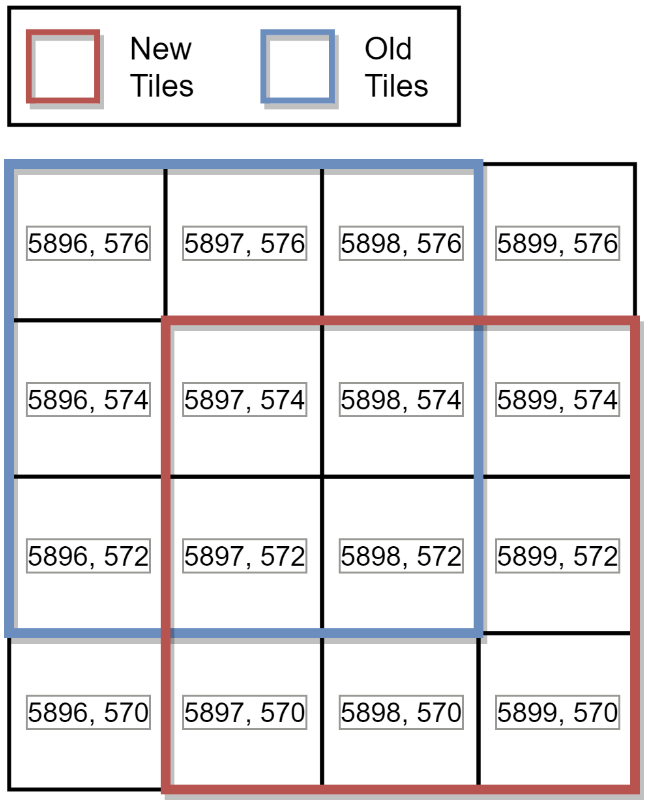


Figure 16: Tile overlap among old and new tiles.

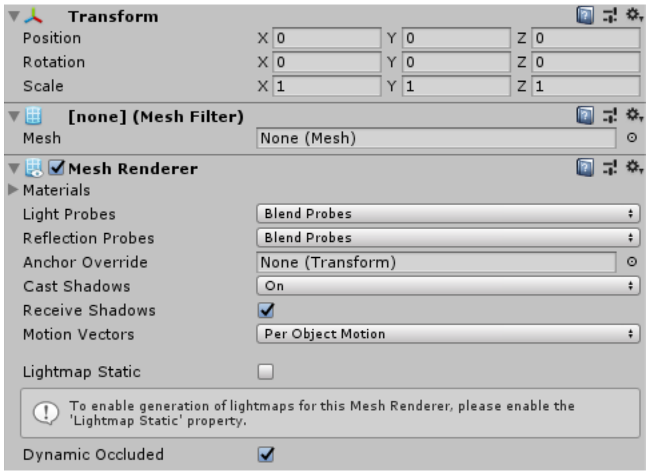
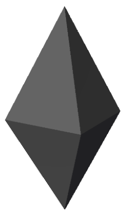

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
[none] (Mesh Filter)
Mesh None (Mesh)
Mesh Renderer
Materials
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Cast Shadows On
Receive Shadows ✓
Motion Vectors Per Object Motion
Lightmap Static ☐
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Dynamic Occluded ✓

Figure 17: Content of the inspector for prefab shared by *Point Data* and *Line Data*.

Whenever a tile is unloaded, all objects associated with that particular tile is added to a “pool”. We have represented this in Unity as a queue of *game objects*. These objects were either pointers or lines on the map and were deactivated before being *pooled*. Whenever drawing new pointers or lines, the *Render Manager* first consults this queue. As long as the pool is not empty, game objects are drawn from there, rather than being instantiated. This speeds up the rendering of new tiles, as spawning new *game objects* is a relatively heavy process.

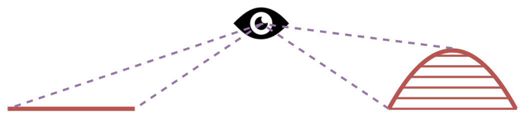
Point Data and *Line Data* are based on the same prefabricated game object. These game objects are referred to as *prefabs* in Unity. The only difference is the particular mesh being rendered. Figure 17 shows the content of the *Inspector* for this shared prefab. The *Transform* component is required for all game objects and is used to position an object within Unity’s coordinate system. Any information on how to render the game object, such as material and lighting, is handled by the *Mesh Renderer*. *Mesh Filter* is the only component that differs depending on the data category. The *Mesh* field is initially left empty but is assigned an appropriate mesh at run time.

Since the geometry of *Point Data* entries are identical, i.e. one single point, constructing their mesh is simple. Blender has been used to create a custom 3D model of a gem, as seen below.

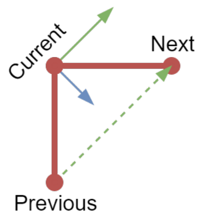


This model is particularly good for representing single points because of its low triangle count, of only eight triangles. Even with hundreds of data entries rendering simultaneously, there would not be a noticeable impact on the performance of the program. The mesh information of this 3D model is stored in the *Render Manager* and inserted into *Mesh Filter→Mesh* when placing new data on the map of type *Point Data*.

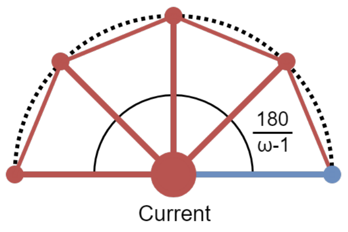
Line geometry is more complex to visualize as all entries are unique. They are made up of a series of non-linear points, rather than single geographical locations. This means that meshes need to be procedurally generated at run time. To take full advantage of the 3D environment, we will be representing this data category as arched lines. This will make the lines easier to observe from far away, as seen below.



They mesh generation for these lines have been handled in a separate script called *Line Renderer*. All information this script is given to construct the mesh is an array of coordinates. The construction of the lines' extremities are special cases, so we will focus on the procedure throughout coordinate 2 to $n - 1$. For this range, every coordinate lies between two other coordinates. The first step is to decide the relative direction of the line passing through the current point. This is to avoid artifacts where the line follows a 90-degree turn. A simple, but effective approach, is to use the vector passing from previous coordinate to next coordinate as seen below.



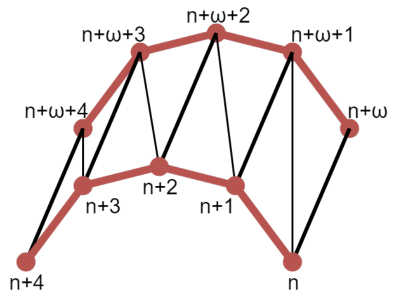
Next, we need to find the perpendicular vector (blue), which can be found by calculating the cross product of the local forward vector (green) and the global up vector. This vector point towards the first vertex in the arc, going counter-clockwise. The "fan out" of vertices can be seen below.



The length of the blue vector determines the radius of the line mesh and can be configured within the script. Location of all vertices can be calculated by rotating the blue vector around the forward vector. To obtain a full semicircle, we always rotate the blue vector by $180/(\omega - 1)$ degrees, where ω is number of vertices along the arc.

A higher ω-value will make the line smoother, but could impact performance.

All vertices are placed into an array in the order they are calculated. Knowing this, we can easily retrieve vertices of previous arcs, as seen below.



n denotes the array index for first vertex of **previous** arc. After each section of the line have been constructed, this value is increased by ω. Vertices by themselves do not produce a visible mesh, so we also need to define a set of triangular faces connecting them. For this particular case, each line-section can be divided into $\omega - 1$ rectangles, which can again be divided into two triangles. One such rectangle is the region formed by n, $n+1$, $n+\omega$ and $n+\omega+1$. The rectangle is split into two triangles by introducing a diagonal from n to $n+\omega+1$. To ensure that the triangles are facing outwards, we also need to consider "winding order". The three vertices making up each triangle need to be given in clockwise order. For the specified region, the triangles can be defined as seen below.

$$Triangle_1 = (n, n+1, n+\omega+1)$$
$$Triangle_2 = (n, n+\omega+1, n+\omega)$$

What makes the endpoints of the line different, is that they have access to either *Previous* or *Next*, but not both. This slightly changes the approach for calculating the forward vector. For the first point on the line, it's defined as *Next - Current*, while the last point on the line defines it as *Current - Previous*.

The full geometry retrieved from *Statens Vegvesen* is too dense. All points on the line do not need to be consulted to construct an appropriate mesh for the *Line Data*. For all road generations, the program intentionally skips every second point to increase the speed of computations. Examples of generated meshes can be seen in Figure 18. These are all real examples generated when visualizing the *Speed Limit* throughout Stavanger.

Just rendering lines and points would not convey a lot of information, except for the existence of data. By coloring objects depending on the value of their property, the data visualization becomes much more meaningful. Before we can do this, we first need to know the range of numeric properties. There also need to be a way to assign interpolated colors to textual data. The only way we can find the range of numeric properties is by iterating over all cached entries and find the minimum and maximum values. This could potentially be a heavy operation, and is only performed on-demand. The textual values are discrete, which means that we need a very different approach for defining an appropriate range.

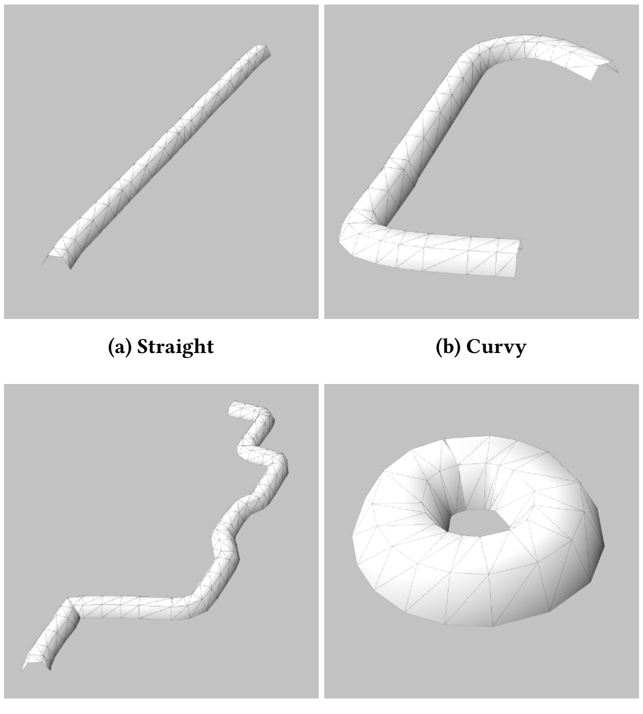
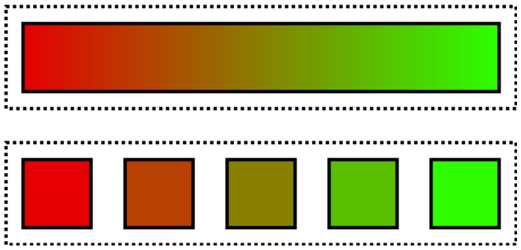

(a) Straight (b) Curvy (c) Very curvy (d) Roundabout

Figure 18: Example meshes generated by the *Line Renderer*.

Since all possible occurrences are known beforehand, we set each entry as keys to a dictionary. The respective values range from zero to one and are assigned incrementally. An example range for numeric and textual values can be seen below.



To increase the value of the data colorization, we need to show a legend. This is important as users need to be informed of the scale and range of data colors. Because of the big difference between property types, two separate types of legends need to be created: one for numeric properties and one for textual properties. Since one property type operates in a continuous range, and one operates in a discrete range, both layout and initialization differ.

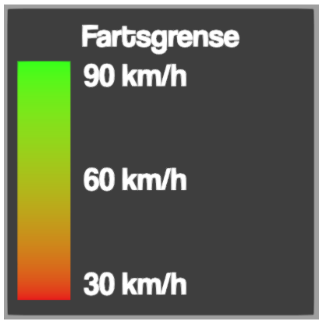
An example of a numeric legend can be seen in Figure 19a. This is from the property *Speed Limit* of the data collection *Accidents*. Aside from the legend title, there is a gradient rectangle and a display of minimum, middle and maximum values. These values were retrieved from the range calculated before visualization. The unit displayed with the value comes from the short-form cached unit for the specific property. For the gradient rectangle, a custom shader was made, interpolated between two colors depending on the vertical position within the figure. This is a much better approach than using textures, as there is no theoretical limit to the resolution. There will be no apparent pixelation, regardless of viewing distance, as seen below.

This is the middle of the numeric legend, as seen very close. The gradient still follows the exact same resolution as the display, while smoothed pixels can be seen around the edges of the text. This kind of shading is ideal for VR environments where the users view should not be obstructed, leaving them able to watch objects arbitrarily close. Not allowing free movement of the view would impacting the feeling of immersion.

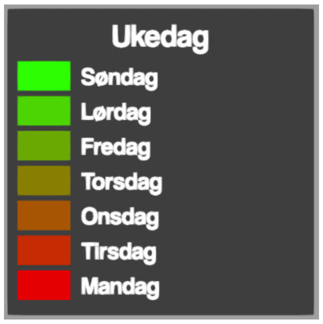
Figure 19b shows an example of a legend generated from a textual property. Similarly to the numeric legend, this property, *Weekday*, is from the data collection *Accidents*. For each entry in the legend, there is a colored tile and a corresponding label. From the bottom, these all follows the order they appear in the cache. The color of the tiles have been interpolated to have a similar gradient as the numeric legend.

Plotting is the most complex operation performed by the *Render Manager*, with three different permutations of property categories to consider and many data entries to consult. There is also a need for a different approach depending on whether we want to perform internal or cross-data comparisons. To evaluate the value of a 3D visualization environment, as opposed to the more common 2D environments, we also want to perform a three dimensional internal scatter plot. The graphical appearance of plots is the same for both internal or cross-data comparisons, so we will start by explaining how the plots are drawn before elaborating how the values are calculated. 3D plotting is very different to the 2D plotting approach and will be mentioned last.

Figure 20a shows an example of a scatter plot as it appears in the application. This plot displays a comparison between the depth and diameter of *Manholes* found in Stavanger. The *min* and *max* values along the axes are retrieved the same way as when creating a numeric legend. Both the scatter plot and line plot is based on a similar plotting technique, where coordinates of points are determined based on the respective values of each property. For instance, a one meter deep manhole with a diameter of 2 meter will be drawn at [2, 1]. It would be impossible to determine if two points overlap, so instead, we resize the existing point relative to the number of overlapping entries. This is a very intuitive way to display clustering. For the data from *Statens Vegvesen*, this is absolutely necessary, as actual values of properties have very little spread, despite technically being continuous. This can be seen clearly in the scatter plot, where all diameters are registered as either 0, 1 or 2 meters. For comparisons between numeric properties, the data collection associated with each axis is determined by which

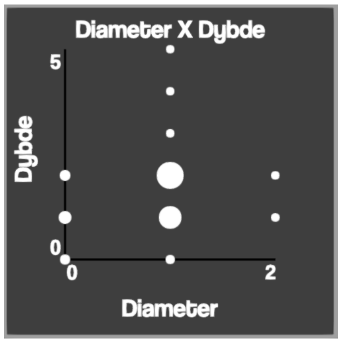


(a) Numeric

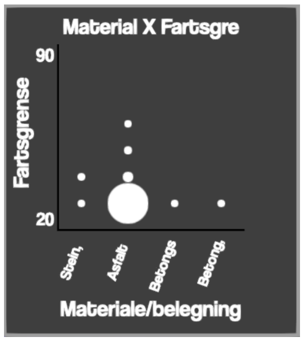


(b) Textual

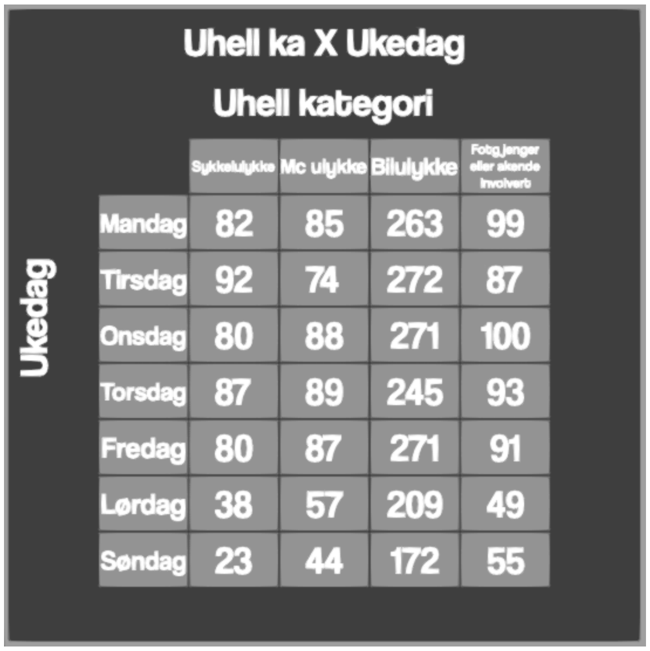
Figure 19: Examples of legends taken from visualizations of *Accidents*.



(a) Scatter plot



(b) Line plot



Uhell ka X Ukedag

Uhell kategori

Ukedag

	Sykkelulykke	Mc ulykke	Bilulykke	Fotgjenger eller akende involvert
Mandag	82	85	263	99
Tirsdag	92	74	272	87
Onsdag	80	88	271	100
Torsdag	87	89	245	93
Fredag	80	87	271	91
Lørdag	38	57	209	49
Søndag	23	44	172	55

(c) Matrix plot

Figure 20: Examples of plots from both internal and cross-comparisons.

order they were selected by the user. In this particular case, *Diameter* was the primary property, while *Depth* was the secondary property.

Figure 20b shows the line plot resulting from a cross-data comparison. The comparison is between speed limits, from *Speed Limits*, and material from *Speed Bumps*. In other words, this plot shows if there is any connection between speed limits and the preferred material used to make speed bumps. Despite being similar to the scatter plot, there is one feature that makes the line plot quite different. The x-axis has been assigned a set of labels, rather than a continuous range. This effectively makes the order of property selections irrelevant, as textual properties are always assigned the x-axis, and numeric properties the y-axis. The y-coordinate of each point is still derived directly from the value of the numeric property, while the x-coordinate is calculated using the equation below.

$$x = \frac{i + 0.5}{n}$$

Where n is the total amount of labels, and i is the index of a specific label, starting from zero. By adding 0.5 to the label index, we can ensure that x-coordinates is centered as expected. The result of this equation is a unit value, which can be multiplied with the width of the plotting region to properly place each data-point.

Figure 20c shows the plot resulting from the comparison of two textual properties. These properties are weekdays and accident categories from the data collection called *Accidents*. The approach for constructing this plot is quite different then the two previous ones. Before plotting, the program evaluates the numbers of labels belonging to each property. Rows are assigned to the property with the most entries. This improves readability as wider tiles are better at displaying text properly. The amount of labels belonging to each property is also used to calculate the size of each tile. For example, the width of the tiles in the given matrix plot was calculated as seen below.

$$width = \frac{w}{n + 1}$$

Where w is the total width of the plotting region, and n is the number of labels belonging to *Uhell kategori*. We have to add one to the number of labels to take into account the empty slot in the upper left corner. After performing the calculations, width and height are put into a grid component, which ensures that child game objects are resized to follow an exact grid. This grid component governs the content of the plotting region, where each tile is considered a separate game object. New children to the plotting region are added left to right, top to bottom. Using this knowledge, we can construct the first row by adding an empty game object, followed by tiles with each possible value from *Uhell Kategori*. The next rows of the plot have to be added using a double loop. For each possible values from *Ukedag*, we want to re-iterate over each possible value from *Uhell Kategori*, to retrieve the total number of incidents where these two property values coincide. Since both properties are textual, plotting the data cannot tell us anything about clustering, but it can display trends. For instance, using the plot, we can see that significantly less accidents happen in Stavanger on Saturdays and Sundays, regardless of the accident category.

The approach for comparing internal and cross-data properties has some similarities. For both situations, we want to compare two properties found on overlapping geographical locations. When working with cross-data comparisons, these two properties have to be found through a series of computations, making the process more complex than internal comparisons. With three possible permutations of property categories for both cross-data and internal comparisons, we're required to construct six unique functions. To avoid having to add huge amounts of extra code, we want to define a unique function for both internal and cross-data comparisons that can perform as much of the common operations as possible. We also want to avoid retrieving an array from this function, before iterating over the values, as it would require the program to iterate over all overlapping properties twice. The best approach comes in the form of *Actions*, which is a zero input delegate defined in Unity's API. Using this, we can define a *handle* for the generic function to treat each pair of properties differently. The generic function for performing internal comparisons can be seen at the start of the next page.

Retrieve internal properties

```
private void GenericCompSelf(
    int dataID, int propIdx1, int propIdx2,
    Action<Pair<Property>> GetPoint
)
{
    int idx = entryDict[dataID].entryIndex;
    DataPacket[] data = entries[idx].data;

    foreach (DataPacket dp in data)
    {
        Property[] props = dp.properties;
        Property p1 = props[propIdx1];
        Property p2 = props[propIdx2];

        if (p1 == null || p2 == null) continue;

        GetPoint(new Pair<Property>(p1, p2));
    }
}
```

A custom class called *Pair* was created for bundling two variables of the same generic type. This is necessary, as *Action* can only take a single datatype. And so, one of the input parameters of this function is another function that takes a pair of properties. For each data packet where both requested properties are defined, the supplied function is executed by calling *GetPoint*. How *GenericCompSelf* is called in the case of two numeric properties can be seen below.

Get points for scatter plot

```
public List<Vector2> GetScatterPointsSelf(
    int dataID, int propIdx1, int propIdx2
)
{
    List<Vector2> points = new List<Vector2>();
    GenericCompSelf(dataID, propIdx1, propIdx2, p =>
    {
        float val1 = ((NumericProperty)p.p1).value;
        float val2 = ((NumericProperty)p.p2).value;
        points.Add(new Vector2(val1, val2));
    });
    return points;
}
```

GetPoint is given as an anonymous function. This allows us to easily add each computed data point to a list within the same function that called *GenericCompSelf*. The only difference for line plots is that *p.p1* is cast to a text property instead of a numeric property.

Getting data for the matrix plot requires a slightly different approach. Throughout the entire computation, the program needs to keep a count for each possible property overlap. A special class, called *DataMatrix*, was developed to make this process faster and more robust. One of the main aims of this class is to avoid utilizing loops when counting occurrences. To keep track of each possible overlap, we use a simple array of integers. The length of this array is computed by multiplying the number of labels belonging to each property. For the situation seen in Figure 20c, the length of the array would have been calculated as seen below.

$$length = [Uhell\ kategori\ labels] * [Ukedag\ labels] = 4 * 7 = 28$$

To properly index into this array, we can use the equation given below.

$$x + y * width$$

Where x and y is the numeric index of a given x and y-label, and *width* is the total amount of x-labels. Finally, two dictionaries, called *xLabel* and *yLabel*, are made to assign an index to each individual label. Retrieving the current count, and counting a single entry, can then easily be performed as seen below.

Important DataMatrix functions

```
public int GetCount(string x, string y)
{ return occurences[xLabel[x] + yLabel[y] * width]; }

public void Count(string x, string y)
{ occurences[xLabel[x] + yLabel[y] * width]++; }
```

If we again refer to the matrix plot, we can verify the correctness of this design by performing an example. *Bilulykke* and *Tirsdag* would have been assigned the indices 2 and 1, giving the overlap an index within the *occurences* array of $2 + 1 * 4 = 6$. Counting left to right, top to bottom, starting from zero, we can see that index 6 indeed cover the expected overlap. The number of overlapping label between two arbitrary internal properties can then be computed using the function below.

Counting overlap for matrix plot

```
public void GetMatrixSelf(
    int dataID, int propIdx1, int propIdx2,
    DataMatrix matrix
)
{
    GenericCompSelf(dataID, propIdx1, propIdx2, p =>
    {
        TextProperty tp1 = (TextProperty)p.p1;
        TextProperty tp2 = (TextProperty)p.p2;

        matrix.Count(tp1.value, tp2.value);
    });
}
```

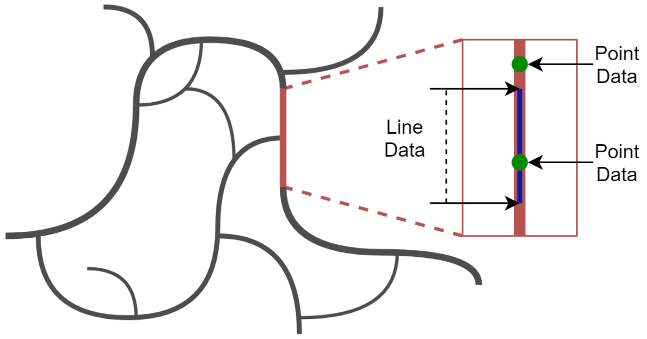


Figure 21: Visualization of overlapping *Point Data* and *Line Data*.

As before, *p* is the pair of overlapping properties sent as input parameter to the *GetPoint* function. By casting both properties to text, we can call the *Count* function of the *DataMatrix*.

GenericComparisonCross aims to perform a similar operation as *GenericComparisonSelf*, but because of the different data categories, this process is more complex. The function signature can be seen below.

Generic cross comparison

```
public void GenericComparisonCross(
    int dataID1, int dataID2, int propIdx1,
    int propIdx2, bool reverse,
    Action <Pair<Property>> GetPoint
)
{ ... }
```

Since the comparison regards two different data types, two sets of data IDs and property indices are required. A boolean *reverse* is also passed over to the function, which is only true if *dataID1* is **not** of type *Line Data*. This is used to make sure that the first data ID always belong to the *Line Data*. For each of the data IDs, we retrieve the full collection of *Data Packets* and all road occurrences. If the same road occurrence exists in both sets, this means that there is a chance of overlapping entries.

Figure 21 shows how the spread of point and line data on a single road could look. In this figure, there are two point data entries, but only one of them overlap the line data. We can also see from this figure that line data does not necessarily cover the entire road section. For each point, we first verify that they contain data regarding the expected property. If this is the case, the program checks to see if the point is covered by the line. As with internal comparisons, the overlapping properties are paired together before being sent to the *GetPoint* function.

3D plotting. All 2D plotting occurred on a plane in GUI space. This approach is no longer sufficient when plotting three-dimensional data, as there is a need for an additional dimension. The way data is loaded before creating this plot is very similar to how data was loaded for internal comparison of two properties, and so primary focus will be on graphical design and challenges.

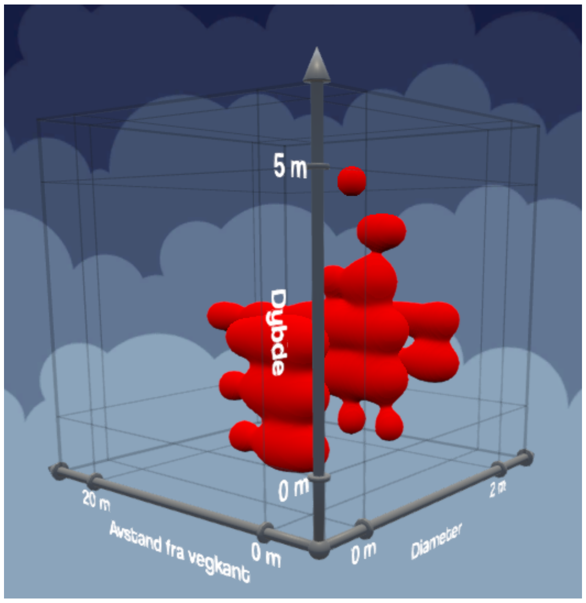


Figure 22: Three-dimensional scatter plot.

Figure 22 shows how a 3D plot appears in the developed software. Similarly to 2D plots, this plot also has labels and min/max values, but they are now assigned to **three** unique axes. The data points within this plot could have been represented by primitive spherical 3D models, but to introduce a more smooth *clustering* of data, a pure shader based approach was chosen.

A shader is a script that, as the name implies, defines the shading and colors of in-game objects. However, it can also be used to achieve post-processing effects, distortion and complex volumetric effects, such as fog and god rays. Everything in Unity has been assigned a shader that determines how each pixel they occupy should be drawn to the screen. An example of a simple geometry shader would be to return a single color with respect to lighting and surface normal. The entire domain seen in Figure 22 is actually a geometric cube, and with a shader as mentioned before, it would have rendered as a single red box. This is far from what is expected from this kind of plot. In order to achieve the final look, we utilize a technique known as *ray tracing*.

Ray tracing follows a quite different approach from standard shading. Instead of returning a color from the surface of the box domain, a ray tracing shader proceeds deeper into the volume of the mesh. This can be achieved by calculating world position and view direction at each respective pixel, as seen below.

Fragment shader component

```
fixed4 frag (v2f i) : SV_Target
{
    float3 worldPosition = i.worldPos;

    float3 viewDirection =
        normalize(i.worldPos - _WorldSpaceCameraPos);

    return raymarch(worldPosition, viewDirection);
}
```

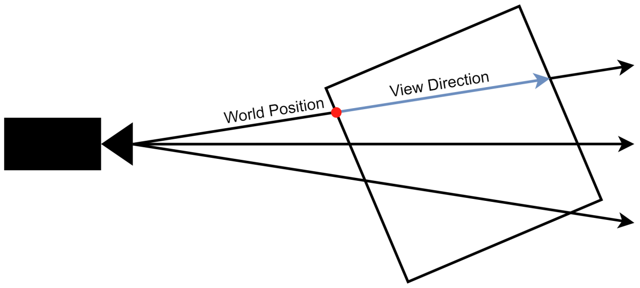


Figure 23: Illustration of ray tracing.

Figure 23 illustrates how world position and viewing direction could appear for a single pixel. Any three-dimensional geometry can now be defined mathematically within the boundaries of this domain. In a similar manner as for 2D scatter plot, this shader is supplied with an array of point data based on property values, as seen below.

Data array

```
int _PointsLength = 3;
float4 _Points[1000];
```

Data for this array is provided by another script. This script is also responsible for setting the value of *_PointsLength*. One of the limitations of arrays in shaders is that three-dimensional values, such as *float3*, is not available. As a result, *float4* has to be used as the data type. However, that is ideal for this situation as the forth value for each data point can be used to provide the radius of each sphere. Different radii on different spheres represent the presence of overlapping data points. To correspond with the world position calculated within the shader, the position of each data point also has to be given in world space.

Ray marching function

```
fixed4 raymarch(float3 position, float3 direction)
{
    for (int i = 0; i < STEP_COUNT; i++)
    {
        float distance = map(position);
        if (distance < .01) {
            return renderSurface(position);
        }

        position += STEP_SIZE * direction;
    }
    return fixed4(1, 1, 1, 0); // Clear
}
```

The function within the shader that handles the logic of the ray marching can be seen above. *STEP_COUNT* defines how many iterations each ray should have within the volume before returning an *alpha* value of 0. For each iteration where the ray is insufficiently close to the surface of a data sphere, the position of the ray is moved deeper into the volume with respect to *STEP_SIZE*. The *map* function calculates the distance between the current position of the ray and the closest surface of a sphere. This is made possible through the mathematical theory of *signed distance functions*. These functions can become quite complex, but remain relatively constant across rendering software. The two SDF methods seen below have been adapted from a post by *Inigo Quilez* [7].

SDF methods

```
float smoothUnion(float d1, float d2, float k = 0.1)
{
    float h = clamp(0.5 + 0.5*(d2 - d1) / k, 0, 1);
    return lerp(d2, d1, h) - k * h*(1.0 - h);
}

float sdf_sphere(float3 p, float4 data)
{
    return distance(p, data.xyz) - data.w;
}
```

The function *smoothUnion* makes it possible to blend together two different signed distance fields with a given weight *k*. *Sdf_sphere* returns the distance to the surface of a sphere, and as mentioned earlier, the *data* variable represents a single data point, with *data.xyz* being the coordinates and *data.w* being the radius. Both these SDF methods make up most of the mapping function seen below.

Mapping function

```
float map(float3 p)
{
    float smooth = sdf_sphere(p, _Points[0]);

    // Smooth successive spheres
    for (int i = 1; i < _PointsLength; i++) {
        smooth = opSmoothUnion(
            smooth,
            sdf_sphere(p, _Points[i])
        );
    }

    return smooth;
}
```

If there is only a single data point available, the function simply returns the distance to the surface of that point. For successive point data, their signed distance functions are smoothed together using the *smoothUnion* method. When the marching ray eventually approaches sufficiently close to a mathematical surface, here defined as 0.01, a color is calculated using the *renderSurface* function seen below.

Rendering of the surface

```
fixed4 renderSurface(float3 p)
{
    float3 n = normal(p);
    return simpleLambert(n);
}
```

Proper shading is necessary to better show details of the plotted data. Simply returning a pre-defined color would have made it difficult to distinguish convex and concave curves. However, before any shading can proceed, the normal vector of each surface first need to be calculated. This calculation turns out to be quite complex as there is no way to easily extract the normal vector from the mapping function. The mapping function can only express how far away from any surface a point is, not the actual direction towards that specific surface. A simple solution is to utilize a three-dimensional gradient descent, as seen below.

Calculating surface normal

```
float3 normal(float3 p)
{
    const float eps = 0.01;

    float x_delta = map(p + float3(eps, 0, 0))
        - map(p - float3(eps, 0, 0));

    float y_delta = map(p + float3(0, eps, 0))
        - map(p - float3(0, eps, 0));

    float z_delta = map(p + float3(0, 0, eps))
        - map(p - float3(0, 0, eps));

    return normalize(
        float3(x_delta, y_delta, z_delta)
    );
}
```

This gradient decent maps positions a little bit before and behind each three-dimensional axis: X, Y, and Z. The gap between the positions is determined by a very small *eps* value. A normal vector can then finally be calculated using the differences in values along each axis. For this approach to work as intended, it is important that there are no sharp edges throughout the geometry. This is no problem for this particular situation as all data points are represented as spheres. Because the mapping function has to be executed a total of six times, the *normal* method can become very intensive. The best way to combat this issue is by reducing the amount of supplied data points. One way this has been achieved is by merging together overlapping data points and instead increasing the radius of a single sphere. With the normal successfully calculated, lighting can be applied to the pixel using the function below.

Lambert lighting

```
#include "Lighting.cginc"
fixed4 wrapLambert(fixed3 normal) {
    fixed3 lightDir = _WorldSpaceLightPos0.xyz;
    half NdotL = dot(normal, lightDir);
    half diff = NdotL * 0.5 + 0.5;
    half4 c;
    c.rgb = _MeshColor * _LightColor0.rgb * diff;
    c.a = 1;
    return c;
}
```

Including *Lighting.cginc* in the shader gives access to a series of lighting related variables supplied by Unity. One such variable is *_WorldSpaceLightPos0*, which represents the position of the primary light source in world space. The calculated *NdotL* expresses the cosine of the angle between the normal of each pixel and the light direction. A normal directly towards the light gets a value of one, and at 90 degrees off it gets a value of zero. If we calculate *c.rgb* using this value instead of *diff*, it would correspond with standard Lambertian lighting models. However, this often results in extremely dark shadows as unlit normals would return completely black pixels. By modifying the *NdotL* value to range from 0.5 to 1, even unlit surfaces will have some degree of lighting. The result is that lighting seems to *wrap* around the surface of the geometry. This is a good way to simulate ambient lighting and gives the final scatter plot a much *softer* look.

4 EXPERIMENTAL EVALUATION

In this section, we will try out the finished software on three sample scenarios and evaluate the produced results. Images from the process will be taken from the perspective of the user. Focus will primarily be on showing and explaining the significant steps of each scenario, which is when new menus or objects are being drawn to the scene. We will also be discussing the computed results, and evaluate if they seem to be as expected.

4.1 Experimental Setup

The Unity project is used to construct a Windows 10 executable build, which is run on a display with resolution 2560x1600. First scenario can be seen below.

1. User selects *Tunnel Sections.*
2. First property is set to *Åpningsår.*
3. User selects self comparison.
4. Second property is set to *Bredde.*

This will test rendering of line data, numeric legends, internal comparisons and plotting of numeric properties.

The second scenario is more complex, to cover more features, and can be seen below.

1. User selects *Manholes.*
2. Property is set to *Bruksområde.*
3. User selects cross comparison.
4. User selects *Speed Bumps* as second property.
5. User is informed that *Point Data* cannot overlap.
6. User selects a new second property, *Speed Limits.*
7. Second property is set to *Fartsgrense.*
8. User compares the two different data collections.

This will test the functionality of cross-comparisons, legends of text properties and rendering of point data. It will also show the plot produced when comparing *Point Data* and *Line Data.* By intentionally selecting two data categories that do not overlap, we also get to test out the error message displayed when cross comparing is impossible. When two properties are successfully selected from different data sets, we will also be able to test out simultaneous rendering. The third and final scenario can be seen below.

1. User selects *Accidents.*
2. First property is set to *Antall lettere skadet.*
3. User selects 3D scatter.
4. Second property is set to *Antall kjørefelt.*
5. Third property is set to *Temperatur.*
6. User move across the map to see rendering update.

Not much attention will be given the first few steps as they are similar to the ones in the previous scenarios. The main functionality that will be tested in this scenario is the 3D scattering plot.

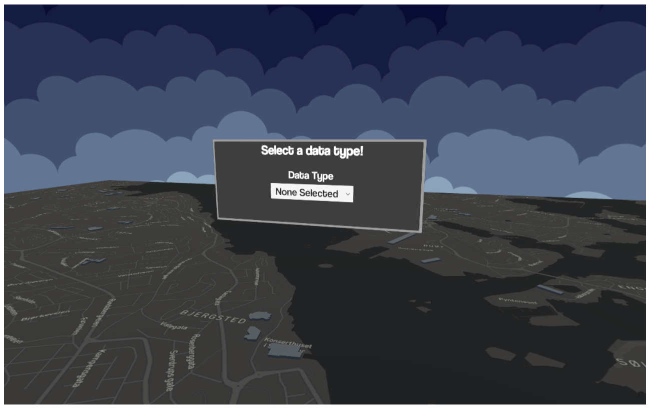


Figure 24: Initial menu, as seen by the user.

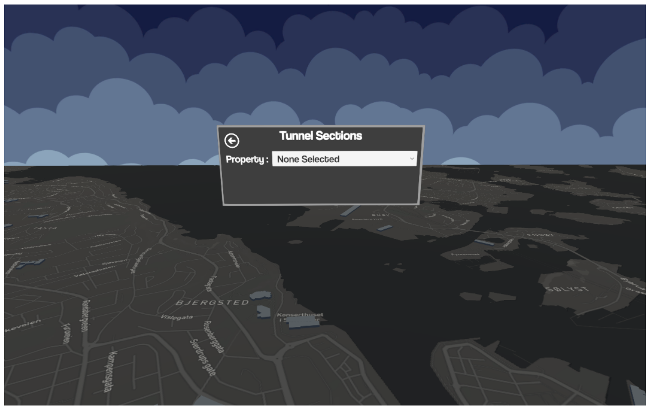


Figure 25: After selecting *Tunnel Sections*.

Data should be plotted as intended in a three-dimensional domain. Pictures will be shown from different angles of the plot to show how different it look when moving around in 3D space. We also want to test that the rendered region follows the user as intended.

4.2 Results

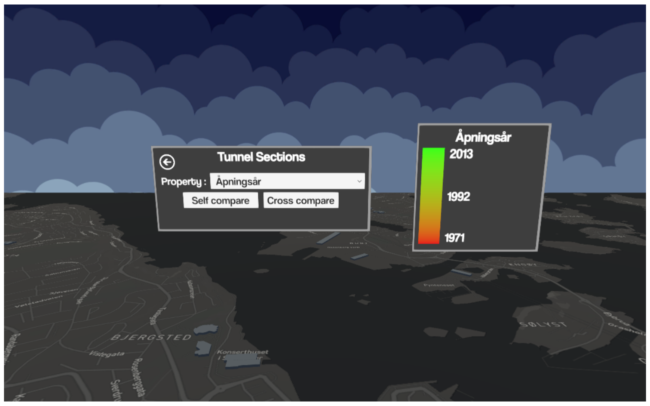
Figure 24 shows the menu initially displayed at the start of every scenario. To give the user an idea of their currently selected geographical location, the map already shows a section of central Stavanger. The aesthetics of the program follows a minimalist style, which also holds true for the skybox. We will start by following the steps of our first scenario.

After selecting *Tunnel Sections*, the user is eventually presented with another menu, as seen in Figure 25. Notice how no data is currently being rendered to the scene. Since no property is selected, there is no way to properly interpolate colors on the various data points/lines. Rendering data at this point would only be able to visualize the presence of data or lack thereof.

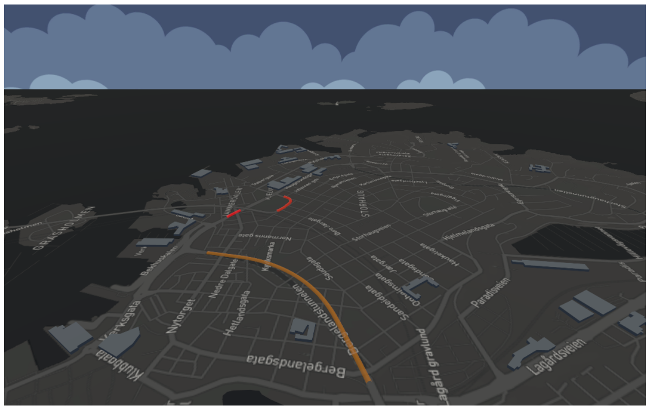
Figure 26 shows how the scene change after setting first property to *Åpningsår*. The menu titled *Tunnel Sections* is still there, but there are now two buttons visible, *Self compare* and *Cross compare*. To the right of this menu, a numeric legend have appeared. This legend informs us that the values of this property spans from 1971 to 2013, for the region of Stavanger. In other words, *Statens Vegvesen* does not have any data on new tunnels opening in Stavanger prior to the year 1971. No data can still be seen towards the north, which is caused by the scarcity of this particular data collection. A look towards the east, as seen in Figure 26b, can confirm that there are in fact data being rendered to the scene. The longest tunnel, called *Bergelandstunnelen*, opened in 1989. This year corresponds well with the interpolated color given this particular section.

When the user interacts with the *Self compare* button, the menu introduces a secondary drop-down, as seen in Figure 27. This is where the user can select which secondary property to compare against. The property *Åpningsår* still exists within this list to keep indices functioning as intended, however, no plot is shown unless properties are unique.

Setting the second property to *Bredde* produces a plot as seen in Figure 28. This plot shows a clear trend regarding the width of tunnel tubes. There could be many explanations for this, almost linear, decrease in width of newly constructed tunnels, one being the unfortunate lack of data. Since there are only nine entries with both properties defined, the plot is quite scarce. However, the most likely source for this result is a mixture of new technology and



(a) North



(b) East

Figure 26: After setting first property to *Åpningsår*.

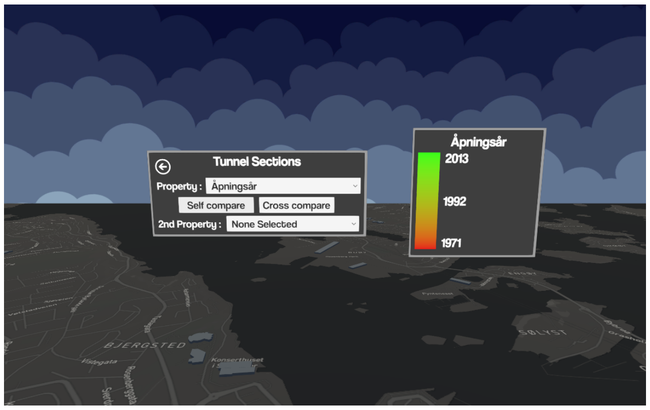


Figure 27: Update to menu after pressing *Self compare*.

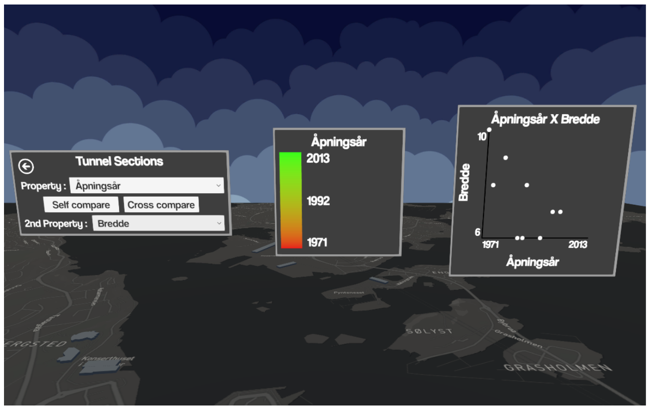
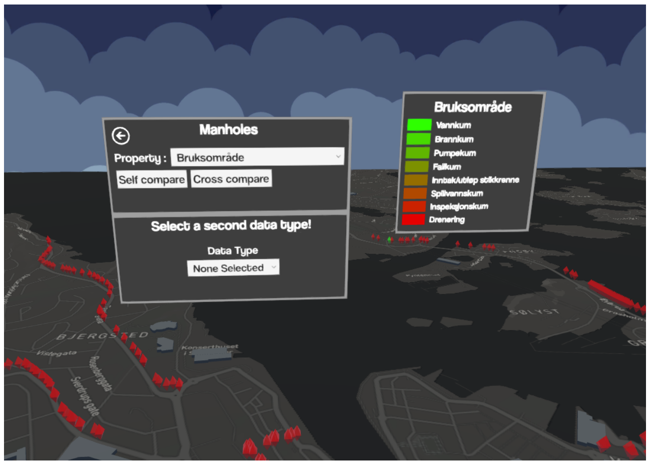
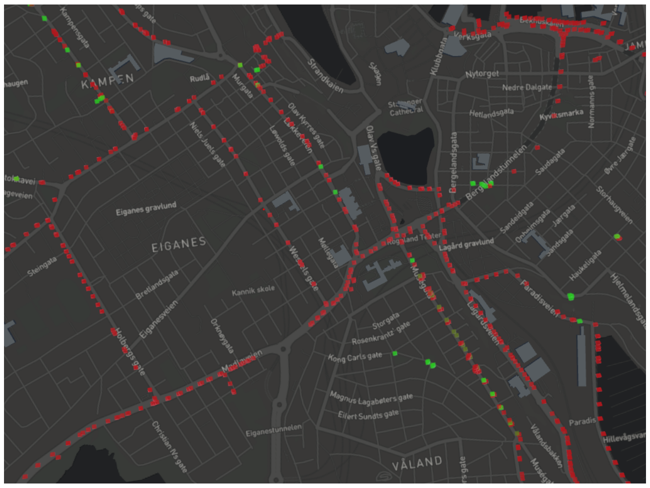


Figure 28: Comparison of *Åpningsår* and *Bredde*.



(a) Front



(b) From above

Figure 29: After clicking cross comparison.

economy. Keeping the width of tunnels at a minimum is extremely favorable from an economic perspective. The smaller the cross section of the tunnel tube is, the fewer explosives are needed.

For the second scenario, the first two steps are very similar to previous ones. Figure 29a shows the player view immediately after interacting with the *Cross Compare* button. As seen in the figure, a second data selection panel appears right under the initial one. This is where the player can select which data and property to compare against. The legend in this situation is slightly different from the one seen in the first scenario. One color is assigned to each possible value of the property *Bruksområde*, instead of a gradient between minimum and maximum values. Since this data type is covering manholes in the municipality of Stavanger, there is much more data than for tunnel sections. Figure 29b shows an overview of the data points located at the center of Stavanger. Many manholes in Stavanger, specifically from smaller roads, does not appear on this data visualization. These have not been truncated, they are simply not provided by *Statens Vegvesen*. Even with this much data, there is no impact on performance as only a small section around the user is being visualized at any given time.

The second panel operates in a very similar fashion to the first one. A dropdown menu lets the user select another data type that they want to visualize over the map. This list of data collections is not filtered, so it's fully possible for the user to select two incomparable data categories, such as *Point Data* and *Point Data*. If the selected data collection have not been cached, the loading screen will show until data have been loaded properly.

After selecting *Speed Bumps* as secondary data collection and *Materiale/belegning* as secondary property, the view is as seen in Figure 30. A red notification informing the user of incomparable data collections can be seen on the secondary panel. However, this does not stop the user from visualizing two different collections of *Point Data* simultaneously. Beside the original points, ranging from green to red, a new set of points, ranging from blue to purple, can also be seen on the map. The color of these points follows the same interpolation technique as the primary data points. This feature can

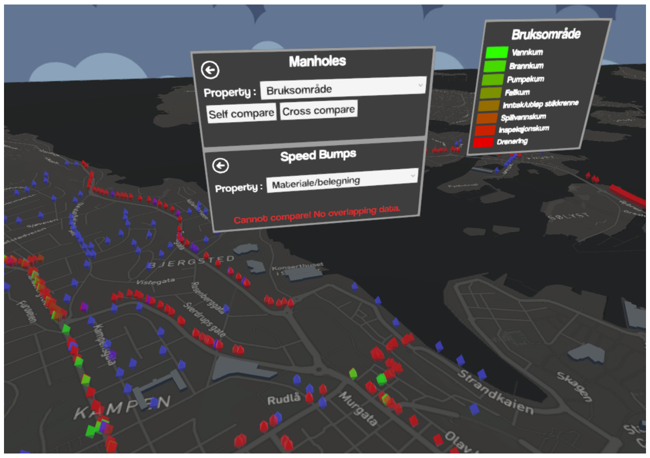


Figure 30: Cross comparison between *Manholes* and *Speed Bumps*.

be utilized by the user to cross-analyze two different collections of data despite them difficult to compare.

Since *Manholes* and *Speed Bumps* cannot be compared, the second data collection is changed to *Speed Limits*. The targeted property is set to be *Fartsgrense*. Figure 31 shows the scene after the user have clicked the button *Compare* that appears at the very bottom of the secondary panel. Also when cross-comparing with compatible data collections, both collections are visualized over the map simultaneously, as seen by the dense network of blue-purple lines. The produced plot attempts to compare usage of manholes with the speed limit on the road where they are located. This plot shows that the primary function for manholes on roads with higher speeds is to prevent unwanted pooling of water. We can also see that the majority of manholes in the municipality Stavanger is used for drainage.

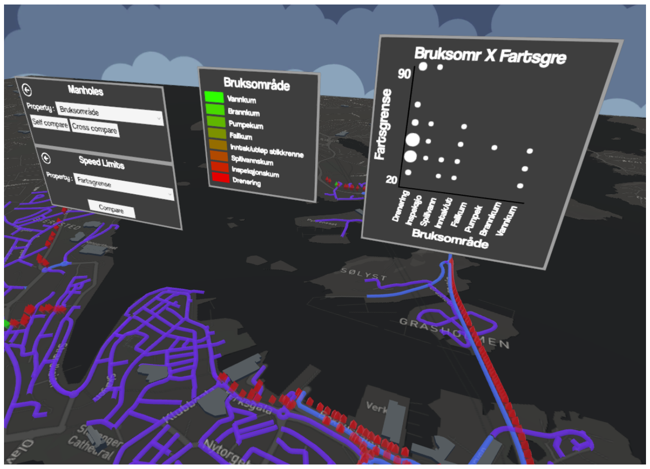


Figure 31: Cross comparison between *Manholes* and *Speed Limits*.

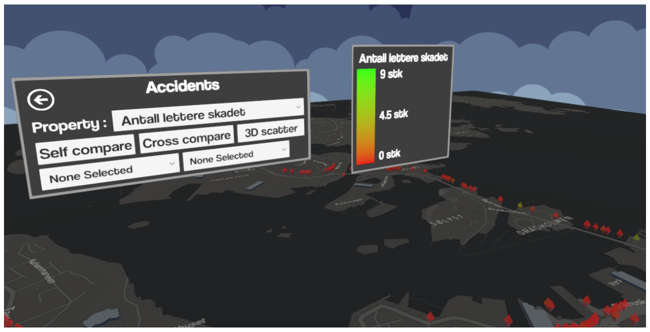
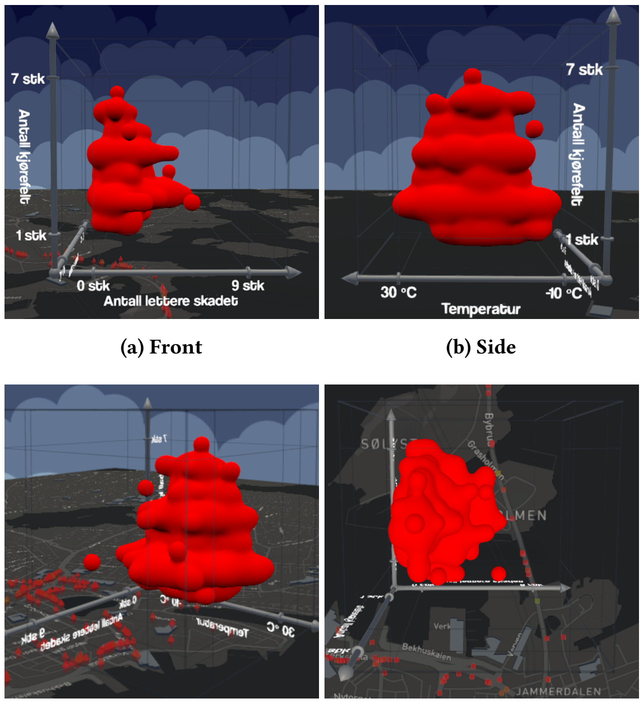


Figure 32: After clicking *3D scatter*.

Step one and two of the final scenario follows a very similar process as earlier scenarios. Figure 32 shows the user's view immediately after selecting *3D scatter* on step three. For limiting the scope of this project, *3D scatter* is only available among numeric properties, and so the specific button only appears when primary property is numeric. The two new drop-down menus underneath the buttons are used to select other properties to compare against. Unless they all have unique numeric properties selected, no plot will be generated.

Setting second and third property to *Antall kjørefelt* and *Temperatur*, respectively, generates the plot seen in Figure 33. Each axis properly displays the property label as well as minimum and maximum values. From each angle, it's very clear which correlation the



(a) Front **(b) Side**

(c) Back **(d) Top**

Figure 33: 3D scatterplot from third scenario.

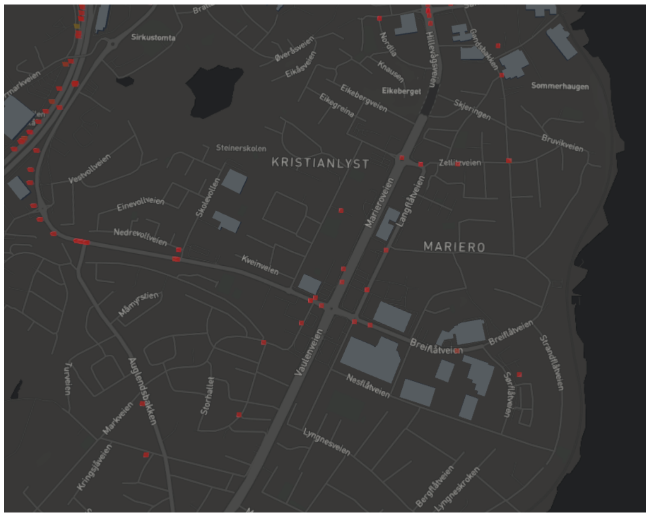


Figure 34: Accidents near Mariero.

different properties have with one another. Figure 33a shows that more people are lightly hurt when accidents occur on roads with fewer lanes. This could be the result of insufficient data, but it does not seem too unrealistic as more lanes generally are considered safer. Figure 33b shows how the amount of lanes and temperature during and accident in Stavanger is related. As expected, there does not seem to be any connection between these two properties. The data points appear to be distributed evenly from this angle, with more data at the bottom caused by a higher frequency of roads with fewer lanes. Figure 33c and Figure 33d shows how the plot looks like from the back and top, respectively. Different angles help visualize different connections between the involved properties, and the shading makes is easier to make out the shape of the plot.

The final step of the third scenario is to test that the map updates the rendered region around the user as intended. Figure 34 shows which data is available as the user approach Mariero. This area is far enough away from the center of Stavanger to be outside of the initially rendered region. However, as seen in the figure, the rendered region have moved to cover the new area around the player.

5 CONCLUSION

We believe that the use of game engines to visualize and compare data, as explained in this paper, is a satisfactory and very powerful tool. The use of three dimensions opens up for more complex plots than its two-dimensional alternatives. The fundamental structure of the program was specifically designed to allow for easy integration of additional property types and data sources. This would make it possible to compare any kind of property across many different sources. Such cross-source comparisons could provide extremely valuable information that would be time-consuming, if not necessarily difficult, to collect otherwise.

Three-dimensional visualization could initially seem more complex than 2D, but Virtual Reality provides numerous advantages. One of them being better GUI management. With virtual reality, it becomes possible to manage menus and legends in 3D space, which creates less clutter. These menus can remain out of view, so more focus can be given to the observation of data. Virtual reality is also arguably more intuitive. Allowing users to lean closer/away, instead of scrolling, and physically look around, instead of panning.

Next step. As a result of time constraints, only data from *Statens Vegvesen* was implemented. Next step could be to introduce more varied data from completely different sources. Additional property categories, such as *Date* and *Geometry*, could also be implemented but would have to be supported with many new types of plots. Another relevant feature that could be implemented is better filtering of data. For instance, the possibility to exclusively visualize data within a given span of time could provide valuable feedback.

REFERENCES

[1] DOMO. 2018. Data Never Sleeps 6.0. https://www.domo.com/learn/data-never-sleeps-6
[2] Blender Foundation. 2018. Blender.org - Home of the Blender project - Free and Open 3D Creation Software. https://www.blender.org/
[3] Ralph Jacobson. 2013. 2.5 quintillion bytes of data created every day. How does CPG Retail manage it? https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/
[4] Blazej Kot, Burkhard Wuensche, John Grundy, and John Hosking. 2005. Information visualisation utilising 3D computer game engines case study: a source code comprehension tool. *Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: making CHI natural* (2005). https://doi.org/10.1145/1073943.1073954
[5] Jae-Gil Lee and Minseo Kang. 2015. Geospatial Big Data: Challenges and Opportunities. *Big Data Research* (2015). https://doi.org/10.1016/j.bdr.2015.01.003
[6] Newtonsoft. 2018. Json.NET. https://www.newtonsoft.com/json
[7] Inigo Quilez. 2018. Inigo Quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
[8] Dean Takahashi. 2017. The U.S. game industry has 2,457 companies supporting 220,000 jobs. https://venturebeat.com/2017/02/14/the-u-s-game-industry-has-2457-companies-supporting-220000-jobs/
[9] Google Trends. 2018. https://trends.google.com/trends/explore?date=all&geo=US&q=%2Fm%2F0dmyvh,%2Fm%2F025wnp,%2Fm%2F02ph70
[10] Unity. 2009. Unity Technologies Renames Unity Indie to Unity and Makes It Freely Available. https://unity3d.com/company/public-relations/news/unity2.6-press