



***A Digital Twin Tale –
The Portland TriMet Network
for Public Transportation***

*How real-time 3D engines
could help building visual experiences
for spatial phenomena...*

*Project
UNIGIS Professional
Paris Lodron University of Salzburg*

Supervision Dr. Christoph Traun

*Winfried Schwan
24.11.2022*

Table of Contents

Abstract.....	6
Acknowledgements.....	7
Foreword.....	8
1. Introduction.....	10
1.1. Motivation.....	10
1.2. Objectives.....	12
1.2.1. GTFS Realtime Data Feed.....	12
1.2.2. GTFS Realtime Data Loader.....	12
1.2.3. ArcGIS Maps SDK for Unity.....	12
1.2.4. ArcGIS Pro.....	13
1.2.5. Unity.....	13
1.3. Mapping “Digital Twin Land”.....	14
1.4. Chapter Overview.....	16
1.4.1. Digital Twins.....	16
1.4.2. General Transit Feed Specification (GTFS).....	16
1.4.3. TriMet Open Data.....	16
1.4.4. Hardware Environment.....	16
1.4.5. Software Environment.....	16
1.4.6. A Digital Twin of Portland Public Transport.....	16
1.4.7. Portland Public Transport Traffic Live – Digital Twin Concepts Applied.....	17
1.4.8. Lookout Mountain.....	17
2. Digital Twins.....	18
2.1. The Early Years of Digital Twins.....	18
2.2. Digital Twin Definitions.....	19
2.3. Benefits of Digital Twins.....	21
2.3.1. Design.....	21
2.3.2. Maintenance.....	22
2.3.3. Training.....	22
2.3.4. Simulation.....	23
2.4. Digital Twin Examples.....	23

2.4.1. Vancouver International Airport	24
2.4.2. Realtime Bitcoin Globe.....	24
2.4.3. Wind Turbine Immersive Digital Twin	25
3. General Transit Feed Specification (GTFS).....	26
3.1. Overview	26
3.2. GTFS Schedule.....	26
3.2.1. Mandatory GTFS Files.....	27
3.2.2. Optional GTFS Files.....	28
3.3. GTFS Realtime	28
3.3.1. Overview	29
3.3.2. The Protocol Buffer Format and GTFS Realtime	30
3.3.3. Alternative Formats as JSON and XML	30
4. TriMet Open Data	31
4.1. Tri-County Metropolitan Transportation Agency (TriMet)	31
4.2. TriMet GTFS Data	31
4.2.1. TriMet Vehicle Location Service.....	32
4.2.2. TriMet Stops Data	33
4.3. TriMet GIS Data.....	34
4.3.1. ArcGIS Pro Preparation	35
4.3.2. TriMet Boundary	36
4.3.3. TriMet Routes.....	37
4.3.4. TriMet Stops.....	38
4.3.5. Derived TriMet MAX Routes	38
4.3.6. Derived TriMet MAX Stops.....	39
5. Hardware Environment	41
5.1. Development Laptop #1.....	41
5.2. Development Laptop #2.....	41
6. Software Environment.....	42
6.1. Postman API Platform	42
6.1.1. Overview	42
6.1.2. Workflows	42
6.2. Microsoft Visual Studio 2019	44

6.2.1. Overview	44
6.2.2. Workflows	44
6.3. ESRI ArcGIS Pro.....	47
6.3.1. Overview	47
6.3.2. Workflows	47
6.4. Unity Real-Time 3D Engine.....	51
6.4.1. Overview	51
6.4.2. Definition and Key Features	52
6.4.3. Unity Components	53
6.4.4. The C# Programming Language and Scripting in Unity	57
6.4.5. The Game Loop	58
6.4.6. Render Pipelines.....	60
6.4.7. Unity goes cross-platform	62
6.5. ESRI ArcGIS Maps SDK for Unity.....	63
6.5.1. Overview	63
6.5.2. Key Features	64
6.5.3. Maps.....	65
6.5.4. Layers	66
6.5.5. Spatial and Data Analysis	66
7. A Digital Twin of Portland Public Transport.....	68
7.1. Architecture Overview	68
7.1.1. RestSharp Library	69
7.1.2. Newtonsoft Json.NET Library	70
7.2. GTFS Realtime Data Loader.....	71
7.3. GTFS Realtime Data Simulator	78
7.4. Unity Project TriMet_Portland_Digital_Twin.....	79
7.4.1. GTFS Data Processor	84
7.4.2. GTFS Stops Data Loader	86
7.4.3. Map Builder	86
7.4.4. Camera Controller	89
7.4.5. Paint Vehicles	92
7.4.6. Display Vehicle Information	93
7.4.7. Animate Headline.....	95
7.4.8. Vehicle Index	96

7.4.9. Escape Key Handler	96
8. Portland Public Transport Traffic Live – Digital Twin Concepts Applied.....	97
8.1. Overview	97
8.2. „Vanilla“ Edition vs. „Trippy“ Edition	97
8.3. Screenshots	98
8.4. Videos.....	99
8.5. Into the Vault.....	100
9. Lookout Mountain... ..	101
Source Code Appendix.....	104
A. GTFS_Data_Loader.....	104
C# file “MainWindow.xaml.cs”	104
XAML file “MainWindow.xaml”	117
B. GTFS_Data_Simulator	120
C# file “GTFS_Data_Simulator.cs”	120
C. Unity Project TriMet_Portland_Digital_Twin.....	123
C# file “GTFS_Data_Processor.cs”	123
C# file “GTFS_Stops_Data_Loader.cs”	133
C# file “Paint_Vehicles_Regular.cs”	135
C# file “Animate_Headline_Regular.cs”	139
C# file “Display_Vehicle_Information.cs”	143
C# file “Vehicle_Index.cs”	146
C# file “Escape_Key_Handler.cs”	146
D. Customized ESRI Code.....	147
C# file “Portland_Map_Builder_Regular.cs”	147
C# file “CustomArcGISCameraControllerComponent.cs”	153
Abbreviations.....	163
References	164
Figures.....	167
Tables.....	170

Abstract

The goal of this project in technical terms is to contribute to enhanced interoperability between the GIS focused software ecosystem of ESRI and real-time 3D engines with Unity as an example of one of the most widely used engines.

The rationale behind this integration effort is to extend the visual language for spatial phenomena by using the sophisticated features of today's real-time 3D engines. These features include realistic materials and lighting, animation, physics, shaders just to name a few. With current generation real-time 3D engines we're pretty close to a photorealistic look and feel of models. But a crucial integration component to connect the up to now distinct universes of GIS and real-time 3D is the extension "ArcGIS Maps SDK for Unity". This add-on to Unity had been published by ESRI a few months ago at the time of writing. The most important feature of this ESRI extension is to add a spatial dimension to Unity's builtin coordinate system that is precisely grounded with longitude and latitude. This is a feature that had been missing in Unity until ESRI released its Unity plugin in June 2022 to fill the gap. Aside from this essential foundation to work with spatial data this extension can be also used to embed maps, layers and other GIS artifacts into a Unity scene.

Integration options between ESRI and Unity have been carefully explored. But with system integration being just a means to an end the overall goal was to build a digital twin of the public transportation network in Portland, Oregon, United States. The vision was about creating a visually appealing, "livin' and breathin'" application that displays the traffic of the Portland public transportation network in almost real time. To get there transit data need to be pulled straight from a GTFS real-time feed provided by Portland's TriMet agency. And to get an almost real-time representation of the TriMet vehicle network GTFS feed data need to be retrieved within short time bursts by a continuously running process. After that feed data has to be stored locally and prepared for consumption by Unity. And once the feed data is shared with Unity I want the data to leverage Unity's advanced visualization features in terms of animation, lighting, moving cameras, materials and, last but not least, C# programming. This integration effort could finally help to have extended design options for visualizing spatial phenomena through leveraging real-time 3D engines. At least this is something I have been hoping for...

Acknowledgements

I would like to thank Karl-Heinz Böhm, my manager at Atos Information Technology, for putting the trust in me that I could pull off something meaningful while exploring the wild and wonderful world of geoscience.

A special “Thank you” also to my supervisor Dr. Christoph Traun who left no question unanswered and gave valuable guidance while I have been working on this project.

Many thanks go out to the entire UNIGIS team from the Paris Lodron University of Salzburg too. Everyone had been there to help throughout the last year when I tried to absorb the sometimes challenging GIS knowledge. I appreciated this a lot.

I’m also grateful to my wife Monika who supported me no matter what while working on this project. In fact she has been the “production manager” behind the scenes who made sure that I kept focus and did not veer off course too much.

Kudos also to the Tri-County Metropolitan Transportation Agency for generously sharing their GTFS and GIS data and for maintaining an excellent developer site. The TriMet data and documentation are top notch and had been essential cornerstones throughout the project.

And last but not least there is Ed Catmull, one of the founders of Pixar, being a source of inspiration when describing his quest to create something beautiful with information technology.

I found his book “Creativity, Inc.” a most interesting read. From his early beginnings at “Industrial Light and Magic” up to the milestone in computer generated movies that is known as “Toy Story” this book helped me to realize that technology and beauty don’t need to be separate entities.

Foreword

I have been fascinated by 3D “things” ever since. The first “3D experience” that made a lasting impression on me happened when I was perhaps 10 years old . As a ten-year-old I came across my uncle’s chemistry book and the only thing I understood was that this book was about somehow dangerous things that make “boom” and “bang”. When I flipped through the pages, I discovered that this book also contained some weird red and blue glasses made of cardboard. And this book had an appendix named “Molekül- und Gitterstrukturen in stereoskopischer Darstellung”, which is in short “Molecules in 3D” for non-German readers. This made no sense to me at all. But that changed when I put on the cardboard 3D glasses. Suddenly I could see molecules (had no idea what molecules were at the time) in their clean and simple beauty and they seemed so real and vivid that I felt I could almost touch them. Looking back, I believe this mesmerizing experience could have been the initial “big bang” that started my passion for all things 3D...

And of course this had been the marvel of anaglyphic 3D glasses...

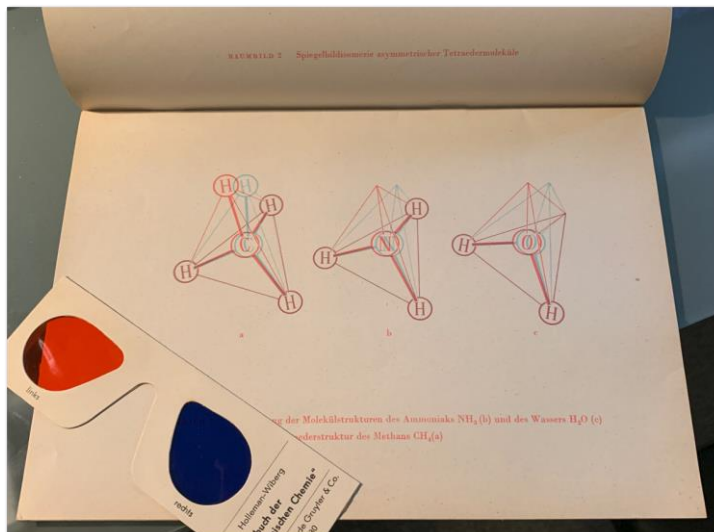


Figure 1: Anaglyphic 3D glasses back in the 1970s... (Source: Own picture)

Fast forward 50 years later... While working in the IT services industry as a consultant for Product Lifecycle Management (PLM) and Computer Aided Design (CAD) I slowly developed a passion for exploring real-time 3D engines. I even became a “Certified Unity Developer” along the way. And it struck me that it would be a major achievement if real-time 3D technology of today’s game engines could be applied to “serious” enterprise use cases in product design, marketing and maintenance just to name the most obvious. And this is what I did over the last years, trying to connect the dots between CAD, PLM and real-time 3D engines...

During this time I had been closely watching what is happening in the 3D realm and I found that ESRI, the company behind GIS powerhouse ArcGIS Pro, had a Unity plugin in development that promised to link GIS data to real-time 3D. That piqued my interest and I wanted to learn much more about GIS technology and how it could be combined with real-time 3D engines...

So the following pages will outline my journey across this Digital Twin project while trying to get to grips with the “ESRI ArcGIS Maps SDK for Unity”, while learning how to benefit from the “General Transit Feed Specification (GTFS)” data feed, while exploring how to get spatial data from ArcGIS Pro

into Unity and many more things that needed to be tackled along the way. And I hope that you will join me in finding some interesting insights during this journey...

1. Introduction

1.1. Motivation

One of the reasons I wanted to work with GIS technology had been that I came across the ESRI plugin "ArcGIS Maps SDK for Unity". I felt electrified when I read up on this. I found this exciting because I had been already working with Unity as a real-time 3D development platform for several years, trying to explore how the advanced graphical capabilities of today's real-time 3D engines could be applied to IT solutions in an enterprise environment.

One of the projects I did during this time was to build a digital twin of a wind turbine in operation. A major challenge of this project was to streamline the wind turbine's CAD data. As CAD 3D data are heavyweights by nature it was hard to imagine that these raw CAD data would perform decently in a real-time 3D environment like Unity. But this challenge could be mastered by using dedicated optimization software. Using the streamlined CAD data Unity was perfectly capable of animating a wind turbine while pulling live sensor data of a wind turbine sitting in a rural area somewhere. One example of sensor data that could be captured was rotation speed of rotor blades. This input had been used to animate the "twin" rotor blades in sync with the rotation of the "real" wind turbine. I vividly remember the excitement when I was able to show this wind turbine "digital twin" linked to its "physical twin" to coworkers and customers.

And I felt a similar kind of excitement when I learned about the ESRI plugin for Unity. I was stunned by the idea that I could do something similar with GIS data. I pondered that the spatial dimension of GIS data could be a key differentiator when used by real-time 3D engines. And I also became aware of a standard called "General Transportation Feed Specification" (GTFS) that can describe public transportation networks like the New York Subway in a spatial manner. While having routes and stops of a particular metro is already key spatial information I was even more interested in the GTFS Realtime format that is describing vehicle positions in almost real-time. I thought what would be a better match than having real-time location data in a real-time 3D engine. I began to deep dive into the GTFS Realtime specification and pretty soon I imagined a 3D scene with moving, glowing dots with each dot representing a vehicle location, continuously updated by the GTFS Realtime feed... and this still blurry image in my mind led to the idea of a "digital twin" of a public transport network based on live data.

I have to admit that this project has some traces of "l'art pour l'art" in it because watching the moving dots on the screen is already a beautiful and satisfying experience – at least for me. But I hope this project also holds enough engineering substance to make it interesting when looking at it from an IT perspective.

After thinking about what I wanted to achieve with this project I came up with a first draft of an architecture. Although it was a rather rough sketch at the time it already featured many elements of what I had been aiming for. This sketch somehow marked the starting point of this adventure in 3D. And it felt like an adventure because I pondered I would be facing many unknowns especially with the ArcGIS Maps SDK for Unity being just out of beta for a month by the time I started working on it. But finally, after a few bumps in the road, everything came together just fine. And as a tiny glimpse into the final result please take a look at a random screenshot below.

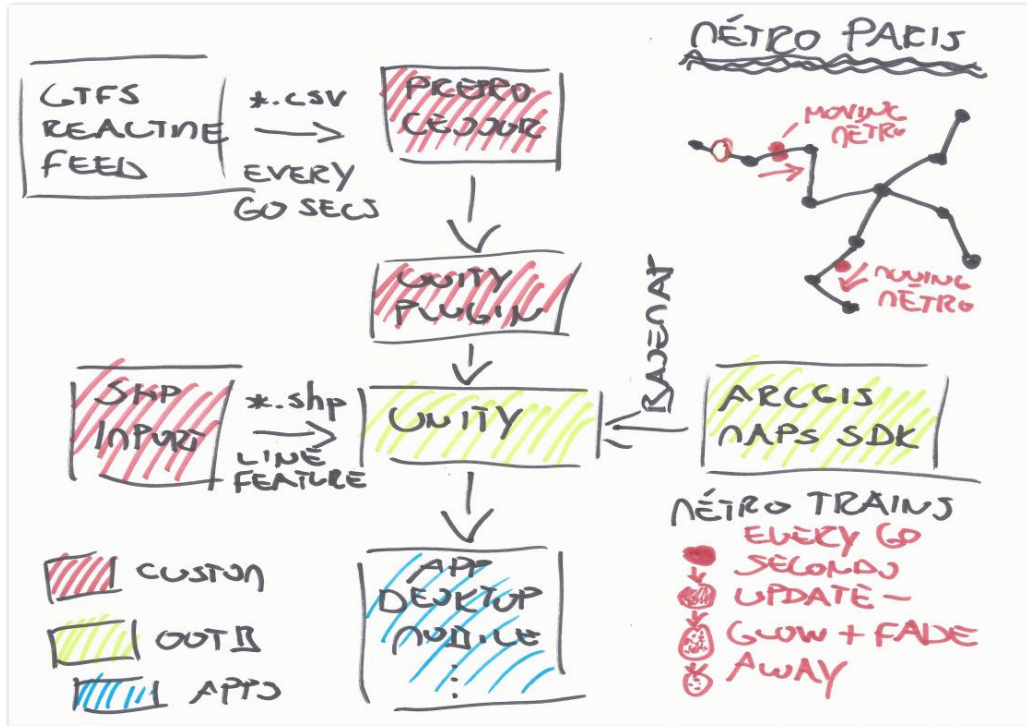


Figure 2: How it all began – first rough sketch on architecture ideas (Source: Own picture)

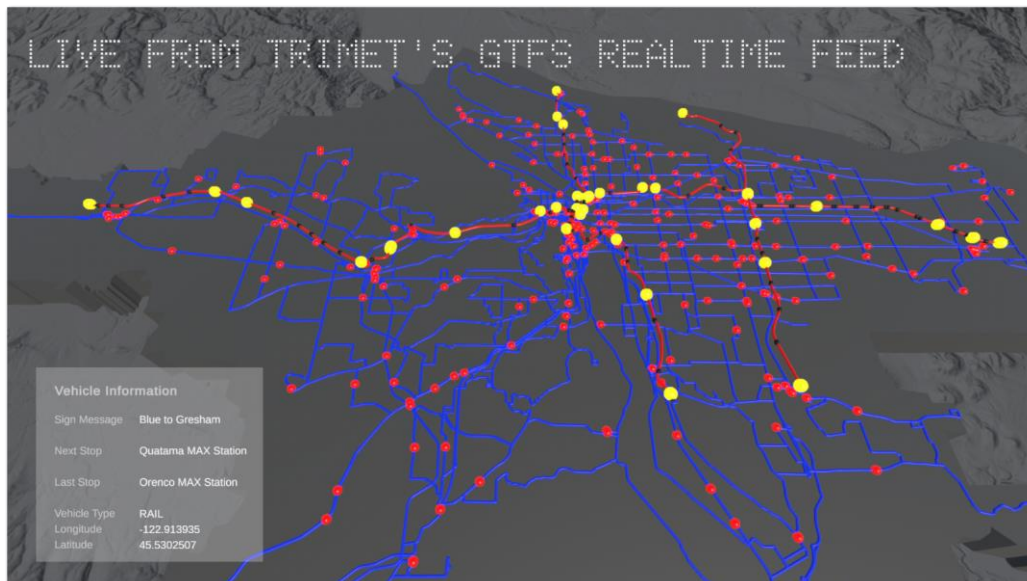


Figure 3: And how it turned out – random screenshot of final Unity application (Source: Own picture)

1.2. Objectives

In this chapter I would like to set up some project objectives to be tackled while being on this technical journey. So the following pages will have a more detailed breakdown on this.

1.2.1. GTFS Realtime Data Feed

The core idea was to visualize the movements of network vehicles across a map. This would be possible only if I had access to network vehicles' location data. And the best way to get location data of vehicles seemed to be using a GTFS Realtime feed. According to the GTFS Realtime documentation this feed would carry precise location data in terms of longitude and latitude for vehicles. So that meant it would be key to find a public transportation network that would share its vehicle location data as part of a GTFS Realtime feed.

Objective → Understand syntax and semantics of GTFS

Objective → Find a GTFS Realtime feed that contains location data

1.2.2. GTFS Realtime Data Loader

It was obvious that once I would identify a GTFS Realtime feed I would need to continuously query GTFS Realtime data and pull the vehicle location payload from the feed. The resulting data should be easily consumable by a real-time 3D engine. It made sense to me to implement this building block as a standalone application that operates independently from the real-time 3D engine. Although it would have been feasible to make it a part of the Unity application I was afraid of doing so would eat too much from the performance needed for rendering frames to the screen and eventually would slow down the final Unity application.

I also pondered that building a plain command line application would not be convenient enough to work with. I would rather prefer to have an easy to use GUI that is making it easier to interact with the GTFS Realtime feed by the usual controls the Windows platform has to offer. And slowly the notion of a "GTFS Cockpit" developed...

Objective → Develop a standalone application that takes care of continuously querying GTFS Realtime data

Objective → Add a convenient user interface to the application

1.2.3. ArcGIS Maps SDK for Unity

This Unity plugin published by ESRI would serve as an essential cornerstone of the project because it would close the spatial gap between GIS data and Unity. By default Unity does not know anything about coordinate systems in terms of longitude and latitude. But ESRI and Unity teamed up to close this gap. In a joint effort by both companies they made it possible to use projected and geographic coordinate systems in Unity and locate objects precisely.

Objective → Get to grips with SDK installation and other preconditions like getting an ESRI developer key

Objective → Understand how to work with ArcGIS Maps SDK for Unity

Objective → Push location data from GTFS Realtime feed into ArcGIS Maps SDK for Unity

1.2.4. ArcGIS Pro

I also expected that certain spatial data like routes or stops of a public transportation network would be provided as shape files. At the same time I already knew from first research that ArcGIS Maps SDK for Unity would support 3D scene layer packages or *.slpk files. This raised the question of how to transform point, line and polygon features into a 3D scene layer package.

*Objective → Set up a workflow in ArcGIS Pro to create *.slpk files from shape files*

1.2.5. Unity

Unity as the visualization engine would need access to all vehicle location data provided by the “GTFS Data Loader” to display vehicle locations. So one of the tasks that needed to be tackled was to enable Unity to continuously consume location data from the GTFS source. From my experience with Unity I thought that this task had to be a separate thread in Unity otherwise the burden of processing GTFS data could hurt the frame rate when visualizing vehicle movements at runtime.

Objective → Build an independent Unity thread to consume vehicle locations

Of course no representation of spatial phenomena would be complete without maps and layers. Using Unity and the ArcGIS Maps SDK for Unity I imagined to have a kind of “canvas” that would have several layers like routes and stops painted on it. Vehicle movements would happen against this backdrop to provide spatial context.

Objective → Build a local scene in Unity with several layers that serve as a spatial “canvas”

I also saw an opportunity to use some of the rich visualization features of Unity like emissive materials and animations to enhance the visual appeal of the project and to make it interesting to look at. When I considered design options I eventually went for a non-realistic, stylized look. I pondered that a clean look could help avoiding information overload when several hundred vehicles would move on the screen at the same time.

Objective → Build out a clean scene and set up emissive materials and animations for vehicles in Unity

Another thing I considered essential was some kind of scene title in Unity to provide information on what the scene is all about. In broader terms this meant thinking about

the user interfacing part of the scene. I envisioned using the Unity package TextMeshPro for this task since this is the de facto standard with Unity when working with text information.

Objective → Set up a scene title in Unity using TextMeshPro package

Finally I deemed it necessary to have some more bits of information about the moving vehicles on the screen. The idea was to let the mouse pointer hover over a particular vehicle and have an info panel updated with basic information about it i.e. its sign message, the type of vehicle, its last and next stop and last but not least longitude and latitude of the vehicle position.

Objective → Set up a vehicle information panel in scene using TextMeshPro package

Since it turned out that the information provided by the GTFS Realtime feed contains only numerical ids for describing stops it became necessary to leverage additional GTFS Schedule data to have a plain text description of stops available. In fact there is a mandatory file as part of the GTFS Schedule specification that is called `stops.txt`. This file describes all stops of a network with both their numerical id and textual description. So I anticipated having a mapping requirement to be able to display last and next stops as plain text at the vehicle info panel.

Objective → Set up a mapping between stop id and stop name

Finally all of these building blocks needed to work with each other seamlessly within a unified(!) Unity project. I anticipated that there would be some “glue” code programming needed to let each component talk to others as needed.

Objective → Integrate all project components with each other

1.3. Mapping “Digital Twin Land”

With all major objectives now settled in the previous section it is time to embark on the journey. And since every journey needs some careful preparation there is nothing better than having a map even if it’s very basic. But it’s also hard to miss that my map is lacking any straight lines because a lot of detours happened along the way. But as a matter of fact I finally made it to the finish line...

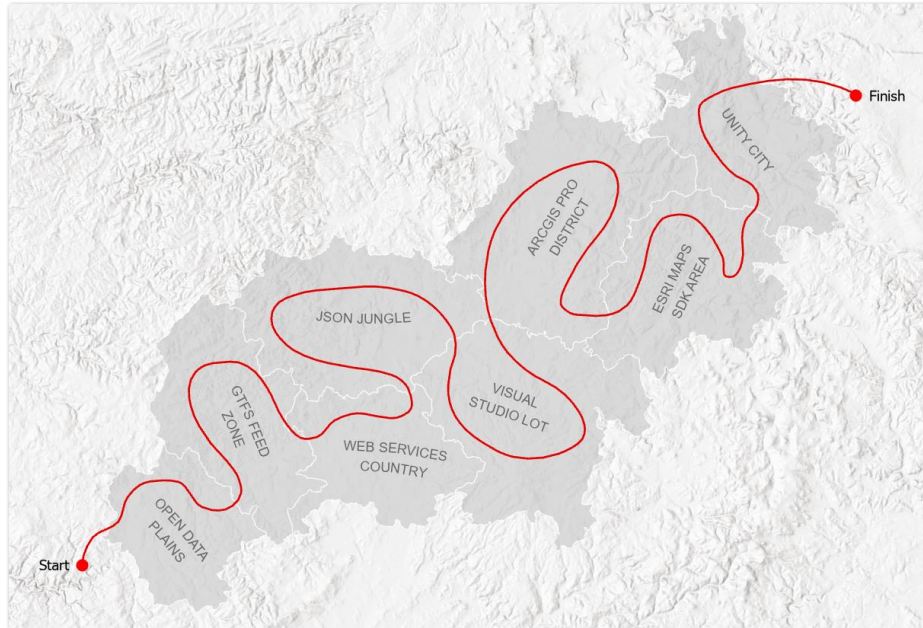


Figure 4: My travel route across “Digital Twin Land” (Source: Own picture)

<OFF THE RECORD>

As seen above I found it somehow nice to have a map like overview on this technical adventure. But once this map had been created I wanted to have even more fun. Being in a playful mood I made up a fictional movie title like “Indiana Jones and the Secret of the Digital Twins” (Thank you so much MS Paint/MS Powerpoint!). And be aware Hollywood, here we come :-). And no fedora hats have been worn while working on this project ;-)

</OFF THE RECORD>



Figure 5: Having fun with Indiana Jones typefaces – with tongue in cheek (Source: Own picture)

1.4. Chapter Overview

This overview is meant to provide a brief summary of all chapters.

1.4.1. Digital Twins

In this chapter I'd like to discuss what the notion of a Digital Twin is all about. Starting with a quick look at the early beginnings of the Digital Twin concept I will move on to possible definitions of a Digital Twin. That means trying to sketch what a definition of a Digital Twin could look like and also finding some general traits of a Digital Twin. I will also try to get the question answered for what reasons it could be beneficial to embrace Digital Twins from an enterprise perspective. Finally this section concludes with some hand-picked examples of Digital Twins.

1.4.2. General Transit Feed Specification (GTFS)

This part will cover the GTFS standard for sharing transit information including stops, routes, trips, fares and the like. A quick history of GTFS will be trailed by some more background on GTFS Schedule and GTFS Realtime. Finally the subset of the specification that is relevant to this project is considered in more detail.

1.4.3. TriMet Open Data

TriMet, the public transportation agency of Portland, is sharing a wealth of open data. Its open data consist of various files and feeds based on the GTFS standard. TriMet also offers transit data like routes and stops in a GIS-friendly shapefile format. So this section sets out to give an overview of all the resources provided by TriMet.

1.4.4. Hardware Environment

This part contains a description of hardware that has been used. Given how close hardware and software are linked to each other this chapter may provide some insights on hardware preconditions for working with real-time 3D technology in general. Not very surprisingly it turned out that powerful hardware is key for getting decent results because poor GPU performance can hurt the project experience altogether.

1.4.5. Software Environment

With the description of the underlying hardware settled by the previous chapter software is another project layer to be looked into. So this part will give an overview of all the software pieces used throughout the project. Software specific workflows will be also covered when they contributed to building this digital twin.

1.4.6. A Digital Twin of Portland Public Transport

In this chapter various software artifacts that sum up to a Digital Twin of the Portland public transport network get discussed. Starting with the GTFS data source up to the final

real-time visualization and everything in between, all software building blocks are on display to provide a comprehensive picture of the Digital Twin architecture.

1.4.7. Portland Public Transport Traffic Live – Digital Twin Concepts Applied

This part sets out to highlight the visuals of the Portland Digital Twin. Screenshots and a teaser video are meant to bring the application to life. The video will focus on interacting with the Digital Twin and how live information from the GTFS feed has been embedded into the application to provide additional bits of information.

1.4.8. Lookout Mountain...

In this section I'm looking forward to take a view from the top. This part is about looking back, about looking in all directions but also about looking ahead. So this chapter is meant as a kind of coda that tries to reflect on this endeavour, to think about the lessons learned and to ask "Where do we go from here?". In a way I could have worked on this project forever because the ideas continued to keep coming. But as always time constraints have been finally responsible for limiting the number of features to build...

2. Digital Twins

This chapter is meant to take a closer look at Digital Twins. After looking into the beginnings of Digital Twins I will consider possible definitions of Digital Twins and what benefits Digital Twins could bring to the table. Eventually this section wraps up with some interesting examples of Digital Twins.

But maybe it's interesting too to read about my personal history when it comes to Digital Twins. When I had been working in the Product Lifecycle Management (PLM) domain the term Digital Twin was (and still is) a hot topic in the PLM community. There was a stream of conversations over the last years that looked at PLM, Industry 4.0 and Digital Twins as areas with a lot of overlap. And when I tried to come up with a definition of PLM some years ago to help me with explaining the notion of PLM to "newcomers" I put it like this:

PLM enables and supports the collaborative creation, management and use of product information across the enterprise from product idea to end of life. It's integrating people, processes, business systems and information to shape a product information backbone.

Maybe it's helpful to add some more background here: PLM systems are often used to manage "products" that are very complex. Things like cars, ships and aircraft are good examples for products with overwhelming complexity. It is almost impossible nowadays to manage product development without any smart software support because products like this literally consist of millions of parts that in one way or another are depending on each other.

The interesting thing here is that many of the features that can be found in PLM can be also found in conversations around the buzzword Industry 4.0: Connecting people, processes, machines, information, IT... the discussion threads are very similar. And in fact the evolution of technologies like Internet of Things, 5G always-on connectivity, Cloud, Web Services helped to pave the way for an ever more connected manufacturing industry.

This interconnected fabric out of PLM and Industry 4.0 might have been the fertile ground, at least in the manufacturing industry, on which ideas around Digital Twins started to flourish. And when thinking about PLM which is often tightly integrated with Computer Aided Design (CAD) systems it almost comes naturally to beef up a CAD model of a car so that it resembles the physical product as much as possible. Almost needless to say that 3D technology is a key foundation for any Digital Twin because its "physical twin" is in three dimensions after all. If the ambition is to get as close to the "physical twin" as it gets then 3D technology is a key enabler.

But for now I'll take off my former PLM glasses and take a look at the beginnings of the Digital Twin era along with one of the first attempts on a Digital Twin definition.

2.1. The Early Years of Digital Twins

There seems to be a consensus that American space agency NASA has been among the first to coin the term "Digital Twin". And this makes perfect sense because as a space agency they are constantly looking for ways to make their vehicles more safe, more reliable, more predictable and possibly life-saving in the end. In retrospect it feels almost natural that the concept of a digital twin took shape at NASA first because of the need to test and simulate vehicles as close

as possible to the “physical twin”. Since any malfunction with NASA vehicles could possibly lead to a disastrous outcome NASA went to great lengths to minimize any risk of failure. When thinking about it it feels almost like a logical step to not test/simulate subsystems only but the entire vehicle. And with mimicking the behaviour of a whole vehicle we’re already pretty close to a blueprint for a Digital Twin.

One source that seems to confirm that NASA has been a pioneer when it comes to Digital Twins is a research article titled “A Review of the Roles of Digital Twin in CPS-based Production Systems” (PROC 2017A). According to this article...

The first definition of the DT was forged by the NASA as “an integrated multi-physics, multi-scale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin. It is ultra-realistic and may consider one or more important and interdependent vehicle systems”: this definition first appeared in the draft and after in the final release of the NASA Modeling, Simulation, Information Technology & Processing Roadmap in 2010 [16,17]. From that moment on, aerospace researchers started referring to the said NASA roadmap as the seminal work to define the DT (as an example [18]).

When looking at this definition today it already contains many terms that can be overheard in current discussions on Digital Twins. Terms like “simulation”, “models”, “sensors”, “ultra-realistic”, “history” would easily fit into today’s picture of Digital Twins.

With the “inventor” of the Digital Twin concept probably identified now and with a first attempt on a comprehensive definition of a Digital Twin in place it might be interesting to see how Digital Twin definitions evolved over the years.

2.2. Digital Twin Definitions

When browsing the research paper mentioned earlier (PROC 2017A) I noticed that it contains quite a few definitions on Digital Twins. It’s no surprise that some of the definitions found are closely coupled to aircraft industry. And it’s not difficult to understand that a concept beneficial to the reliability of spacecraft can be also applied to aircraft. So it looks like aircraft industry had been at the forefront of Digital Twin adoption. There is one compact definition that I found especially appealing even if it’s tightly coupled with aircraft.

Ultra-realistic, cradle-to-grave computer model of an aircraft structure that is used to assess the aircraft’s ability to meet mission requirements.

I like this definition because it highlights “ultra-realism” and there is also this sense of “lifecycle” again because “cradle-to-grave” is nothing else than a product’s lifecycle from idea to end of life. And I also feel that tracking an aircraft across its entire lifecycle is something that can create valuable data. As an example data might provide insights on Mean Time Between Failures (MTBF) with replaceable parts thus enabling “predictive maintenance”, another buzzword from the Industry 4.0 crowd. The idea here is to replace a part before it is expected to fail and to avoid downtimes by doing so. But this is just an example on how to benefit from continuous data capture. In broader terms this is about recognizing trends along a timeline that could help to identify a design’s weak spots because they fail over and over again.

But today the term Digital Twin is not limited to spacecraft and aircraft anymore. One random flashlight that shows how much the notion of a Digital Twin has entered the mainstream is the latest “ESRI Konferenz 2022” that took place in October 2022. This is a conference hosted by ESRI for German speaking audiences to showcase the latest advancements in GIS technology. It featured several tracks and all of them built upon the idea of Digital Twins. In fact these tracks had been called “Governmental Twin”, “Urban Twin”, “Infrastructure Twin”, “Business Twin” and “Environmental Twin”. That might lead to the conclusion that Digital Twins seem to be something important to be dealt with in the GIS domain. At least according to ESRI...

I admit that it is quite a pivot from rockets to geosciences but this conference snapshot might indicate that the idea of Digital Twins is about to become a cross-sectional technology which is leveraged by a range of varied industries. This is also reflected in an article named “Digital Twin: Enabling Technologies, Challenges and Open Research” (IEEE 2020A). This article shows that most Digital Twin research in the academic community is happening in the “manufacturing”, “healthcare” and “smart cities” areas.

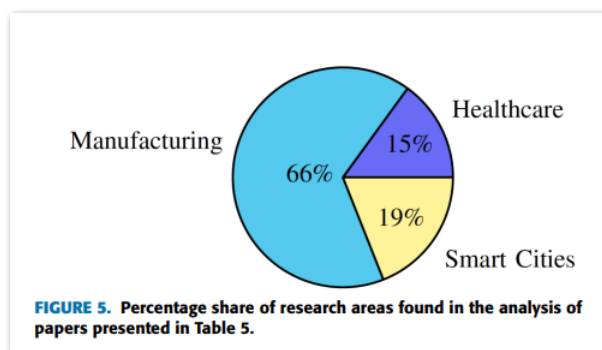


Figure 6: Amount of Digital Twin research across different industries (Source: IEEE)

Although there is an overwhelming percentage of papers that deal with Digital Twins in manufacturing it looks like “smart cities” as a genuine GIS topic are catching up. And I don’t find it surprising that “smart cities” are behind because it’s one thing to have a digital twin of a car but to have a digital twin of an entire city is a completely different story because of its much grander scale and its much grander technical challenges.

After reviewing different definitions across time and industries I was also curious what Unity as a major promoter of Digital Twins has to say on this. After all this project is largely based on Unity’s real-time 3D engine. In fact Unity dedicated a complete e-book to this topic. It is called “The what, why and how of digital twins” and can be downloaded from Unity’s resources site (UNITY 2022D). And this is what Unity’s definition of a Digital Twin looks like:

A digital twin is a dynamic virtual copy of a physical asset, process, system or environment that looks like and behaves identically to its real-world counterpart. A digital twin ingests data and replicates processes so you can predict possible performance outcomes and issues that the real-world product might undergo.

It’s somewhat surprising that the aspect of “ultra-realism” is missing from this definition because this is one of Unity’s defining features to have almost photorealistic visualization features. The same is true for the interactive and immersive dimension. Maybe Unity’s definition lacks these features because Unity takes them simply for granted. But at the same

time I appreciate that this definition has a much broader perspective on Digital Twins. It is much broader because it does not limit them to asset-like products (car, ship, aircraft) but also includes processes, systems and environments. Especially the notion of environments has a lot of overlap with GIS technology.

So after getting quite a bit of inspiration from these different approaches to Digital Twin definitions it is time to come up with my own take on a definition.

For sure a high degree of realism is required for any model to qualify as a Digital Twin. Real-time 3D engines can provide the foundational technology needed to achieve this kind of realism. I also like the wide scope of Unity's proposal when it comes to areas that Digital Twins concepts can be applied to. And I also feel that the "alive" notion is a crucial part of any Digital Twin. This translates to being connected 24/7 to gather live data from a "physical twin". And it doesn't matter if this is done by IoT or 5G technology or by any other means. The important thing here is that there is a "lifeline" between digital and physical twin. A lifeline that connects the twins almost like a "data umbilical cord". Continuous data updates from the physical twin can make Digital Twins more lifelike in their behaviour. And to unleash the full value of this live data I would also recommend to have a data history to be able to extract insights from this. Insights that could possibly predict the future when it comes to maintenance as an example. With all these ideas in mind I'll give it a try with a definition of a Digital Twin:

A Digital Twin is a highly realistic model of a physical asset, process, system or environment. Its foundational real-time 3D technology enables an almost photorealistic appearance while being interactive and immersive at the same time. It can easily cope with real-time data because of its rich integration capabilities to interface with any data source. Continuous data exchange in real-time informs the notion of a data lifeline between digital and physical twin which in turn enables a Digital Twin to behave in an ever more realistic and timely behaviour. Finally a Digital Twin unleashes its potential even more if it has a data memory to predict the data future.

With that preliminary definition now in place I feel confident to talk some more about possible benefits of Digital Twins...

2.3. Benefits of Digital Twins

Since the manufacturing industry has been at the forefront with Digital Twin adoption it might be interesting to look at their reasons for embracing Digital Twins. And I'm convinced that there are quite a few domains where Digital Twins might help to get better results fast. But because this is a very general statement it may help to bolster this with some examples.

2.3.1. Design

For now I will stick with a car. Obviously the design of a car is a lengthy, time-consuming process with many, many people involved. In the early stages of design a Digital Twin could help to foster collaboration because there is no need to gather in person at a design draft. Of course it helps to meet in person to evaluate a design but this is no hard requirement any more. It could be simply done with having a Digital Twin of the design that is accessible by the whole team. The realism of a Digital Twin would help to immerse

into the design. Since working on a car design is a very collaborative effort with people from many disciplines the Digital Twin concept may help a team to align on design decisions. Being on the same page at any point in time would enable the team to make informed design choices. And this approach might also help to catch any design issues earlier. To put it like that: It is not taken for granted that a car engine does fit under the hood exactly. This is something that needs to be checked carefully upfront (Perhaps someone from the automotive industry would be bursting with laughter when reading this but it's easy to see what I mean). And if it comes to car batteries these kind of issues are not that far off. But anyway being able to play with what/if scenarios based on a Digital Twin is something that could contribute to getting better, more robust designs faster.

2.3.2. Maintenance

I already touched upon the example of a wind turbine Digital Twin when I talked about my experience in this area in section *1.1 Motivation*. There are quite some reasons why having a Digital Twin of a wind turbine could help with maintenance. In many cases wind turbines are located in a sparsely populated area or even off shore. That means it is key to know as much as possible about its operational status when being at a distant operation center. Since wind turbines are packed with sensors that capture temperature, rotor speed and a lot of other metrics it would be feasible to let a Digital Twin leverage this data. By mirroring the rotor speed for example a Digital Twin could resemble the physical wind turbine in operation. When the realism that comes with Digital Twins is combined with the rich data captured by the turbine sensors this gets pretty close to a lifelike representation.

As outlined before this rich data might enable an operator to recognize trends of key metrics, to predict maintenance and also to prepare for doing maintenance in the field. It's not hard to imagine that wind turbine maintenance on site needs to be carefully planned. Of course it's necessary to have the route to the site planned, sometimes a crane truck will be needed which has its own planning challenges, spare parts need to be identified and prepared and last but not least there might be a requirement for special tooling that the maintenance team has to carry. The need to consider all these requirements underpins the logistic challenges that are imposed on the maintenance staff.

In front of this background it's evident that the more an operator knows about the site status in advance the faster and better any maintenance measures can be prepared. And chances are that by having insights on sensor data history a wind turbine could be prevented from a surprising failure that would cause emergency maintenance.

2.3.3. Training

Considering the advances that have been made with VR (Virtual Reality) and AR (Augmented Reality) there are also reasons to leverage Digital Twin technology in training scenarios. Since both VR and AR are powered by real-time technology it comes almost natural to use a model that mimicks its physical counterpart for training purposes. Once again aircraft may be a convincing example of saving time and money by using a Digital

Twin. When I think of pilot training to prepare for flying a commercial aircraft it is just too expensive to do the complete training on the real thing. And it might be also quite dangerous to do that. I admit that the border between Digital Twin and simulation gets blurry here but there is a strong case for doing expensive or dangerous training with a Digital Twin.

Even when a failure would not lead to a disastrous outcome as in pilot training I can imagine a lot of use cases in manufacturing where AR technology based on Digital Twins could speed up training and flatten the learning curve. When assuming there is an expensive piece of factory equipment that a operator needs to be trained on this could be done in VR to get the operator up to speed on basic workflows (“If LED X flashes red then part Y needs to be checked”). The Digital Twin environment in VR could also provide helpful hints what needs to be done to help with mastering the machinery.

As soon as basic workflows have been learned an operator could start working on the physical equipment in an “augmented” environment. It is “augmented” because it leverages AR technology based on a Digital Twin. The AR device might give guidance in a tooltip like fashion when working the machinery but the device could also serve as a guide along the lines of “push button A, then adjust lever B”. Every time if expensive machinery with a steep learning curve needs to be handled or if working in a hazardous environment then Digital Twins combined with VR/AR technology could make a workplace more safe and more efficient.

2.3.4. Simulation

For sure there is some overlap between training and simulation domains as highlighted by the previous example of a pilot training. It’s a “conditio sine qua non” for a pilot training that the Digital Twin based training environment has to simulate the behaviour of real aircraft as much as possible. But there might be also other use cases where lifelike simulation is a requirement in its own right.

When thinking about a Digital Twin of an entire city it might be insightful to play with different what/if scenarios to explore possible outcomes. How urban planning might affect traffic flows across the city, how traffic flow could be optimized to avoid accidents and how coverage by public transport could be improved, all these areas would benefit from a Digital Twin with data memory and simulation capabilities. Simulation features might not be an integral part of a Digital Twin architecture but pretty sure a Digital Twin can be considered a foundation that simulation features can be built upon because it is already modeled after reality. So the basics are already there for enriching the twin with simulation features.

2.4. Digital Twin Examples

To add more color to the picture of Digital Twins I handpicked some examples how this concept has been applied across different industries.

2.4.1. Vancouver International Airport

One of the most interesting examples I found was a Digital Twin of the entire Vancouver airport in Canada that pushes the limits of what can be done in the Digital Twin realm. Rich live data, photorealism, maintenance, it's all there. Some of the highlights of this example that folds GIS, BIM, IoT and real-time 3D into one unified application can be found in the linked article (BITC 2022A).



Figure 7: Digital Twin of Vancouver International Airport (Source: Vancouver International Airport)

2.4.2. Realtime Bitcoin Globe

Another example of a Digital Twin that I found appealing because of its lightning fast 3D visualization, because of its nice visuals and because of its worldwide real-time monitoring of Bitcoin mining activities is the “Realtime Bitcoin Globe”. It is based on the “WebGL Globe” project of the Google Data Arts Team and it shows transactions happening on the Bitcoin blockchain in real-time. It also features the locations where blocks have been mined and the amount of mining activity across the globe. This example is also a nice example for a process based Digital Twin (BITC 2022A).



Figure 8: Digital Twin of Bitcoin mining activities (Source: Experiments with Google)

2.4.3. Wind Turbine Immersive Digital Twin

In earlier sections I talked a bit about a wind turbine Digital Twin that I had been working on some time ago. So with all due modesty I'd like to present two snapshots from this Digital Twin. CAD data of the wind turbine had been repurposed as building blocks for visualization. To be honest this Digital Twin is not geolocated in any way but the sensor data had been pulled in real-time from its physical counterpart by leveraging web services. Even a kind of continuously updated "sensor data ticker" had been embedded in this application, similar to a stock ticker.



Figure 9: Digital Twin of Wind Turbine (Source: Own picture)

3. General Transit Feed Specification (GTFS)

3.1. Overview

Although I will be using only selected features of GTFS during this project it might be helpful to take a closer look at some of the GTFS basics. When I began learning about GTFS I came across an excellent online training on GTFS hosted by the World Bank Group (WORLDBANK 2022A). This has been a major source of my current GTFS knowledge. This training also gives some background on GTFS history and how GTFS came to be. So the following section is an excerpt from the introductory module of this training that outlines the early beginnings of GTFS...

Where does the story of GTFS begin? First stop, USA! In 2002, Bibiana McHugh, then IT manager of geographic information systems for Portland's transit authority, Trimet, asked her team, "Why can't getting directions on the Internet be as simple as getting driving directions? This simple question led to an experiment, using Portland's transit service database to create an online trip planning tool for transit services. The proof-of-concept worked! And long story short, Portland's transit data structure was quickly adopted by Google as the basis for a new standard for transit service data -- the General Transit Feed Specification, GTFS for short.

What started as a simple question developed into one of the important transit data specifications over the years. As a "lingua franca" for transit related data like routes, schedules, stops and so on it built a common ground for describing transit data and made it a lot easier to exchange transit data in a standardized fashion. The support by IT giant Google also made sure that the specification became widely adopted by both transit agencies and 3rd party developers. Up to this day Google maintains a vast array of resources to support developers with guides, code samples and community support (GOOGLE 2022A). Due to its open source nature and extensibility there is also a vivid GTFS developer community out there.

3.2. GTFS Schedule

The core of the GTFS Schedule specification or sometimes simply called GTFS consists of a collection of comma separated value (CSV) files that contain various transit related data. Even if they are CSV files it is common practice in the GTFS community to save them with a *.txt file extension. Some of these files are mandatory while others are optional. Since all data are based on the familiar CSV format it is pretty easy to process GTFS files. A typical example of a GTFS file, in this case the routes.txt file is shown below. All GTFS files have in common that the first line describes the meaning of the different fields while the next lines contain the actual data. In this example this is route_id, along with route names in a short and long version and other route specific information.

```
route_id,agency_id,route_short_name,route_long_name,route_type,route_url,route_color,route_text_color,route_sort_order
1,TRIMET,1,Vermont,3,https://trimet.org/schedules/r001.htm,,,400
2,TRIMET,2,Division,3,https://trimet.org/schedules/r002.htm,,,500
2a,TRIMET,FX2,Division,3,https://trimet.org/schedules/r002a.htm,,,350
4,TRIMET,4,Fessenden,3,https://trimet.org/schedules/r004.htm,,,600
6,TRIMET,6,Martin Luther King Jr Blvd,3,https://trimet.org/schedules/r006.htm,,,800
```

Figure 10: Sample GTFS file routes.txt

It's worth noting that all GTFS files are interlinked with each other by using a shared field. This type of file organization is very similar to the organization of tables within a relational database. To stick with the `routes.txt` example this file is connected to two other GTFS files by having a common field. Using the example of `routes.txt` these two GTFS files are `agency.txt` and `trips.txt`. The file name `agency.txt` already implies that it contains additional information about the transportation agency like its name and contact details. The `trips.txt` file is used for describing all the trips that are scheduled on a given route. The following picture tries to highlight the relationship between these three files and how they are interwoven with each other.

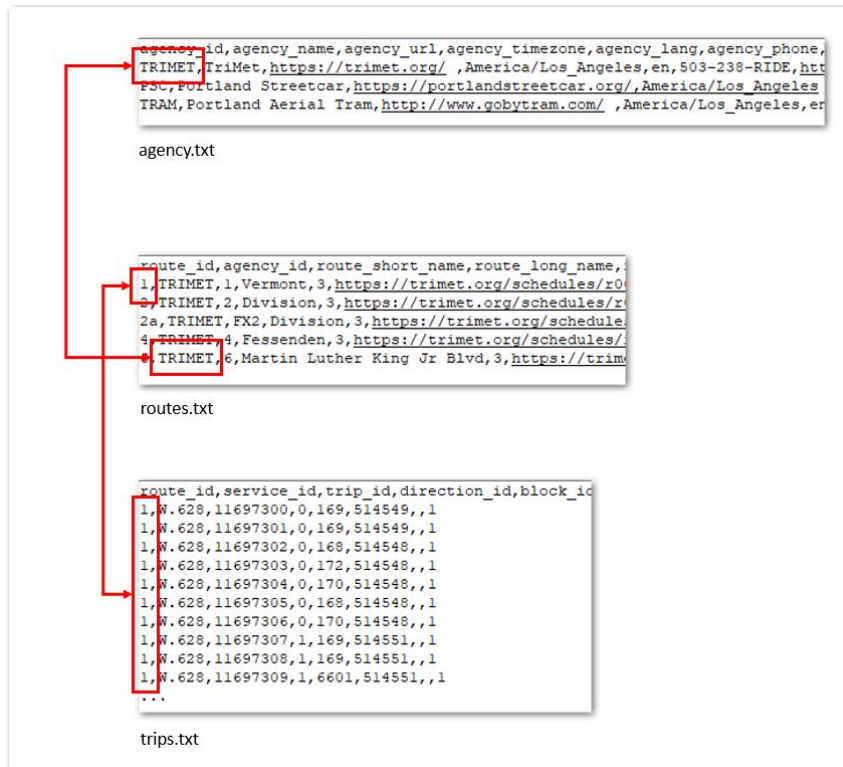


Figure 11: Exemplary GTFS file relationship using a shared field

This type of relationship is true for all files that are bundled in the GTFS Schedule standard, it is a basic principle of how GTFS files are organized and how they refer to other GTFS files that contain supplementary information.

3.2.1. Mandatory GTFS Files

Now that some basic traits of GTFS files have been explored it's time to go into more detail. The GTFS Schedule specification distinguishes between required and optional files. At least all required files have to be present to qualify a feed as complying with the GTFS specification. In this section the required files defined by the GTFS standard will be explained some more. Those mandatory files help to answer basic questions about a public transport network. Some of the questions are along the lines of "Who is the operator of this transit service?", "What are the routes and what is their destination?", "What are the stops on a given route and how are they named?" and also "What are the

days and hours of operation?”. From a GIS perspective the file `stops.txt` is especially interesting because it contains latitude and longitude data for each stop thus making it a great data source for visualizing point features. The following table summarizes the mandatory GTFS files and their meaning.

GTFS Schedule Mandatory Files	
<code>agency.txt</code>	Transit agency details
<code>routes.txt</code>	Route names
<code>trips.txt</code>	Inbound and outbound trips that describe a route
<code>calendar.txt</code>	Days and hours of operation
<code>stops.txt</code>	Stop names and where they are located
<code>stop_times.txt</code>	Ordered list of stops for a given trip including arrivals and departures

Table 1: GTFS Schedule Mandatory Files

3.2.2. Optional GTFS Files

Even if the sum of mandatory files already describes a fully fledged GTFS feed there is a set of optional files that can further enhance the information content of a feed. Again the following table provides an overview of supplementary GTFS files and their meaning. But as pointed out at the beginning of this section only a limited set of GTFS features will be used throughout this project. I will focus on the GTFS features required for mirroring movements of network vehicles in real-time so any description of optional GTFS features is just added for the sake of completeness.

GTFS Schedule Optional Files	
<code>feed_info.txt</code>	Feed metadata like publisher, version, expiration date, etc.
<code>fare_attributes.txt</code>	Ticket prices, currency and similar
<code>fare_rules.txt</code>	Rules on how to apply fares to a specific route
<code>frequencies.txt</code>	Trips that happen in regular time intervals
<code>shapes.txt</code>	Points describing the path associated with a route
<code>calendar_dates.txt</code>	Indicates limited services like public holidays
<code>transfers.txt</code>	Custom rules for selected transfers

Table 2: GTFS Schedule Optional Files

3.3. GTFS Realtime

As technology evolved over the years and GPS became more affordable this technology finally went into mainstream. So it became feasible to embed additional data into a GTFS feed that beefed up its contents even more. Most notably here is the availability of realtime vehicle position data at any point in time of operation across the entire vehicle fleet. Other extensions that are also part of GTFS Realtime are trip updates and service alerts although these features are out of scope for this project.

Perhaps it's best to let Google as one of the main contributors to the GTFS Realtime specification share its take on the definition of the format and let them share what they had in mind when creating the GTFS Realtime specification (GOOGLE 2022B).

GTFS Realtime is a feed specification that allows public transportation agencies to provide realtime updates about their fleet to application developers. It is an extension to GTFS (General Transit Feed Specification), an open data format for public transportation schedules and associated geographic information. GTFS Realtime was designed around ease of implementation, good GTFS interoperability and a focus on passenger information.

According to the revision history of the GTFS Realtime specification the initial version had been released in 2011, about 6 years after the specification of the GTFS Schedule standard (GOOGLE 2022c). This is worth noting because 2011 had been quite different from today in terms of capable internet connections. Back in the day it couldn't be taken for granted that there is rich bandwidth everywhere. And this seems to have influenced a design decision to rely on the binary i.e. compact Protocol Buffer format as a data carrier instead of – as an example – the more verbose and bandwidth consuming XML format that was already well established at the time.

3.3.1. Overview

In this section some of the major components of the GTFS Realtime extension will be introduced. To set GTFS Realtime apart from GTFS Schedule it may be helpful to take a look at the `routes.txt` file from the GTFS Schedule specification. The `routes.txt` file describes all routes that a public transportation network is composed of. This data is expected to be stable over a longer period of time. Although it may change when new routes are added and others are removed the general assumption is that changes of `routes.txt` only happen every six to twelve months when a new schedule is released.

Compared to these more static data GTFS Realtime covers transit related data that is expected to change very frequently. Most importantly for this project GTFS Realtime covers all the Vehicle Positions in a network which by nature are expected to change continuously due to vehicle movements (“The vehicle x is at position y at at time z”). Other parts of GTFS Realtime include the Trip Updates feed that is meant to notify on any delays with a given route at a specific stop (“The vehicle x is delayed by y minutes”). Last but not least the Service Alerts feed contains information about any major service disruptions like accidents, construction work or emergencies (“Stop x is not available due to maintenance”). Even if this information is essential to build a full blown passenger information system it's not in scope for this project. Because this project first and foremost sets out to mirror realtime vehicle positions only the vehicle location feed will be used.

Nevertheless the following table provides an overview of all feeds that make up GTFS Realtime.

GTFS Realtime Feeds	
Vehicle Positions	Information on a vehicle's position i.e. its WGS84 longitude and latitude
Trip Updates	Estimated arrival or departure time for any given stop
Service Alerts	Notifications of any service disruption caused by technical issues, emergencies, etc.

Table 3: GTFS Realtime Feeds

3.3.2. The Protocol Buffer Format and GTFS Realtime

When specifying GTFS Realtime Google decided to use the Protocol Buffer format as the underlying data format. This decision may have been influenced by the fact that Google already used this format internally to enable efficient data storage and exchange. The following excerpt from Google's developer site summarizes the key features of the Protocol Buffer Format (GOOGLE 2022D).

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. It's like JSON, except it's smaller and faster, and it generates native language bindings.

Although the Protocol Buffer format is considered the native GTFS Realtime format some GTFS data providers decided to make their GTFS Realtime data available in other popular formats like JSON or XML.

Since the Protocol Buffer format is binary it is required to have additional tooling in place to read and write data. The `protobuf` format relies on `*.proto` files to define the data structure of Protocol Buffer files. These `*.proto` files are roughly comparable to schema files in XML. Other pieces of software needed to work with the Protocol Buffer format are the `protoc` compiler that uses the previously mentioned `*.proto` file and programming language specific bindings to read Protocol Buffer data based on a `*.proto` definition.

From a developer's perspective I felt that the JSON format was compact enough to ensure sufficient performance. I felt that the additional performance gains resulting from the even more compact Protocol Buffer format could be neglected. On the plus side of using JSON were the human readable data format, its lesser dependency on additional tooling like `protoc` or language bindings and the ability to use tried and true libraries for JSON processing.

So the assessment of the Protocol Buffer format when comparing it to the benefits of JSON finally informed the decision to go forward with JSON as the preferred data format.

3.3.3. Alternative Formats as JSON and XML

As discussed previously some GTFS Realtime data providers offer GTFS data not only in Protocol Buffer format but in additional data flavours too. TriMet as one of the GTFS pioneers is one of them. While the Trip Updates and Service Alerts feeds are still served in Protocol Buffer format the Vehicle Locations feed is served in JSON or XML style. From TriMet's documentation it looks like the Protocol Buffer format for the Vehicle Locations feed has been abandoned by TriMet and has been replaced by the more convenient JSON and XML formats (TRIMET 2022B).

Other GTFS Realtime data providers like the Massachusetts Bay Transportation Authority (MBTA) as an example still deliver Vehicle Locations in Protocol Buffer format but also have other data format options like JSON and Enhanced JSON. While the plain JSON feed complies exactly with the specification the Enhanced JSON feed adds value by enriching MBTA feed data with MBTA specific extensions (MBTA 2022A).

4. TriMet Open Data

4.1. Tri-County Metropolitan Transportation Agency (TriMet)

Who or what is TriMet? The short answer is that TriMet takes care of the public transportation needs in Portland, Oregon, United States. The following “mission statement” from TriMet’s excellent web presence outlines what TriMet is up to (TRIMET 2022A).

TriMet provides bus, light rail and commuter rail service in the Portland, Oregon, region. Our transportation options connect people with their community, while easing traffic congestion and reducing air pollution — making our region a better place to live.

More detailed information on TriMet’s past and present can be also found at Wikipedia (WIKIPEDIA 2022A).

TriMet, formally known as the Tri-County Metropolitan Transportation District of Oregon, is a public agency that operates mass transit in a region that spans most of the Portland metropolitan area in the U.S. state of Oregon. Created in 1969 by the Oregon legislature, the district replaced five private bus companies that operated in the three counties: Multnomah, Washington, and Clackamas. TriMet started operating a light rail system, MAX, in 1986, which has since been expanded to five lines that now cover 59.7 miles (96.1 km), as well as the WES Commuter Rail line in 2009. It also provides the operators and maintenance personnel for the city of Portland-owned Portland Streetcar system. In 2021, the system had a ridership of 44,508,200, or about 148,500 per weekday as of the first quarter of 2022.

4.2. TriMet GTFS Data

TriMet maintains an excellent web site but its developer site is even better. TriMet’s GTFS related data are available at a dedicated section of this site (TRIMET 2022B).

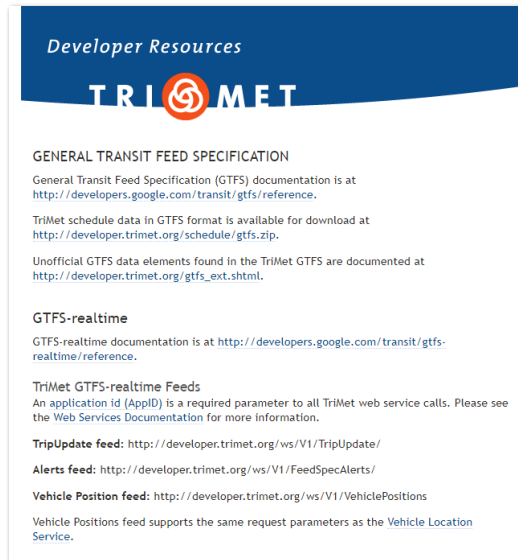


Figure 12: TriMet developer site GTFSS data (Source: TriMet)

Since my focus of interest was primarily on the vehicle positions I will cover the Vehicle Location feed in the following part in more detail.

4.2.1. TriMet Vehicle Location Service

The vehicle location service is provided by TriMet as a web service which can be accessed by using http or https protocols. The base URL to connect to the service is

```
http(s)://developer.trimet.org/ws/v2/vehicles
```

Feed data can be gathered in JSON or XML format depending on the XML parameter of the service call. The XML parameter can be set to false or true. I went for getting the data in JSON format due its less verbose format when compared to XML. A sample call to retrieve JSON data could look like this:

```
http(s)://developer.trimet.org/ws/v2/vehicles?appID=appKey&xml=false
```

It is noteworthy that each time the service is called an application id issued by TriMet must be submitted as part of the call otherwise the call is rejected. This is because TriMet requires any developer to register and to apply for an application id before accessing TriMet feed data.

```
http(s)://developer.trimet.org/ws/v2/vehicles?appID=appKey&xml=false
```

But once a service call has been run successfully the response contains rich information on all vehicles in operation. A sample response in JSON format could look like this:

```
{
  "resultSet": {
    "queryTime": 1661770670264,
    "vehicle": [
```



```
{
  "expires": 1661771194347,
  "signMessage": "Green Line to City Ctr",
  "serviceDate": 1661756400000,
  "loadPercentage": null,
  "latitude": 45.5311233,
  "nextStopSeq": 1,
  "source": "tab",
  "type": "rail",
  "blockID": 9061,
  "signMessageLong": "MAX Green Line to City Center/PSU",
  "lastLocID": 8370,
  "nextLocID": 8370,
  "locationInScheduleDay": 14195,
  "newTrip": false,
  "longitude": -122.5637122,
  "direction": 1,
  "inCongestion": null,
  "routeNumber": 200,
  "bearing": 346,
  "garage": "RUBY",
  "tripID": "11764077",
  "delay": -59,
  "extraBlockID": null,
  "messageCode": 1086,
  "lastStopSeq": 9,
  "vehicleID": 222,
  "time": 1661770654636,
  "offRoute": false
}
]
}
```

Of course a typical JSON response result set will contain much more instances of “vehicle”. While working on the project I found that at peak times about 400 vehicles operate at the same time.

4.2.2. TriMet Stops Data

Data returned by the vehicle location service also encode the previous and next stop that the vehicle encounters on its route. Since I wanted to display stop names at an info panel in the final application the names of the stops were important information. However, the stops were encoded numerically in the GTFS Realtime data stream and are not available in plain text. To resolve the numerical field `stop_id` to a full text `stop_name`, the GTFS Schedule file `stops.txt` had been used. Since GTFS Schedule data like `stops.txt` are provided in CSV format it was pretty straightforward to map `stop_id` to `stop_name`.

The following image shows how the JSON response from the Vehicle Location Service and the static stops.txt file are linked to each other.

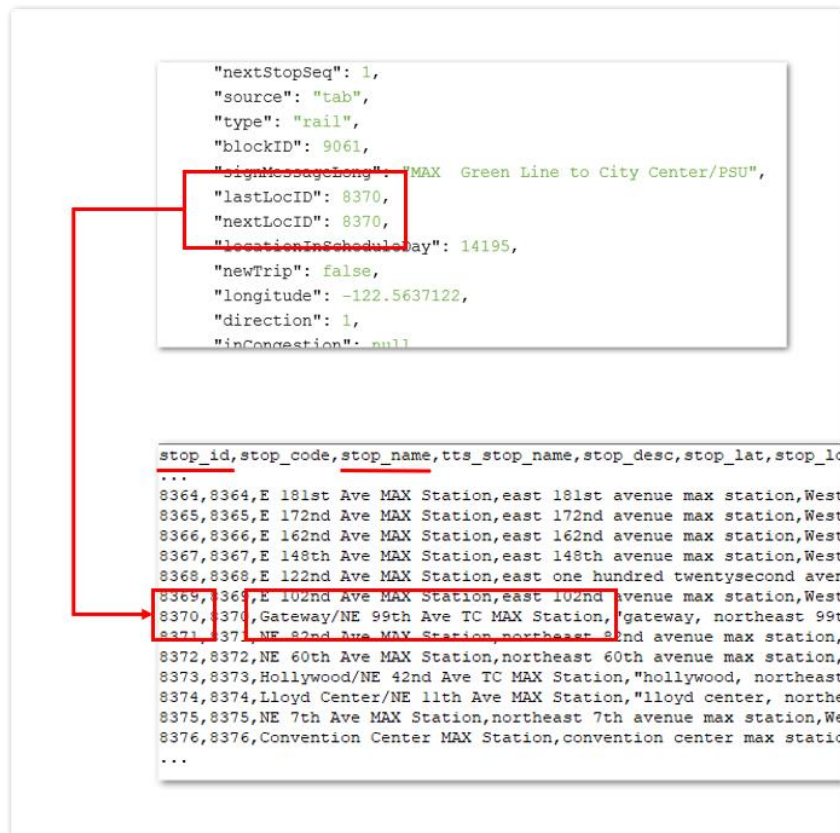
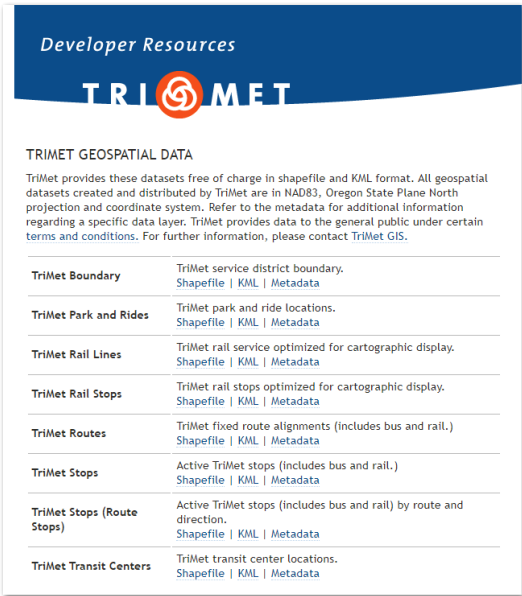


Figure 13: How a stop_id can be resolved to a stop_name

4.3. TriMet GIS Data

Geospatial data related to TriMet could be also retrieved from TriMet's developer resources site (TRIMET 2022c). As seen in the following screenshot the site provides TriMet geospatial data in ESRI Shapefile and KML format. Since working with ESRI ArcGIS Pro throughout the project was a given the Shapefile vector feature format was an obvious choice to move forward with.



The screenshot shows the 'Developer Resources' section of the TriMet website. It features the TriMet logo and a heading 'TRIMET GEOSPATIAL DATA'. Below this, a paragraph explains that TriMet provides datasets free of charge in shapefile and KML format. A table lists various datasets with their descriptions and available formats (Shapefile, KML, Metadata).

Dataset Name	Description	Formats
TriMet Boundary	TriMet service district boundary.	Shapefile KML Metadata
TriMet Park and Rides	TriMet park and ride locations.	Shapefile KML Metadata
TriMet Rail Lines	TriMet rail service optimized for cartographic display.	Shapefile KML Metadata
TriMet Rail Stops	TriMet rail stops optimized for cartographic display.	Shapefile KML Metadata
TriMet Routes	TriMet fixed route alignments (includes bus and rail.)	Shapefile KML Metadata
TriMet Stops	Active TriMet stops (includes bus and rail.)	Shapefile KML Metadata
TriMet Stops (Route Stops)	Active TriMet stops (includes bus and rail) by route and direction.	Shapefile KML Metadata
TriMet Transit Centers	TriMet transit center locations.	Shapefile KML Metadata

Figure 14: TriMet developer site GIS data (Source: TriMet)

Out of the different spatial data from the TriMet site I selected the following ones as building blocks for the final Digital Twin:

- TriMet Boundary
- TriMet Routes
- TriMet Stops

While “TriMet Boundary” is a polygon feature that describes the area that is serviced by TriMet, “TriMet Routes” is a polyline feature that covers all routes that make up the TriMet network. Finally the “TriMet Stops” point feature holds all stops along the routes. All features carry rich metadata that can be easily accessed via the respective attribute tables.

The remaining features “TriMet Rail Lines” and “TriMet Rail Stops” were data subsets of “TriMet Routes” and “TriMet Stops” respectively so these features were redundant and could be neglected.

I also felt that features like “TriMet Park and Rides” and “TriMet Transit Centers” were not relevant in the context of building a Digital Twin of live traffic so I neglected them as well.

4.3.1. ArcGIS Pro Preparation

Assuming that the GIS data of interest have been downloaded I needed to do the following steps for data preparation with ArcGIS Pro:

- Add a folder connection to the folder that is hosting the TriMet shapefiles
- Get the shapefiles into the local geodatabase by using the tool “Export → Feature Class(es) To Geodatabase...”
- And finally add the features to the current map

It turned out that coordinate systems needed to be aligned because all TriMet data had been using the projected coordinate system “NAD 1983 HARN StatePlane Oregon North FIPS 3601 (Intl Feet)” or EPSG 4152.

Since I knew from the documentation that the “ArcGIS Maps SDK for Unity” could only cope with the projected coordinate system “WGS 1984 Web Mercator (auxiliary sphere)” or EPSG 3857 I needed to project the coordinate system to have WGS 1984 as a common coordinate system.

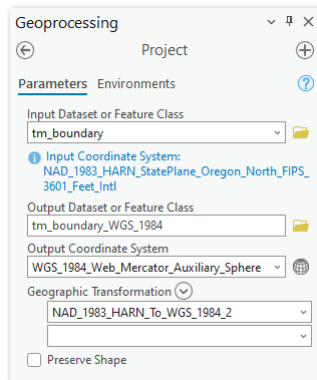


Figure 15: Projecting to “WGS 1984 Web Mercator”

A check of “Map Properties → Transformation” showed that a transformation was no longer necessary after using the “Project” tool and I repeated the same steps for all layers with “NAD 1983 HARN StatePlane Oregon North FIPS 3601 (Intl Feet)”.

The following sections in this chapter provide some more detailed information and visuals of the used TriMet features.

4.3.2. TriMet Boundary

Feature Layer <i>TriMet Boundary</i>	
Feature Layer Name	tm_boundary
Feature Layer Type	Polygon
Feature Layer Description	Describes the area that is serviced by TriMet public transportation agency

Table 4: Feature layer TriMet Boundary

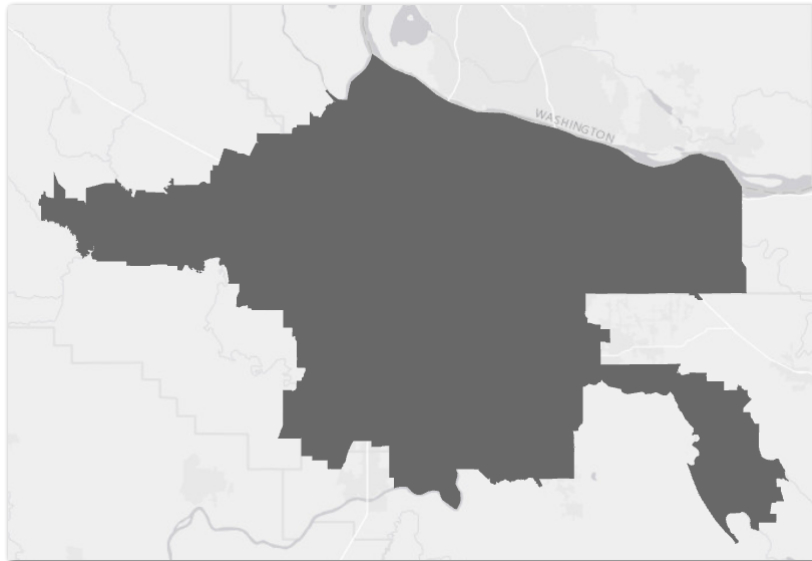


Figure 16: Feature layer TriMet Boundary

4.3.3. TriMet Routes

Feature Layer TriMet Routes	
Feature Layer Name	tm_routes
Feature Layer Type	PolyLine
Feature Layer Description	Describes the routes that are serviced by TriMet public transportation agency

Table 5: Feature layer TriMet Routes

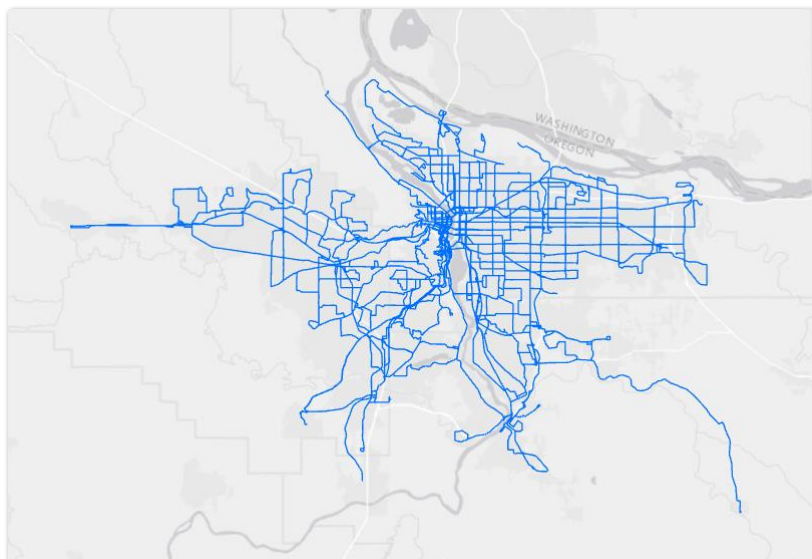


Figure 17: Feature layer TriMet Routes

4.3.4 TriMet Stops

Feature Layer <i>TriMet Stops</i>	
Feature Layer Name	tm_stops
Feature Layer Type	Point
Feature Layer Description	Describes all stops that are serviced by TriMet public transportation agency

Table 6: Feature layer *TriMet Stops*

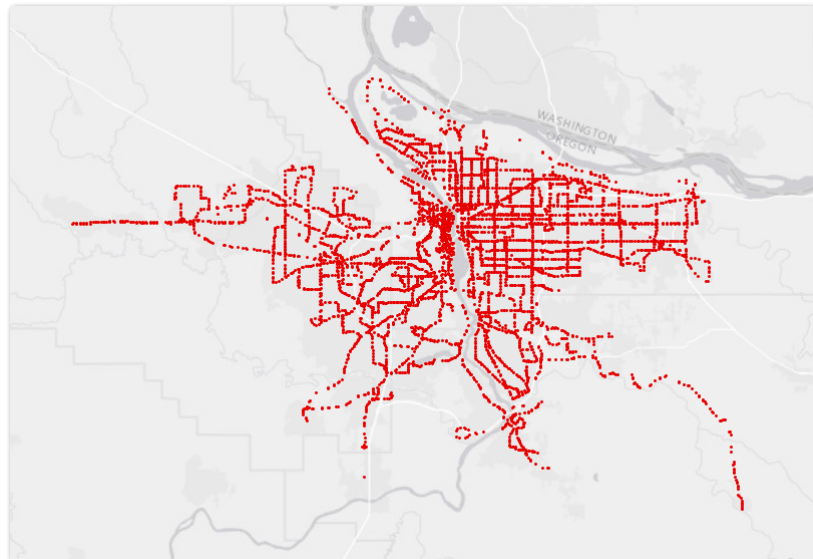


Figure 18: Feature layer *TriMet Stops*

4.3.5. Derived TriMet MAX Routes

I also wanted to extract data from the TriMet Stops and Routes features that are used by TriMet's light rail which is called MAX. The reason was that I imagined to have two "subnetworks" with the first one used by buses and the second one used by light rail vehicles. It helped that the respective attribute tables of these layers contained a field called `type` that allowed to separate MAX related data from BUS related data. So I used the ArcGIS Pro tool "Feature Class To Feature Class" to create MAX only routes and stops features.

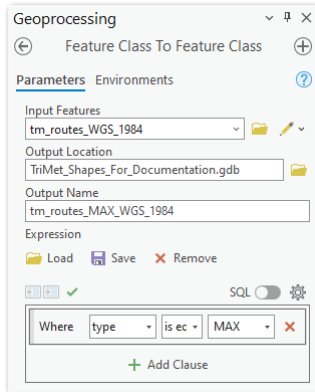


Figure 19: Separating MAX routes from all routes

Feature Layer TriMet MAX Routes	
Feature Layer Name	tm_routes_MAX
Feature Layer Type	PolyLine
Feature Layer Description	Describes all routes that are serviced by TriMet MAX light rail

Table 7: Feature layer TriMet MAX Routes

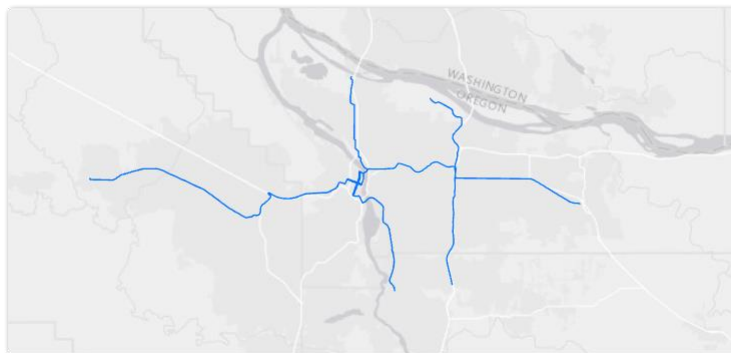


Figure 20: Feature layer TriMet MAX Routes

4.3.6. Derived TriMet MAX Stops

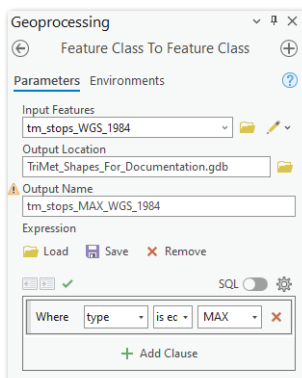


Figure 21: Separating MAX stops from all stops

Feature Layer <i>TriMet MAX Stops</i>	
Feature Layer Name	tm_stops_MAX
Feature Layer Type	Point
Feature Layer Description	Describes all stops that are serviced by TriMet MAX light rail

Table 8: Feature layer *TriMet MAX Stops*

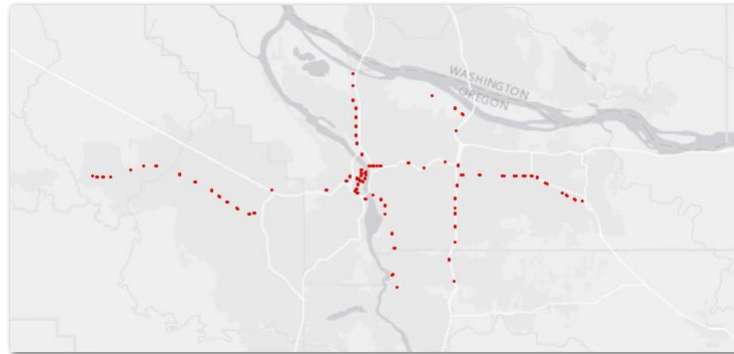


Figure 22: Feature layer *TriMet MAX Stops*

5. Hardware Environment

5.1. Development Laptop #1

The major part of development for this project has been done on “Development Laptop #1”, a three-year-old system that is beginning to show its age. Although it still is a capable system for most development tasks its last generation GPU gets under pressure when it comes to 3D heavy tasks as in some cases with ArcGIS Pro and most of the time with Unity. Frame rates sometimes dropped so low that the results were too poor to be acceptable.

Technical Data Development Laptop #1	
Model Name	Dell XPS 15 9570
Processor	Intel Core i9-8950HK 8 th Generation
Memory	32 GB DDR4 Memory 2.666 MHz
Storage	1 TB PCIe SSD Disk
Graphics	NVIDIA GeForce GTX 1050 Ti with 4 GB GDDR5
Operating System	Windows 10 Pro 64 Bit
Year of Purchase	2019

Table 9: Technical Data Development Laptop #1

5.2. Development Laptop #2

Most of the time performance on “Development Laptop #1” had been good enough but in some cases I switched to “Development Laptop #2” because I wanted to know how the application would be affected by using a current generation GPU. And indeed it was amazing to see how graphic performance had been boosted by using an RTX GPU.

Technical Data Development Laptop #2	
Model Name	Dell XPS 17 9710
Processor	Intel Core i7-11800H 11 th Generation
Memory	16 GB DDR4 Memory 3.200 MHz
Storage	1 TB PCIe SSD Disk
Graphics	NVIDIA GeForce RTX 3050 with 4 GB GDDR6
Operating System	Windows 11 Home 64 Bit
Year of Purchase	2022

Table 10: Technical Data Development Laptop #2

All development has been done with an internet connection in place. In terms of bandwidth the connection was capable of transmitting 400 Mbit/s when downloading data and 20 Mbit/s when uploading data.

6. Software Environment

6.1. Postman API Platform

Since access to GTFS Realtime data at TriMet is based on web services I found it helpful to have a convenient tool for testing web services on a very basic HTML level. I felt I had to get the basics right first to feel confident enough to take the next steps. I considered this a precondition to implement high level logic like setting up a REST client programmatically. My tool of choice has been the Postman API Platform. I already knew its capabilities quite well because I had used it in former projects. I expected Postman to enable me to fire very basic HTTP requests against the TriMet web services. Using Postman would also allow me to immediately check the outcome. So in this section the Postman API Platform is introduced along with some sample tests of the TriMet web services.

6.1.1. Overview

The Postman API Platform is meant to ease a web developer's life by providing convenient test options for APIs. The main focus regarding APIs is to help with testing web based REST APIs that are sitting on the http protocol. It is possible to create http requests based on the most popular request types like `GET`, `POST` and `PUT` just to name a few. Request headers and parameters can be modified in an easy to use, web based user interface. After setting up a request it can be sent by a clicking a button. If the request has been successful the response is immediately available for further evaluation. In case of error the interface provides an http response status code. For example this could be the well known `401 Unauthorized` response if there is some issue with access rights. If the web server administrator has a sense of humour the request could even yield a status code like `418 I'm a teapot` (MDN 2022B). But kidding aside these status codes can oftentimes help to understand the cause when something went wrong. A comprehensive list of response status codes can be found at the MDN Web Docs site, the former Mozilla Developer Network that is hosting comprehensive documentation for web developers. Nowadays it is also supported by Microsoft, Google and other tech companies (MDN 2022A).

6.1.2. Workflows

The Postman website (PM 2022A) requires registration and after signing up a "Workspace" can be set up to prepare testing. Tests can be bundled in "Collections" to keep the various tests organized. The following screenshot shows how the workspace looks like with a "TriMet Collection" and several test requests.

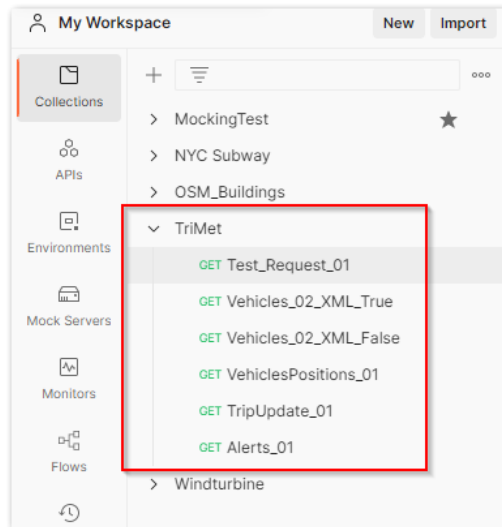


Figure 23: Postman workspace with TriMet collection

By using the collection’s context menu any http request can be added to the collection and be given a unique name. The call I sampled below is the one that I used to make my feet wet with the TriMet web services. It had been pulled from the TriMet developer documentation to get things up and running (TRIMET 2022D). After selecting the GET method the web service URL can be entered. Request headers and parameters are immediately recognized by Postman from the URL and can be modified as needed.

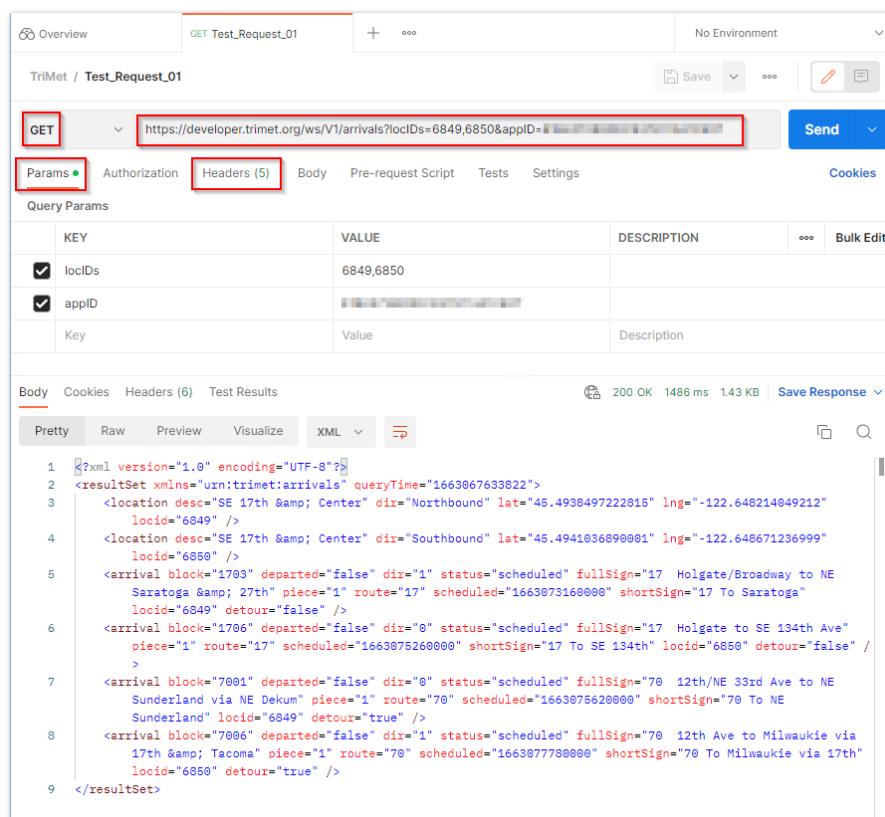


Figure 24: Postman workspace with sample request and response

After having sent the request by pushing the “Send” button the response appears in the lower half of the workspace and can be explored in detail. I used this basic Postman workflow to learn the ropes on how TriMet web services are working in general.

6.2. Microsoft Visual Studio 2019

6.2.1. Overview

Microsoft’s Visual Studio is one of the leading integrated development environments (IDE) used in the IT industry. It is oftentimes used for developing applications targeting the Windows platform. But today Visual Studio is not limited to the Windows platform any more. Almost every important programming language is supported with C++ and C# being two of the most prominent examples. With its rich editing, compiling, debugging and testing capabilities Visual Studio is a one stop shop for any Windows programmer’s needs. For almost 30 years the Visual Studio IDE family has been an important cornerstone for Windows development. By chance I started learning C in the 1980s with one of the early predecessors of Visual Studio that was called Microsoft QuickC at the time. So I know for sure that Microsoft IDEs have been around that long. Even although this had been the humble beginnings of IDEs it was already miles away from pure command line work for compiling and linking applications which was required earlier. So it felt very natural for me to stick with Visual Studio when working on this project aside from its rich development features.

But as of time of building this digital twin I decided not to use the latest version of Visual Studio which would have been Visual Studio 2022. Instead I went with the previous version that is Visual Studio 2019. The reason for this decision was my assumption that the 2019 version would have a greater maturity a few years after its initial release and I hoped for almost all of its bugs being resolved. Since the application types I wanted to develop were a plain command line application and a GUI based Windows application I felt that Visual Studio 2019’s abilities would be more than sufficient to achieve this. The assumed greater robustness of Visual Studio’s 2019 iteration was more important to me than having the latest features of Visual Studio 2022.

So from a version perspective I went with Microsoft Visual Studio Community 2019 (Version 16.11.16 from 2021).

6.2.2. Workflows

In the beginning I thought that using a plain command line application to deal with the GTFS Realtime data stream would be sufficient. But rather quickly I pondered that having a user interface for interacting with the GTFS real-time feed would be more convenient and time saving too, especially in the midst of developing and testing. So the design decision of having a graphical user interface ultimately led to three project types when working with Visual Studio. However, all of these project types rely on C# as a programming language.

1. Console Application

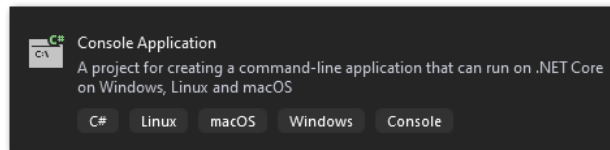


Figure 25: Visual Studio project template for Console Application

This Visual Studio project type is the foundation for building a console application. I used this template for the console application “GTFS Realtime Data Simulator”. This application simulates the GTFS data stream in an offline environment when it’s not possible to access the GTFS Realtime service directly due to a missing internet connection. The simplicity of this application did not require a sophisticated user interface from my point of view.

2. Windows Presentation Foundation (WPF) Client Application

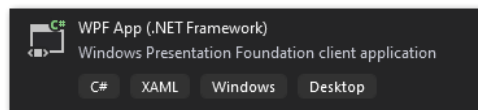


Figure 26: Visual Studio project template for WPF App

As pointed out before I deemed it helpful to have a user interface for the more elaborated application “GTFS Data Loader”. Having a user interface would make it easier to interact with the GTFS Realtime feed. I imagined to leverage the usual suspects when it comes to Windows controls, things like Buttons, Text Fields or an OpenFileDialog to enable quick access to a file folder.

But when it comes to Windows UIs there are quite a few technologies to choose from, most notably the WinForms framework, the WPF framework and the WinUI framework that is closely coupled with Microsoft’s Universal Windows Platform (UWP). With WinForms being an outdated technology already and WinUI adding more complexity in terms of UWP development I felt that using a conservative but pragmatic approach would be best. So I picked WPF as a mature UI technology that has already stood the test of time because it has been around since 2006 (WIKIPEDIA 2022B).

3. Visual Studio as part of the Unity Editor

Even if I will introduce Unity in more detail at a later section it might be appropriate to already come up with a slogan like description of the Unity Editor. Unity can be considered a full blown 3D development platform in its own right and I am somewhat tempted to call it the “Visual Studio for 3D development”. In fact it is one of

the leading development environments for building interactive and immersive real-time 3D content. It is even recommended by Microsoft to target its augmented reality device HoloLens 2 just to name a prominent example (MSFT 2022A). However, 3D artifacts are one part of the Unity equation. The other part is adding logic to 3D assets. A simple example for adding logic would be to let a 3D cube rotate along its Y axis. This can be achieved by attaching a C# script to the cube that takes care of the cube rotation. So a big part of creating a project with Unity is its scripting side that controls 3D artifacts' behaviour and quite literally breathes life into a 3D scene.

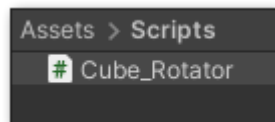


Figure 27: C# script within the Unity Editor

To make the development workflow as smooth as possible there is a seamless integration between Visual Studio and Unity Editor. This makes it quite convenient for a designer/developer to write and test C# code in a Unity environment. In fact a double click on a to be modified C# script within a Unity project is enough to bring up Visual Studio and start working on the code. After the code has been changed it is sufficient to save the changed file and let Unity recognize the script change. As soon as a code change has been spotted by Unity its scripting backend takes care of compiling the modified source code again without any user interaction.

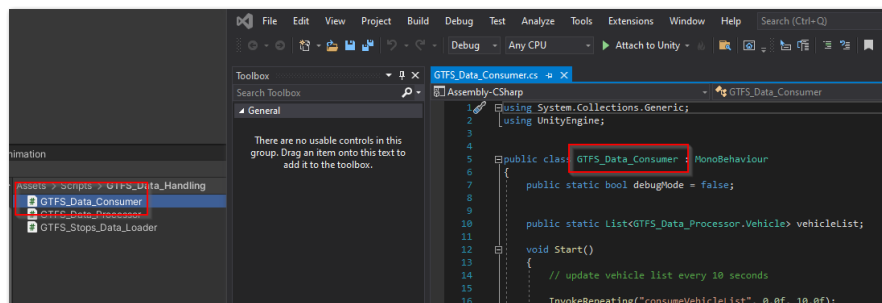


Figure 28: Unity Editor and Visual Studio side by side

To summarize working with Visual Studio three distinct workflows have been used. There are two more conventional workflows within Visual Studio to develop standalone console and GUI applications. The third workflow is a more integrated one thanks to a close coupling between Unity Editor and Visual Studio.

6.3. ESRI ArcGIS Pro

6.3.1. Overview

Since this project is meant for a GIS savvy audience I will keep the introduction of ArcGIS Pro short and sweet. ArcGIS Pro is released by Environmental Systems Research Institute (ESRI) which is one of the leading providers of Geographic Information Systems (GIS). ESRI offers a vast range of GIS applications that target desktop, server and mobile platforms while having a strong focus on use cases that are built upon spatial data. As part of the ESRI software suite the ArcGIS Pro application can be considered the “flagship” desktop application when it comes to dealing with GIS data. Its features include best in class tools for creating, importing, transforming, analyzing, managing and visualizing spatial data.

During this project I started out using a 2.x version of ArcGIS Pro but in the final stages I ended up working with version 3.02.

I also needed to come up with some custom workflows in ArcGIS Pro during this project. So the next section will cover these workflows in more detail.

6.3.2. Workflows

Looking a little bit ahead it was mandatory to use one of ArcGIS Pro’s formats that is called Scene Layer Package, a data format that usually has an `*.slpk` file extension. It was mandatory because this is one of the data formats that ESRI’s ArcGIS Maps SDK for Unity is willing to accept. This format packages an ArcGIS Pro 3D scene layer in a way that it can be easily shared as a file. 3D scene layers are also the type of layer that is recommended by ESRI for visualizing vast amounts of 3D data in a scene.

I had learned from the ArcGIS Maps SDK for Unity documentation that 3D Object Scene Layers would be supported and could be consumed as local `*.slpk` files (ESRI 2022A). Since I had in mind to use various shape files provided by TriMet in the final Digital Twin application I needed to find a way of transforming `*.shp` files to `*.slpk` files. The following steps try to outline how I managed to do this. To shed more light on this workflow I selected the `tm_boundary` shape file provided by TriMet as an example. The `tm_boundary` shape file is a plain polygon feature describing the Portland region serviced by TriMet. This layer has been already more thoroughly discussed in previous section 4.3.2. *TriMet Boundary*.

1. Importing the `tm_boundary` shape file in ArcGIS Pro

The import had been achieved by creating a folder connection using the folder that hosts the `tm_boundary` shape file. Although a reprojection to “WGS 1984 Web Mercator (auxiliary sphere)” or EPSG 3857 turned out to be necessary this had been a pretty straightforward step.

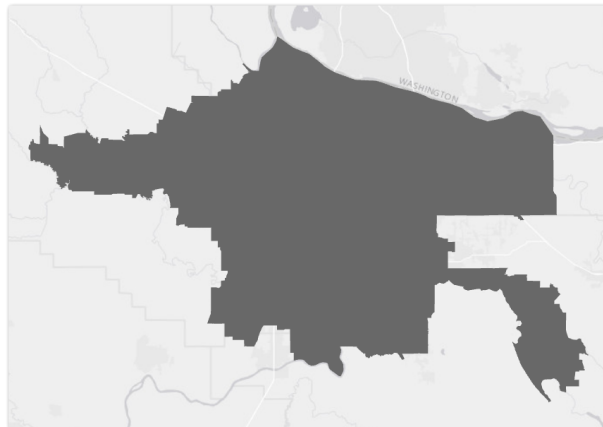


Figure 29: Layer tm_boundary in ArcGIS Pro

2. Converting the map to a local 3D scene
3. Adding a height attribute to the attribute table

Since I imagined to have the TriMet service area as a kind of 3D canvas of the scene I added a `height` attribute to the attribute table and set it to 1000 m to have a more 3D like foundation of the scene.

OBJECTID *	Shape *	area_sq_mi	acres	height	Shape_Length	Shape_Area
1	Polygon	533	341554	1000	55736,837057	2810844823,898222

Figure 30: Attribute height in attribute table tm_boundary

4. Extruding the tm_boundary feature layer

After extending the attribute table I used the newly created height attribute to extrude the layer to get a first 3D representation of the TriMet service area.

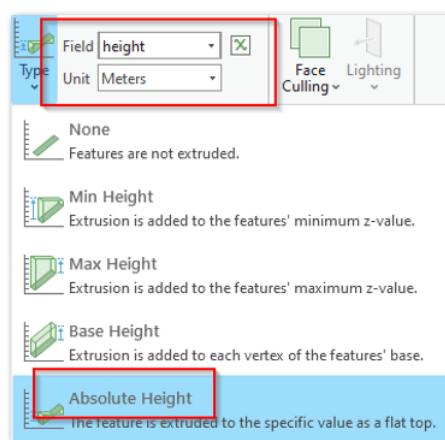


Figure 31: Attribute based extrusion of layer tm_boundary



Figure 32: First 3D representation of layer tm_boundary

5. Getting from a Polygon to a Polygon Z shape

As an intermediate step and to keep the height information I used the tool “Feature To 3D By Attribute” from the 3D Analyst toolset.

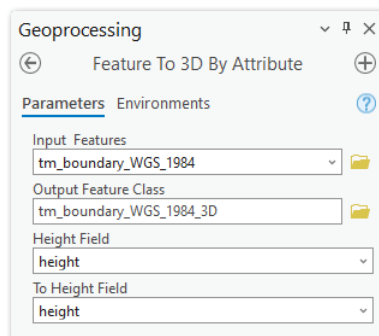


Figure 33: Using the tool “Feature To 3D By Attribute”

As a result the shape type in the attribute table had been marked as “Polygon Z”.

OBJECTID	Shape	area_sq_mi	acres	height	Shape_Length	Shape_Area
1	Polygon Z	533	341554	1000	455736,837057	2810844823,898222

Figure 34: Confirming the Polygon Z shape type in attribute table

6. Getting a multipatch version from a 3D layer

Since I knew from the ArcGIS Pro documentation that a multipatch feature is required to build a *.slpk package the next step was to transform the 3D layer by using the tool “Layer 3D To Feature Class”.

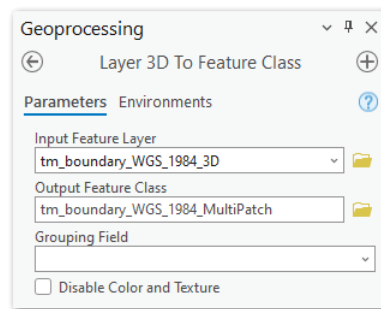


Figure 35: Creating a MultiPatch feature

As an outcome the attribute table of the resulting feature carried “MultiPatch” as a shape type.

 A screenshot of the attribute table for the layer 'tm_boundary_WGS_1984_MultiPatch'. The table has columns: OBJECTID, Shape *, area_sq_mi, acres, and height. The first row shows OBJECTID 1, Shape * MultiPatch, area_sq_mi 533, acres 341554, and height 1000. A red box highlights the 'Shape *' column and its value 'MultiPatch'.

OBJECTID	Shape *	area_sq_mi	acres	height
1	MultiPatch	533	341554	1000

Figure 36: Confirming the MultiPatch shape type in attribute table

7. Building a *.slpk package from a MultiPatch layer

The concluding step of creating an *.slpk package was to use the tool “Create 3D Object Scene Layer Content”. It created the *.slpk file that would be shared with Unity later.

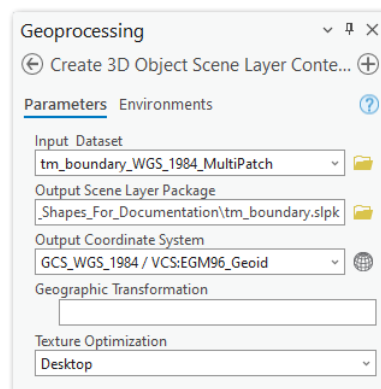


Figure 37: Creating an offline 3D Scene Layer Package

The previous steps are meant to show the general workflow how to get from *.shp file to *.slpk file. The same steps had been applied to other TriMet shape files such as tm_routes and tm_stops. The resulting *.slpk files served as input for building out the main digital twin scene in Unity finally.

6.4. Unity Real-Time 3D Engine

The next sections will take a closer look at Unity as one of the most widely used engines for real-time 3D visualization. I decided to work with Unity because I already had quite a bit of experience with it. But for the sake of completeness it is noteworthy that there is also Unreal Engine created by Epic Games. Unity and Unreal are the undisputed industry leaders when it comes to real-time 3D engines. Unreal Engine is at least as powerful as the Unity Engine but has a somewhat steeper learning curve because Unreal Engine scripting is natively done in C++. Just to try a comparison: In the automotive market there are premium brands like Mercedes and BMW and I feel that something similar is true with Unity and Unreal in the 3D industry. Both Unity and Unreal have their pros and cons but both are perfectly capable of rendering high end 3D graphics. There is nothing wrong with picking one or the other. But given my previous experience with Unity I decided to work with this engine.

From a version perspective I worked with Unity Editor version 2021.3.4f1 LTS on this project.

6.4.1. Overview

Unity is a real-time 3D engine developed by Unity Technologies (UNITY 2022A).

The term engine in an IT context often refers to a complex software system that bundles a set of features to achieve a certain goal with a strong focus on performance. For example a database engine takes care of all the heavy lifting under the hood if an SQL query is processed. Since a database query should be as fast as possible usually a lot of low level programming in C++, C or even assembler goes into building engines to get them very fast in what they are doing. Along the same lines a real-time 3D engine takes care of rendering a 3D scene to a computer screen as fast as possible.

Performance of real-time 3D engines is often measured in frames per second with the term frames per second (fps) referring to the number of images an engine can render to the screen within a second. As a rule of thumb 60 frames per second is widely accepted as a baseline because the human eye begins to perceive fluent movements by this rate. But a frame rate of 60 fps also means that every 0.017 seconds the graphics subsystem needs to display a single frame on the screen. To achieve this kind of speed it is necessary to have carefully crafted low level code that talks straight to the Graphics Processing Unit (GPU).

But for now it might be enough to think of a real-time 3D engine as a quite complex and highly optimized piece of software that is able to bring 3D artifacts to a screen very fast.

When Unity saw the light of day many years ago it had a strong focus on developing games. It was meant to enable orchestration of 2D or 3D content with program logic and other content with the final goal of creating a game. Over the years the Unity engine went through lots of iterations after its first release in 2005. And since about 2015 the scope of Unity applications began to widen. Some industries found that advanced 3D visualization features could help with product design and reviews, with training, with maintenance tasks and also with marketing. Speaking of marketing product configurators turned out to be one of the compelling examples how real-time 3D technology could be put to good use with creating an engaging customer experience.

With the game industry still being the core of its business today Unity now supports a wide range of non-gaming industries too. Some of the most prominent industries where real-time 3D has been embraced are automotive, engineering, architecture and construction industries. More and more industries beyond the gaming realm have recognized the benefits of using real-time 3D content for design, maintenance and training as some of the more widely adopted use cases. The rise of Virtual Reality and Augmented Reality devices like Oculus Quest or Microsoft HoloLens also led to a new wave of use cases that had not been considered before. All of these devices need to be fed with 3D content and often Unity operates as a kind of hub for pushing 3D content to these platforms.

6.4.2. Definition and Key Features

This section tries to give a definition for a real-time 3D engine along with some of its key features. I already touched upon the term engine referring to a highly optimized software system. A real-time 3D engine can be considered as a software system based on graphics technology that enables real-time 3D technology.

Perhaps it might be helpful to compare movie technology to real-time 3D technology to see what sets them apart. Comparing a negative to a positive might sometimes help to spot the differences. Movies can also open a door into a 3D world even if this world is on a flat silver screen. But a movie is meant for passive consumption, there is no way for an audience to interact with the movie content. The movie audience is limited to “reacting” to the movie content in some way.

With real-time 3D content things are different. One of the key features of real-time 3D technology is the option to interact with the content in one way or another. To stick with the previously mentioned product configurator: A well known example might be a car configurator. It allows the selection of different car coating colours just to get a more lifelike impression of how a car would look like when painted with the selected color.

Selecting the color in an interactive manner is what sets real-time 3D apart. To change a car’s color would not be possible with movie content. At least not right now. So I found interactivity to be a key feature of real-time 3D technology.

Another feature often found with real-time 3D technology is its immersiveness. It is true that movies can be an immersive medium too. Enjoying a movie in a cinema with dimmed lights can be quite an immersive experience. But real-time 3D technology can take this immersion one step further. It is much more immersive navigating a 3D cityscape than having a 2D representation of a city. The cities around us are in three dimensions after all. Different ways of interacting with a 3D representation of a city can be imagined. Zooming in to a point of interest or zooming out to get a better overview are some of them. Another option that comes to mind is to rotate a city model to look at it from different angles.

While these options are already helping to understand the city model better, recent technologies often summarized as “Mixed Reality” contribute to an even more comprehensive immersion. According to Microsoft’s definition “Mixed Reality” as an umbrella term includes “Augmented Reality” and “Virtual Reality” (MSFT 2022B).

Especially when using “Virtual Reality” with Oculus Quest as an example the immersion into a 3D world is almost complete because a user is completely cut off from the surrounding reality (for good or for bad).

Another feature found at least with the two leading real-time 3D engines is the ability to target different platforms with the same 3D content. “Write once and run anywhere” is the promise here. And it is not just a promise. It means that once 3D content is sitting in Unity it can be pushed to desktop platforms like Windows, Linux and Apple but also to mobile platforms like Android, iOS and finally many off mainstream platforms.

With all these considerations in mind the following definition of a real-time 3D engine could capture its major features:

A real-time 3D engine is the technical foundation for bringing real-time 3D technology to life. Real-time 3D technology provides interactive and immersive 3D experiences that can be deployed to various platforms like desktop, mobile or even Mixed Reality devices.

6.4.3. Unity Components

This section tries to give an overview of Unity’s component based architecture. This might help to understand how working with Unity looks like.

The most important entity in Unity is the so called `GameObject`. The name `GameObject` might be confusing at first because working with Unity is not only about games any more. But the name is still showing Unity’s roots in game development and has been kept by Unity over the years. So a `GameObject` can be viewed as a kind of general container for Unity components.

This might lead to a fundamental question: What is a Unity component? I will try to explain this by example. A plain sphere is one of the most basic 3D objects. In Unity speak this sphere is a `GameObject` with multiple components attached to it. A glimpse into the Unity editor might help to get the idea and also to introduce the concept of a Unity scene.

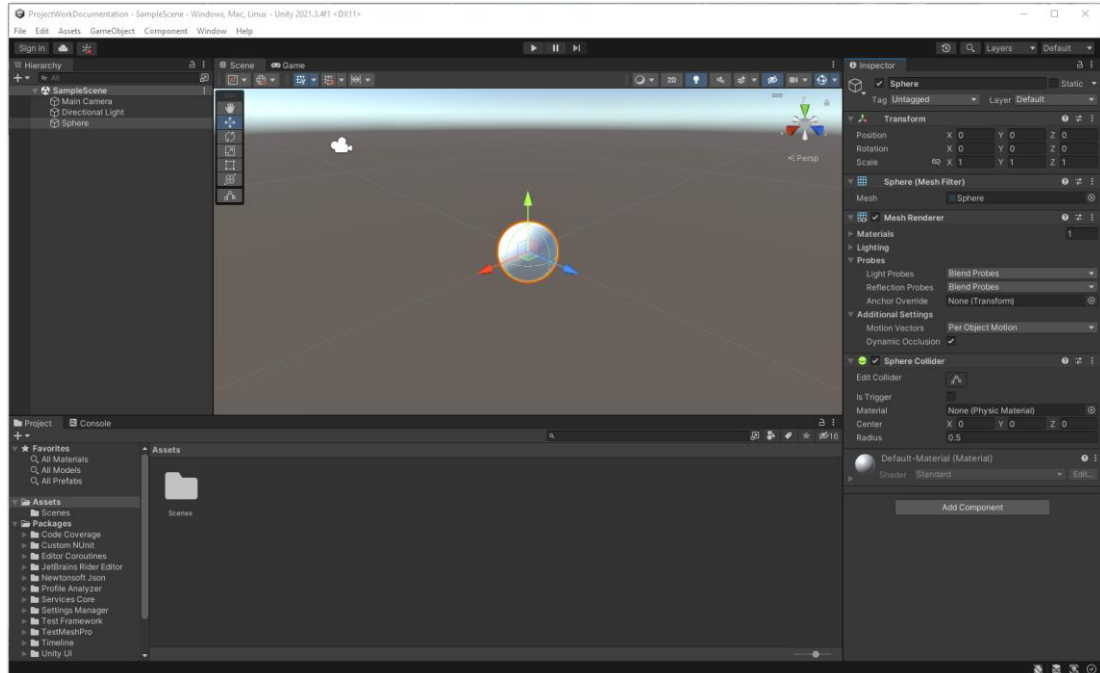


Figure 38: Unity Editor with sample scene opened (Source: Own picture)

When a `GameObject` is a container for components then a Unity scene is a container where Unity's `GameObjects` live in. And a scene can contain as many `GameObjects` as needed. The example scene above shows a single `GameObject` called `Sphere` (In fact there are two other `GameObjects` in the scene named `Main Camera` and `Directional Light` but to focus on the idea of components these are not considered for now).

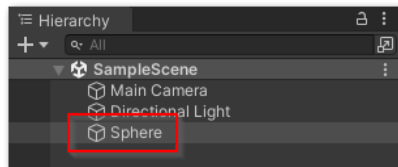


Figure 39: Sphere `GameObject` in a Unity scene hierarchy

The visualization of this sphere in the Unity Editor is pretty sober but this is the visual representation of the `Sphere` `GameObject`.

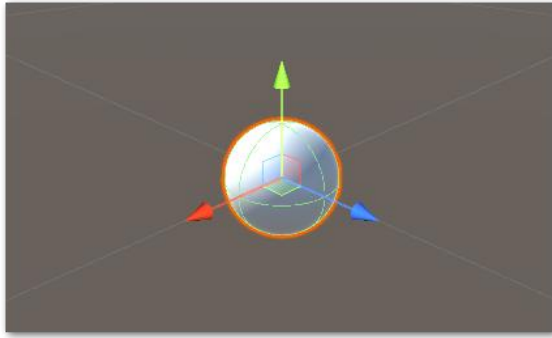


Figure 40: Sphere GameObject displayed in a Unity scene

There is also a panel called “Inspector” in the Unity Editor. The name already hints at its function which is to explore and change properties of `GameObject`s. And these thematically grouped sets of properties attached to a `GameObject`s are known as Unity components.

With the `Sphere` `GameObject` being selected the Inspector displays all the components that are attached to the `Sphere` `GameObject`. Even with this almost primitive 3D object there are already five components attached to the `Sphere` `GameObject`.

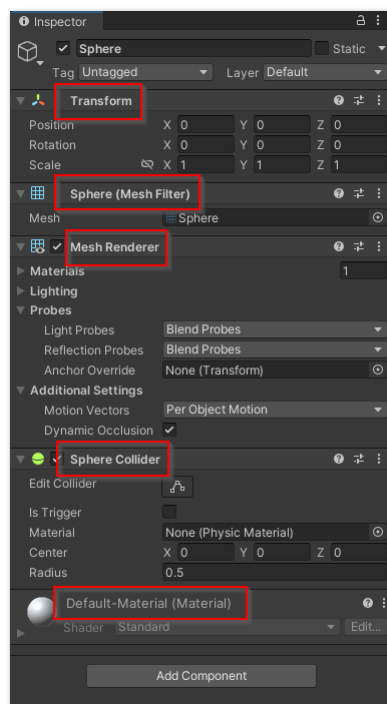


Figure 41: Components attached to Sphere GameObject

These five components are responsible for the `Sphere` `GameObject`'s behaviour and appearance. A closer look at these components might explain the `GameObject`/`Component` relationship even more.

1. Transform

The `Transform` component is one of the most essential components of any `GameObject`. In fact there is no game object that does not have a `Transform` component. In general the `Transform` component is responsible for describing position, rotation and size of any `GameObject` within Unity's builtin coordinate system.

2. Mesh Filter

The `Mesh Filter` component lets Unity know which 3D mesh needs to be rendered to the screen. This component just holds a reference to the mesh to be processed by the engine or to be more precise by its render pipeline. Render pipelines will be explored more thoroughly in one of the next sections.

3. Mesh Renderer

The `Mesh Renderer` is responsible for bringing the `Sphere GameObject` to the screen while taking into account that there is lighting, shadows, material and other features that may influence the appearance of the sphere.

4. Sphere Collider

This component belongs to the group of colliders. Colliders are responsible for creating realistic behaviour when two spheres are touching each other for example. Without a collider those two spheres would move right through each other like some kind of ghost sphere defying the laws of physics. But with a collider attached two spheres would have a lifelike behaviour with bouncing off from each other.

5. Material

The `Material` component finally plays an important role when it comes to the visual appearance of the `Sphere GameObject`. `Materials` bundle a whole set of properties that are responsible for a `GameObject`'s visuals. With color being one of the most basic properties there are many more properties to fine tune a `GameObject`'s look. The sphere may have a metallic appeal, it may have a shiny, polished look, it may look wet, it may receive or cast shadows, it may reflect or emit light and much more. But these are only a few selected options amongst many on how to tweak the appearance of a sphere `GameObject`.

With these sample components examined in more detail it may have become more obvious that components are crucial building blocks when working with Unity. And it is worth noting that the five components discussed so far are only the tip of the iceberg. In fact there are hundreds if not thousands of Unity components to work with and I guess it is way beyond the scope of this project to cover them all. But looking a few pages ahead

the “ESRI Maps SDK for Unity” is another example that relies on components to enter the GIS dimension with Unity.

Moving on to the next section it’s worth pointing out that scripts can be also attached to a `GameObject` as a component of its own. This option opens up a wide range of options in controlling a `GameObject` and the next part will take a closer look at Unity scripting and how it can be used to get a `GameObject` come alive.

6.4.4. The C# Programming Language and Scripting in Unity

C# is one of today’s major programming languages. It can be considered a general-purpose, object oriented programming language. Its first version has been released in 2001 with Microsoft being the driving force behind its development. Until today it is in active development with major new versions released every few years. Starting with its first release the language has been an integral part of Microsoft’s .NET ecosystem, a framework for developing applications targeting the Microsoft Windows platform (WIKIPEDIA 2022c).

In a way C# feels like the Microsoft edition of Java because both C# and Java are closely coupled to their respective runtime software environments. That means those languages do not talk directly to the underlying hardware but to an intermediate software layer that abstracts away from the hardware. To be precise Java relies on the Java Runtime Environment (JRE) for application execution while C# expects the Common Language Runtime (CLR) to be present to run programs.

While C# in general can be used for developing plain and simple applications with a calculator as a very basic example the usage of C# in the Unity environment is somewhat different.

Most often C# scripts are found in Unity as a component attached to a `GameObject`. An attached C# script is closely linked to a `GameObject` and is able to access all its properties like position, rotation or size to name some of the most frequently used. To stick with the previous `Sphere GameObject` example an attached C# script component could enable the sphere to rotate along its Y axis. The C# script’s logic is what lets the sphere rotate. The next screenshots show how a C# script can be attached to a `GameObject` and how that looks like in the Unity IDE.

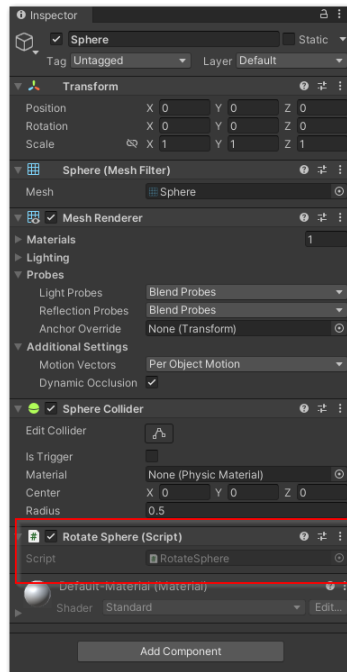


Figure 42: Inspector view of C# script attached to Sphere GameObject

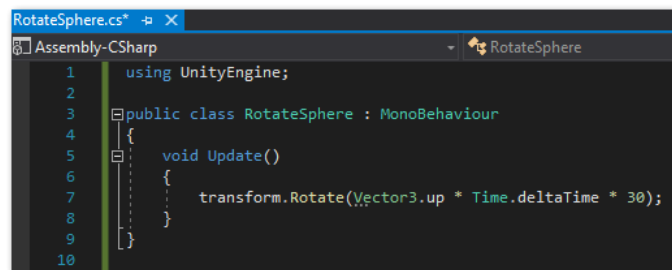


Figure 43: Sample C# script to let a Sphere GameObject rotate

Attaching scripts to `GameObjects` is one of Unity's key features to influence `GameObject` behaviour. The options are almost endless because each and every property of a `GameObject`'s component like a `Material` color or a `Transform` position can be modified by scripts.

The same is true when working with the components of the "ESRI Maps SDK for Unity". One example is to work with the `Basemap` component of ESRI's SDK to switch to a different basemap or to change the position of a point feature during runtime. In fact the Unity scene that displays the Portland network has been built completely with C# scripts that are attached to `GameObjects` and change `GameObject` properties in one way or another.

6.4.5. The Game Loop

The tiny code snippet from the previous section as shown in Figure 43 is a good opportunity to discuss a design pattern that almost all real-time 3D engines have in common. The mentioned snippet has a couple of code lines only but the featured

`Update()` method is an integral part of a pattern often called a “game loop”. It may be more appropriate to call this pattern an “application loop” but the name “game loop” is still alive due to Unity’s roots in game development.

The notion of a game loop may become clearer when compared to a simple calculator application that is written in C. This application would have a `main()` function that is called when the app is run. The `main()` function may ask a user for entering numbers, then performs the calculation and finally displays the calculation result on the screen. After this has been done the application terminates. There may be helper functions to keep the code organized like getting the user input for example. But those would be called only once from the overarching `main()` function. The bottom line is that the application is run and after the result has been printed to the screen it will terminate. In rough strokes the application lifecycle may look similar to the next figure.

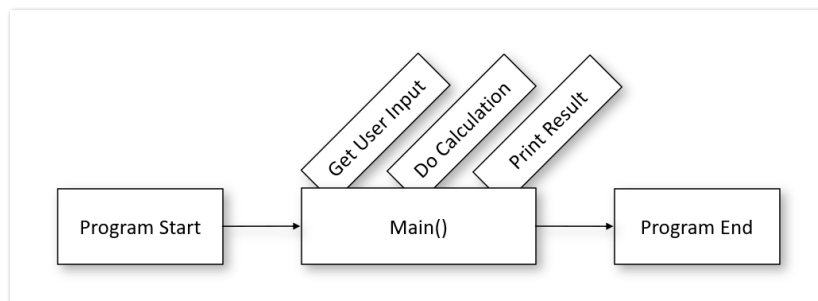


Figure 44: The structure of a simple C program

Compared with the figure above the pattern of using of C# in real-time 3D engines is a different story. For now I will consider the engine as a black box where all the low level magic happens to get artifacts like a spinning 3D sphere to the screen. I guess the inner workings of a real-time 3D engine are way beyond the scope of this project. Right now it might be enough to know that the engine enters a “game loop” when started. As soon as the loop is running it is striving to do three things as fast as possible.

1. Capture and process any user input
2. Keep track of the application’s overall state, for example the position of 3D objects
3. Render the image to the screen

As soon as task 3 is completed this sequence starts all over again. In fact this sequence is executed ad infinitum as long as the engine is running. This endless looping informs the “game loop” naming.

At first sight this sequence may look trivial but a key difference to the simple calculator app described previously is that this sequence can be executed up to several hundred times per second! But the mileage may vary depending on the underlying hardware. How fast the loop executes also depends on the complexity of the 3D scene, the number of 3D objects in the scene, how sophisticated the lighting is and a lot of other dependencies.

But it is always desirable to have these three tasks executed at least 60 times per second. Because this rate means that the 3D scene is rendered to the screen 60 times per second

thus resulting in a display frame rate of 60 frames per second (fps). Starting with 60 fps the movements in a 3D scene begin to be perceived as really fluent by the human eye which is an important part of a good user experience.

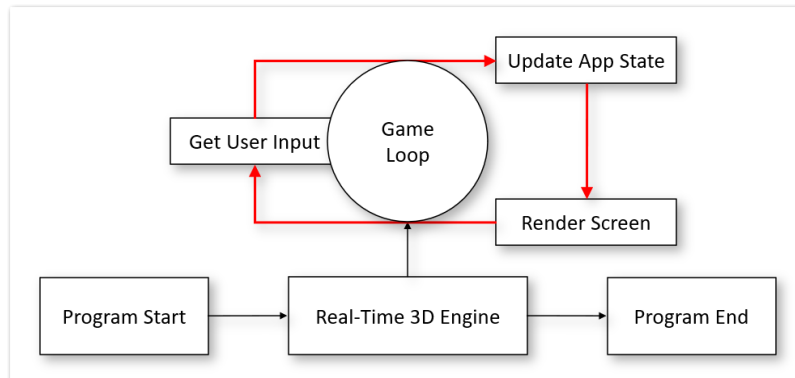


Figure 45: How a real-time 3D engine implements a “game loop”

Coming full circle again from the beginning of this section: The second task of the game loop i.e. “Update App State” is exactly what happens when the engine finds an attached script to a `GameObject` that contains the `Update()` method. By calling the `Update()` function 60 times per second the sphere is spinning just a tiny bit each time. This way the real-time 3D engine ensures that rotation of the sphere is perceived as a fluent movement.

```

1  using UnityEngine;
2
3  public class RotateSphere : MonoBehaviour
4  {
5      void Update()
6      {
7          transform.Rotate(Vector3.up * Time.deltaTime * 30);
8      }
9  }
10
  
```

Figure 46: Sample `Update()` function

To summarize this brief discussion of a “game loop”: There is a close relationship between this loop and every `Update()` method found in any C# code attached to any `GameObject` in Unity. All `Update()` methods in a Unity project are run each time before a new frame is rendered to the screen.

6.4.6. Render Pipelines

Since I have been using Unity for this project it makes sense to briefly discuss the notion of a render pipeline within Unity context. As the name already implies a render pipeline is responsible for “piping” 3D artifacts to the screen. One end of the pipeline is a 3D scene with all its artifacts and the other end is the screen that displays the scene. The render pipeline is responsible for managing all operations in between that are necessary to get a

3D model to a flat 2D screen. Render pipelines are playing a pivotal part in any real-time 3D engine and are responsible for bringing 3D scenes to life.

It is also worth noting that render pipelines can be customized. But this might require some additional skills like shader programming which can be a daunting task. Daunting because this is pretty much like assembler programming for graphic chips.

If tweaking a render pipeline by shader programming can be seen as the high end of pipeline customizing then modifying selected properties of a shader exposed by Unity in its “Inspector” can be considered the much easier end. Tweaking exposed shader properties to get a metallic surface on an object could serve as an example. So most of the time a pipeline along with its default shaders creates enough headroom to build visually appealing applications.

Render pipelines are worth to be discussed at least in general because with Unity there are three different render pipelines to choose from. But not every pipeline works with every device and not every Unity extension known as plugin plays nice with any pipeline. So this is something that needs to be carefully checked beforehand. Obviously the planned use of ESRI Maps SDK for Unity is something to be considered when it comes to render pipelines.

Because of this I want to give a quick overview of the render pipelines available in Unity ordered by performance and requirements.

1. Built-in Render Pipeline

The Built-in Render Pipeline is Unity’s default pipeline. It is also the “oldest” pipeline that has been present in Unity since its beginnings. But with recent advances in graphics technology it sometimes begins to show its age when it comes to leveraging features of the latest GPUs. Although this pipeline has its limitations it is nevertheless a good starting point when developing with Unity. It even performs decently with older hardware because it is the least resource hungry pipeline. I also came across devices that work exclusively with the Built-In Render Pipeline so this is something to be explored upfront when developing for a specific device.

Since the ESRI Maps SDK for Unity does not support the Built-in Render Pipeline it has not been considered for this project.

2. Universal Render Pipeline (URP)

Compared with the Built-in Render Pipeline the Universal Render Pipeline (URP) is more advanced in terms of graphic capabilities. Customizing the pipeline is easier and more flexible so there is no need to fall back to shader programming. That is why it is also known as a Scripted Render Pipeline (SRP).

Unity also went to great lengths to make shader programming more accessible with URP by a tool called ShaderGraph which is essentially a visual scripting interface to shader programming. In terms of visuals and performance this render pipeline is able to target a vast array of platforms like smartphones, tablets or desktops. URP is

a good compromise with its balancing of above-average graphics quality and hardware performance requirements.

Other than that URP is supported by the ESRI Maps SDK for Unity which means it had to be considered for this project.

3. High Definition Render Pipeline (HDRP)

The most advanced render pipeline in Unity is the High Definition Render Pipeline (HDRP). It has some features in common with URP such as being a Scripted Render Pipeline and supporting ShaderGraph. But in general it is designed to create high end graphics that can leverage some of the latest features of today's GPUs like raytracing. But these high end capabilities come at a price because in terms of graphics hardware it is also the most demanding pipeline. It requires graphics hardware from the latest generation to work reasonably with HDRP. I found that it can be hard to get a decent frame rate with HDRP if the hardware is not powerful enough. But when the hardware is on par to match HDRP requirements it can be astonishing to look at the almost photorealistic results.

HDRP is also supported by the ESRI Maps SDK for Unity which makes it the right choice when striving for truly amazing, high end results.

Because I needed to include the ESRI Maps SDK for Unity in the project only two choices were left for selecting a render pipeline: URP and HDRP. But given that my hardware was not capable enough to work with HDRP I decided to go with URP because it supports "ESRI Maps SDK for Unity" and because the development hardware was solid enough to cope with the performance requirements for URP.

6.4.7. Unity goes cross-platform

Unity as a 3D development environment supports a wide range of platforms and devices. The idea of cross-platform is to develop an application once and deploy it to many different platforms like desktop, tablet, mobile and the like with only minor changes to the code base. In fact, it is amazing how many platforms are supported by Unity. The latest count on platforms that can be targeted with Unity exceeds 20 platforms with Windows, Mac and Linux as the most prominent desktop platforms and Android and iOS as the most widely used mobile platforms (UNITY 2022B).

That almost makes me tend to think of Unity as a kind of "3D content hub (including GIS artifacts)" that enables 3D content distribution to a vast variety of platforms. Even if the main platform for this project is Windows I felt tempted to prepare this project also for the Looking Glass holographic display (LG 2022A) or even the Microsoft HoloLens augmented reality device (MSFT 2022C) because both platforms are supported by Unity as well. But for now the Windows platform is the platform that I will focus on for this project. In general, once the project data are hosted by a Unity project this can be a kind of launchpad to bring the same content to other platforms without reinventing the wheel again.

6.5. ESRI ArcGIS Maps SDK for Unity

6.5.1. Overview

The ESRI ArcGIS Maps SDK for Unity enables connecting the ESRI and Unity ecosystems to leverage Unity's visualizing capabilities for a new type of spatial aware applications. Until ESRI had released its SDK Unity had no spatial awareness in terms of longitude and longitude let alone coordinate systems like WGS84. Of course Unity has its internal coordinate system working with X, Y and Z coordinates to enable placing of objects in a 3D space but these coordinates have nothing to do with any projected or geographic coordinate system.

That all changed with the release of ESRI's ArcGIS Maps SDK for Unity back in June 2022. Thanks to ESRI's Unity integration effort artifacts from ESRI can be brought to Unity without losing their spatial fidelity. This had not been possible in Unity before without having custom software. Some examples for ESRI artifacts are Basemaps or 3D Scene Layer Packages. As soon as ESRI artifacts are part of a Unity scene it is pretty easy to leverage Unity's capabilities like realistic materials, lighting, weather effects, animation, physics and so on. Perhaps it's best to let ESRI explain why real-time 3D engines like Unity might open the door to compelling new GIS applications. According to the ESRI presentation "Introduction to ArcGIS Maps SDK for Game Engines" held at 2021 ESRI Developer Summit this integration between the GIS and the real-time 3D domains enables (ESRI 2021A):

Next generation 3D GIS solutions

- *Access to real-world, geospatial data*
- *Photo-realistic 3D and AR/VR experiences*
- *Interactive and immersive*
- *Modeling and simulation*
- *Real-time analysis*

The same presentation also provides some insights on the rationale behind ESRI's decision to embrace real-time 3D engines (ESRI 2021B).

Why game engines?

- *Premium rendering experience*
- *Physics, animation, special effects*
- *Cross hardware development*
- *Existing developer community*

And I agree with this. Even if the abilities of the ESRI universe in terms of 3D visualization are already impressive but the visualization features of real-time 3D engines step up the game(!) by entering a whole new level...

And for the sake of completeness in terms of versions I have been working with ESRI Maps SDK for Unity version 1.0.

6.5.2. Key Features

There are quite a couple of features that come with the ESRI ArcGIS Maps SDK for Unity. A comprehensive feature overview can be found ESRI's ArcGIS Developers site (ESRI 2022b) so I will cover selected features only. Which features to introduce has been informed by the extent to which they have been used during the project.

As hinted at before the option to geolocate Unity's GameObjects in a precise manner can be considered a "game changer" (no pun intended) for real-time 3D engines. Closely coupled with this new capability the SDK also features local and global scenes based on geographic and projected coordinate systems.

Of course the SDK also contains options to leverage Unity's rich features for visualization. Given that interactivity is at the heart of a real-time 3D engines it was only natural to leverage these as well for interfacing with visualized spatial phenomena.

The SDK is also able to handle both online and offline data. Resources shared online as image tile or scene layers can be embedded into a scene but an additional option is to import locally stored packages such as image tile packages (*.tpk or *.tpkx files) or scene layer packages (*.slpk files).

There are also features that support spatial analysis within Unity but in comparison with ArcGIS Pro's rich analysis capabilities they only cover a subset of ArcGIS Pro's toolbox. But that may change in future versions of ESRI ArcGIS Maps SDK for Unity.

To conclude this overview of the SDK's feature set it is worth pointing out that there are two methods to choose from when working with the SDK. ESRI included the so called "Map Creator" in its SDK which is one way to work with the SDK. This "Map Creator" is an easy to grasp user interface that seamlessly blends in with Unity's native user interface. "Map Creator" is meant to make access to SDK features easier for non-programmers. This fancy interface allows for a no code approach while at the same time providing some guidance when working on a spatial aware scene.

But since the SDK is entirely based on components as discussed in a previous chapter scenes can be also built from scratch with C#. This is the second way of working with the SDK. It's also helpful that the SDK includes sample code to get familiar with this approach. This is also the route I took because of the following reasons: I just felt more in control when building out a scene completely code based than by entering values in a form.

And I also noticed some minor glitches when working with the "Map Creator". Perhaps it was my fault but sometimes I found that key parameters had been lost and I had to start over again. So I also hoped for a greater robustness when working with a code based approach. Just for illustration purposes a screenshot of the "Map Creator" interface is shown below.

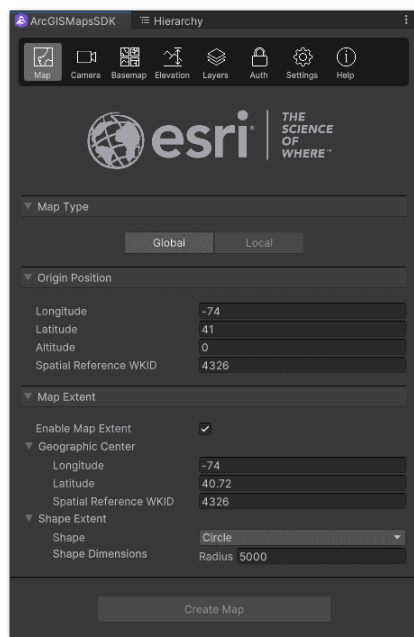


Figure 47: User interface of “Map Creator” in Unity Editor (Source: ESRI)

6.5.3. Maps

Of course maps are an indispensable means when working with spatial phenomena. But in SDK context a map or to be more precise the Unity component named `ArcGIS Map` refers to a kind of container that is holding map related information. To recap the notion of a component: A `Transform` component in Unity may hold information on a `GameObject`'s position, rotation and size. Something similar is true when speaking of an `ArcGIS Map` component which is holding information on the spatial context in a narrower sense. This information can be attached to a `GameObject` to give spatial awareness to any `GameObject`.

Some of the properties of the `ArcGIS Map` component include if a local or global scene is set up. It is also pivotal information to set the spatial origin of a given scene. The origin of a scene can be described by its longitude, latitude and elevation. In addition a spatial reference WKID is used to refer to a coordinate system. What is visible in the scene is also controlled by the extent definition of the `ArcGIS Map` component. The extent can be set by defining a rectangle or circle based on the previously set up map origin.

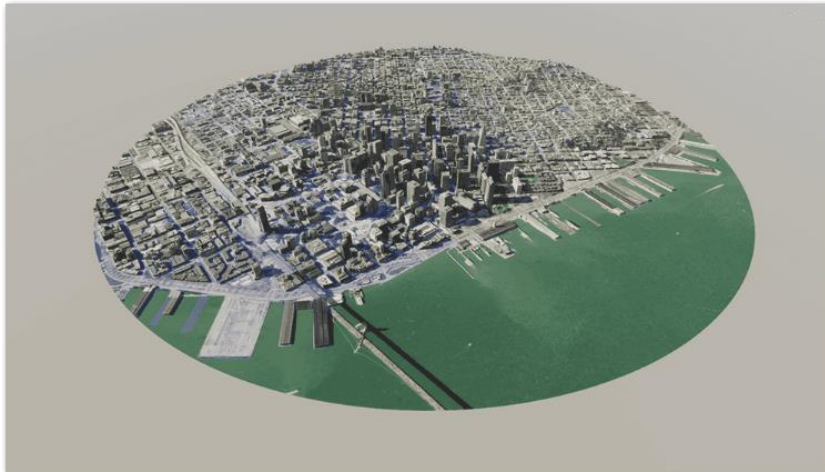


Figure 48: Local scene and circle extent enabled by ArcGIS Map component (Source: ESRI)

6.5.4. Layers

Similar to ArcGIS Pro layers play an important role in the SDK and they come in two flavours: There are basemap layers and data layers.

Basemap usage is currently limited to image tile layers. They can be pulled online by using an image tile service or by using a local image tile package. An additional option for setting up a basemap is to point to an URL that is containing the JSON file of a web map.

When it comes to data layers image tile layers are supported but also scene layers. Again image tile layers can be consumed online as an image tile service or as a locally stored tile package.

With scene layers there is a similar set of options. Supported are 3D object scene layers and integrated mesh scene layers when using an online connection. Their offline counterparts i.e. scene layer package files (*.slpk files) are supported as well.

6.5.5. Spatial and Data Analysis

There is also some tooling in place for geometries and analysis when working with the SDK. Although the SDK's toolset does not match ArcGIS Pro's level when compared by scope or variety it feels like a solid foundation to start with.

To work with geometries ESRI has a geometry engine wrapped into the SDK. This geometry engine supports a couple of well known geometries like Point, MultiPoint, PolyLine or Polygon. There are also dedicated methods in the SDK that enable building any of these geometries by running C# code.

For example this would allow to create custom importers for spatial data like GeoJSON files within Unity. Right now the SDK cannot cope with GeoJSON files but by leveraging the SDK's geometry engine there is a foundation already to build custom importers for spatial data.

Even if using analysis features had not been necessary for this type of digital twin project there are quite a few operations that can be run. Popular examples are the *Buffer*, *Clip* or *Union* operations.

In addition to that spatial relationships can be examined with *Contains*, *Intersects* or *Crosses* as some random examples. But with this section only touching upon some sample features a comprehensive list of operations that cover both geometry engine and spatial analysis can be found at ESRI's SDK documentation page (ESRI 2022c).

7. A Digital Twin of Portland Public Transport

This chapter sets out to examine the various software pieces that the TriMet digital twin is composed of. After taking a bird's eye view on the overall architecture I will take a closer look at the different software building blocks and how they are working together.

7.1. Architecture Overview

The following diagram may help to get an overall idea of this digital twin's architecture.

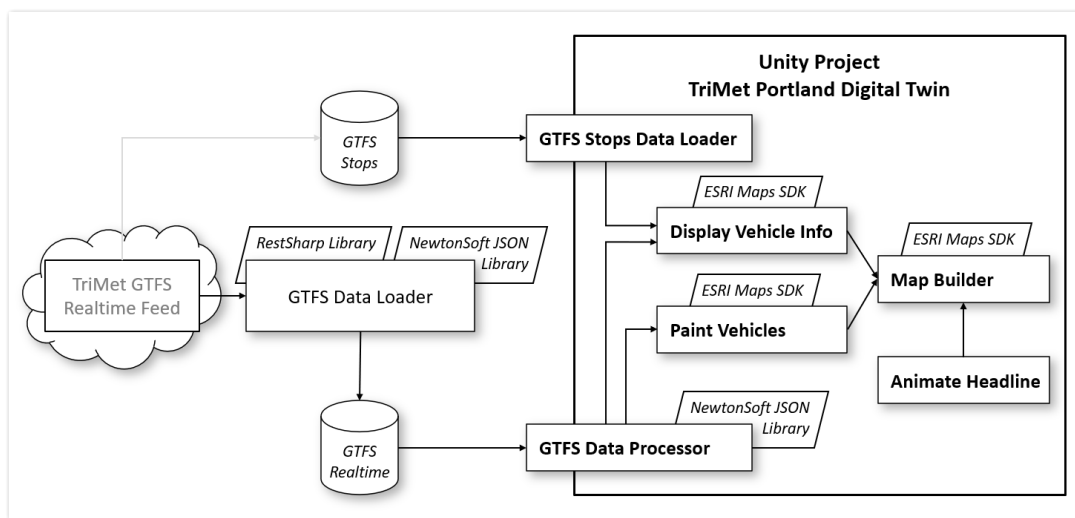


Figure 49: Digital twin application's architecture overview

Starting on the left of the diagram the GTFS feed is displayed as a cloud to highlight its web service nature. This "cloud" is the source of GTFS Realtime data. This is also the source of GTFS Schedule data i.e. the `stops.txt` file that I used for mapping stop ids to stop names. But because getting the `stops.txt` file has been a one off I tried to show this by a weaker connection between the "cloud" and the GTFS Stops symbol in the architecture diagram.

The "GTFS Data Loader" sitting next to the cloud represents the standalone Windows application that takes care of collecting data continuously. Loader collected data are stored in a local folder for further processing. The item marked GTFS Realtime is meant to represent this. Something similar is true for the `stops.txt` file which is also hosted by a local folder named GTFS Stops.

But the power station of this digital twin application is the Unity project which can be seen in the right half of the diagram. It contains a couple of scripts that take care of data management and visualization.

For one the Unity project is responsible for reading realtime data from the local folder that is continuously updated by the "GTFS Data Loader" app. This is done by the "GTFS Data Processor" part. There is also the "GTFS Stops Data Loader" part which loads the GTFS Schedule file `stops.txt` to prepare for later mapping. These data management tasks happen behind the scenes to have the data on hand for the upcoming visualizing tasks.

On the other hand there are a couple of scripts that take care of the visuals. The `Map Builder` script creates a map of the Portland TriMet region from scratch including routes and stops. Adding to that is the `Paint Vehicles` script which displays the location of vehicles on the map based on the previously fetched location data. There is also the `Display Vehicle Info` script that is used to create and update an info panel with more detailed information on a selected vehicle.

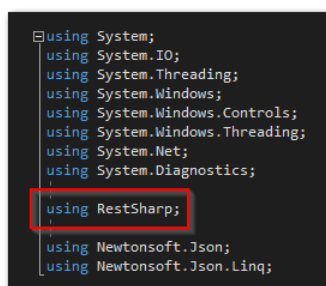
To top everything off a final script called `Animate Headline` creates a nice application title to give some context including but not limited to the local Portland time.

With that general overview in place now I will move on to check out everything in more detail. Perhaps it's best to begin with discussing supporting libraries like RestSharp and Newtonsoft.Json.NET. This may be a good starting point because they provide the foundation to have the app conveniently dealing with Web Services and working with JSON files.

7.1.1. RestSharp Library

Obviously one of the challenges in this project was to access the TriMet web services that carry vehicle location data. Since most of the work in this project is based on the C# programming language and its surrounding .NET ecosystem the RestSharp REST API client library came in handy to get this done. Being an open source project that is made available under an Apache 2.0 license it seemed a good choice to pick RestSharp for this project. According to the website of RestSharp it is one of the most popular libraries when it comes to dealing with RESTful interfaces in a .NET environment (Rs 2022A). It also supports accessing any API that is sitting on top of the http protocol. The library can be downloaded straight from RestSharp's website but it can be also easily embedded into the Visual Studio IDE by using the NuGet package manager. Its website also hosts plenty of documentation and sample code to get started with RestSharp.

Once the library is included by leveraging C#'s `using` directive many convenient high level methods for accessing and processing web services become available.

A screenshot of a code editor showing C# using directives. The code is as follows:

```
using System;
using System.IO;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;
using System.Net;
using System.Diagnostics;
using RestSharp;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
```

The line `using RestSharp;` is highlighted with a red rectangular box.

Figure 50: Leveraging the RestSharp library with C#

To highlight the library's ease of use I selected two code snippets grabbed from the GTFS Realtime Data Loader source code that takes care of building http requests. I bundled request prep in a method called `buildRequest()` and by using the RestSharp data types `RestRequest` and `RestResponse` it was only a matter of a few lines of code to build an http request to be fired later in the `getAndSaveVehicleData()` method.

```
private static void buildRequest()
{
    request = new RestRequest(triMetWebServiceUrl);
    response = new RestResponse();

    request.AddHeader("User-Agent", "ESRI Unity prototype app");
    request.AddHeader("Accept", "*/");
    request.AddHeader("Accept-Encoding", "gzip, deflate, br");
    request.AddHeader("Connection", "keep-alive");

    request.AddParameter("appID", "XXXXXXXXXXXX");
}
```

Figure 51: Building a request with the RestSharp library

```
//
// fire request to get vehicle positions from TriMet GTFS feed
//
try
{
    response = triMetRestClient.Get(request);
}
catch (Exception e)
{
    Console.WriteLine("Cannot access TriMet web services, exiting now...");
    Console.WriteLine("Exception: " + e.Message);
}
```

Figure 52: Running a request with the RestSharp library

7.1.2. Newtonsoft Json.NET Library

I also took advantage of the Json.NET library when working on the project. Json.NET is a popular open source library that comes with an generous MIT license. This means it can be used in a production environment without any license fees. Since I decided to use JSON as the primary data format when accessing the GTFS Realtime feed I considered additional tooling to be helpful when processing JSON files. Since Json.NET is a library that has been around for quite some years I also felt confident that I wouldn't encounter any surprises by using it.

Similar to the RestSharp library it can be easily included with any Visual Studio C# project by leveraging the NuGet package manager. Once the Json.NET library is included in the project it is just a matter of adding some `using` statements to enable using Json.NET's rich processing capabilities.

```
using System;
using System.IO;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;
using System.Net;
using System.Diagnostics;
...
using RestSharp;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
```

Figure 53: Leveraging the Json.NET library with C#

Selected methods of the library have been used to take care of serializing and deserializing JSON objects as shown in the next code snippet from the “GTFS Data Loader” application.

```
//  
// put prettily formatted JSON data in string formattedVehicleList  
//  
object vecicleList = JsonConvert.DeserializeObject(response.Content);  
string formattedVehicleList = JsonConvert.SerializeObject(vecicleList, Formatting.Indented);
```

Figure 54: The Json.NET library hard at work with serializing/deserializing objects

These have been the essential methods I worked with to avoid reinventing the wheel with JSON processing. Although this is only a tiny bit of Json.NET’s capabilities it was already sufficient to make JSON based development easier.

Some of the other features of Json.NET include its vast support of different .NET runtime environments but also translating between JSON and XML data formats. Again a comprehensive overview of Json.NET’s capabilities along with documentation and sample code can be found at its web presence (Ns 2022A).

Coming up next is a discussion of the GTFS Realtime Loader that sits on top of the libraries introduced in this section. The loader relies on their capabilities to access web services and to process JSON data.

7.2. GTFS Realtime Data Loader

As mentioned earlier this application takes care of retrieving GTFS Realtime data from TriMet’s developer site where the GTFS Realtime feed is hosted. I already touched upon the rationale why I fleshed out this app as a full blown Windows application which is to have a more convenient way of interacting with the GTFS Realtime feed. Perhaps the following screenshot helps to get a general feel for the application.

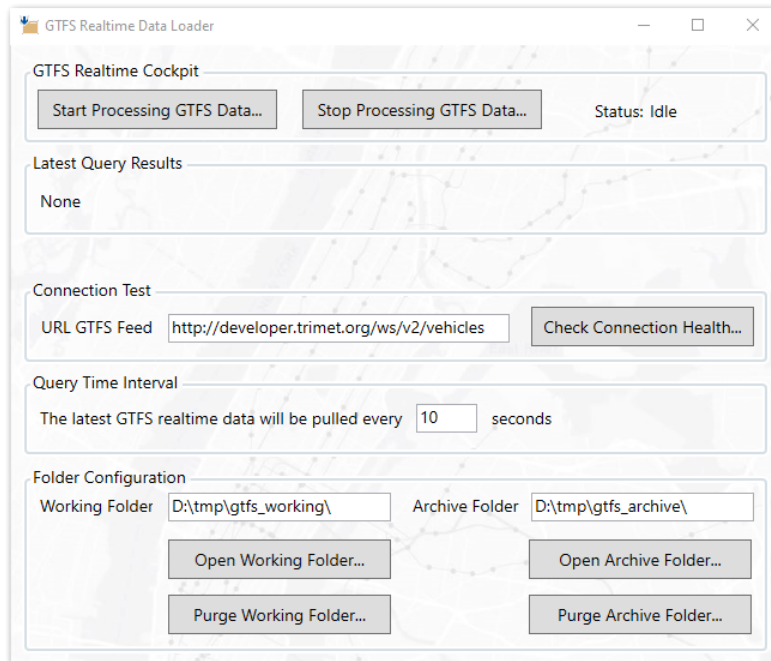


Figure 55: GTFS Data Loader overview

What this application does at its very core is getting the latest GTFS Realtime data from TriMet every 10 seconds and saving it as a JSON file.

But I also want to talk some more about what it's like to work with this app. First things first the application needs to be launched. I created a desktop shortcut to get this done easily.

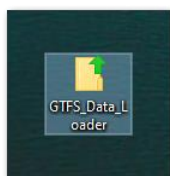


Figure 56: GTFS Data Loader desktop shortcut

Once the application has been launched it is sufficient to click the `Start Processing GTFS Data...` button to do just this. As soon as the button has been clicked the application starts collecting data in the background and its status in the UI switches from `Idle` to `Processing`. When first data are in the app does a quick summary on the data by updating the section `Latest Query Results`. This summary basically tells the number of vehicles in operation combined with date and time. All times are displayed as both Central European Time and local Portland time which happens to be Pacific Standard Time.

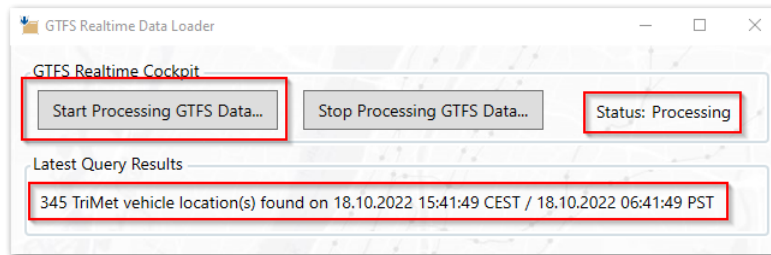


Figure 57: How to run the GTFS Data Loader

<OFF THE RECORD>

When I was in the middle of testing out the application I suddenly found that the feed always provided zero vehicle locations. I went into a state of shock and I thought “What is wrong here? This did already work the other day...”. And after a while I became aware that local time in Germany was something like 11.30 a.m. and of course Portland local time was 2.30 a.m. due to the different time zone. Doh! So Portland was sleeping peacefully and there were no vehicles on the road at all. Zero vehicles! And my application was working perfectly fine even if it didn’t look like that...

</OFF THE RECORD>

When thinking about reusability of the GTFS Data Loader I pondered it would be more flexible to expose the feed URL to the user interface. In theory this application could be also used to collect data from other feeds just by exchanging the URL in the field URL GTFS Feed.



Figure 58: GTFS Data Loader configurable feed URL

Although the GTFS Realtime feed turned out to be very reliable I wanted to have an option to do a quick check if there is a working internet connection and if the GTFS feed at TriMet’s backend is alive. This has been done by submitting a test request to the TriMet web services and checking the response for an http OK status. If an OK status has been found a simple Windows message box notifies that everything is fine, in case of error a warning will be issued.

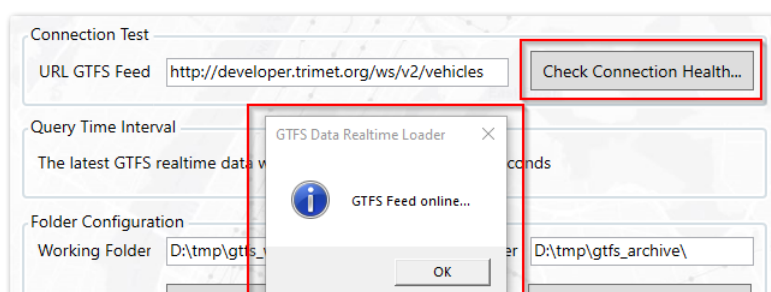


Figure 59: GTFS Data Loader connection health check

According to the GTFS Realtime specification the feed needs to make sure to carry updated vehicle location data every 60 seconds. With the TriMet feed in particular I found that updated

data could be retrieved as fast as every ten seconds. But again to be able to cope with other feeds that are not as frequently updated as TriMet's feed I decided to make the query frequency configurable. So the frequency in seconds is exposed to the user interface and open to modification.

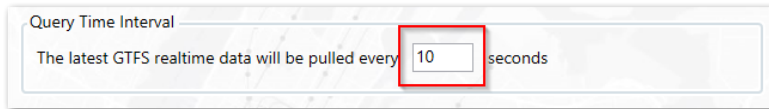


Figure 60: GTFS Data Loader configurable query frequency

The last section of the GTFS Data Loader's user interface deals with the local folder configuration. GTFS Data Loader saves all received files at two locations on disk. The first one is the `working` folder which happens to be `D:\tmp\gtfs_working\` in my environment. The second folder is at `D:\tmp\gtfs_archive\` and serves as a backup or reference folder in case of debugging. Looking a little bit ahead there is an independent thread within the Unity project that monitors the `working` folder and if this thread is spotting a new file it will be immediately processed. After processing has completed the file is moved to a folder called `D:\tmp\gtfs_processed\`. Because of files being moved around as part of the workflow I deemed it helpful to have a reference folder that contains all fetched data in one place which is the mentioned archive folder.

Because I did not want to run Windows Explorer each time when I wanted to examine folder contents I embedded a convenient option to open working and archive folders. This is done by clicking the buttons `Open Working Folder...` and `Open Archive Folder...` respectively. Another quality of life feature is the option to remove all folder contents with a simple click on buttons `Purge Working Folder...` and `Purge Archive Folder...`. While debugging the app I sometimes wanted to have a clean slate to start over with.

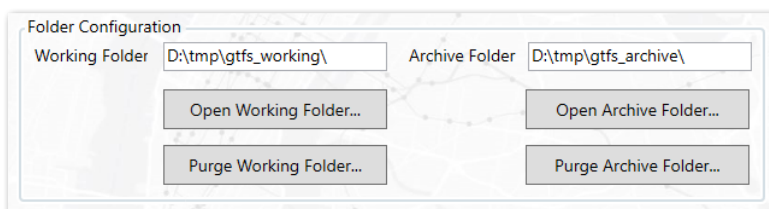


Figure 61: GTFS Data Loader folder configuration

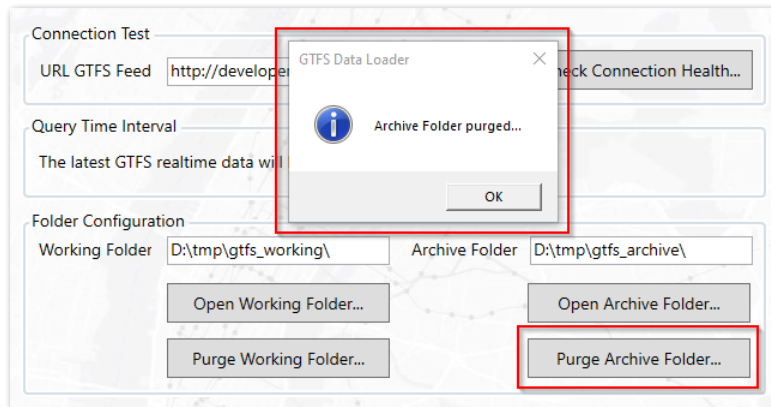


Figure 62: GTFS Data Loader folder cleanup

Perhaps it may be also interesting to take a look at the `working` folder to examine its contents.

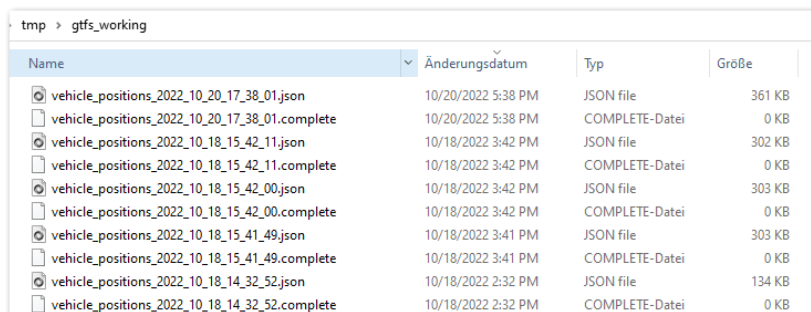


Figure 63: Sample contents of “working” folder

Two types of files can be found here. All files with a `*.json` extension contain feed data in JSON format. The second file type with a `*.complete` extension serves as a kind of semaphore file that is written each time a JSON file transmission has finished. I wanted to make sure that subsequent actors in the workflow would not access any halfbaked JSON file until the transmission had been completed for sure. The thread within the Unity project that is responsible for polling the working folder is on the lookout for any new files with a `*.complete` file extension. When any such file is found the thread considers it safe to process the JSON file with the same base filename. This is how I tried to ensure that no JSON files are processed that are incomplete because they have been in the midst of a transmission.

Finally I went for a file naming convention that encodes date and time of data transmission into the filename. I pondered that this would make it easier to keep track of incoming data and have it sorted by time.

After having discussed the user interface and also how to work with the application it may be appropriate to discuss some of the inner workings of the application now.

So in the next section will try to outline in rough strokes what the code in `MainWindow.xaml.cs` is about. Maybe it's best to start with the user interfacing side of the application, too. When working on a XAML based Windows project all UI elements fire events that can be captured to trigger actions. If, for example, a button is clicked this click event is captured and the method is executed that is linked to this click event. Since there a quite a few

events happening in the app the following table contains all methods that are related to any events caused by interaction with the UI.

UI Event related methods in <i>MainWindow.xaml.cs</i>		
StartProcessingGTFSData_Click()	1	Method called when button „Start Processing GTFS Data...” is clicked
StopProcessingGTFSData_Click()	2	Method called when button „Stop Processing GTFS Data...” is clicked
ConnectionCheck_Click()	3	Method called when button „Check Connection Health...” is clicked
QueryInterval_Changed()	4	Method called when text field „Query Time Interval...” has changed
WorkingFolder_Changed()	5	Method called when text field „Working Folder” has changed
ArchiveFolder_Changed()	6	Method called when text field „Archive Folder” has changed
OpenWorkingFolder_Click()	7	Method called when button „Open Working Folder...” is clicked
OpenArchiveFolder_Click()	8	Method called when button „Open Archive Folder...” is clicked
PurgeWorkingFolder_Click()	9	Method called when button „Purge Working Folder...” is clicked
PurgeArchiveFolder_Click()	10	Method called when button „Purge Archive Folder...” is clicked

Table 11: UI Event related methods in *MainWindow.xaml.cs*

The numbers in the table above refer to the UI elements as shown in the following screenshot of the “GTFS Data Loader” app.

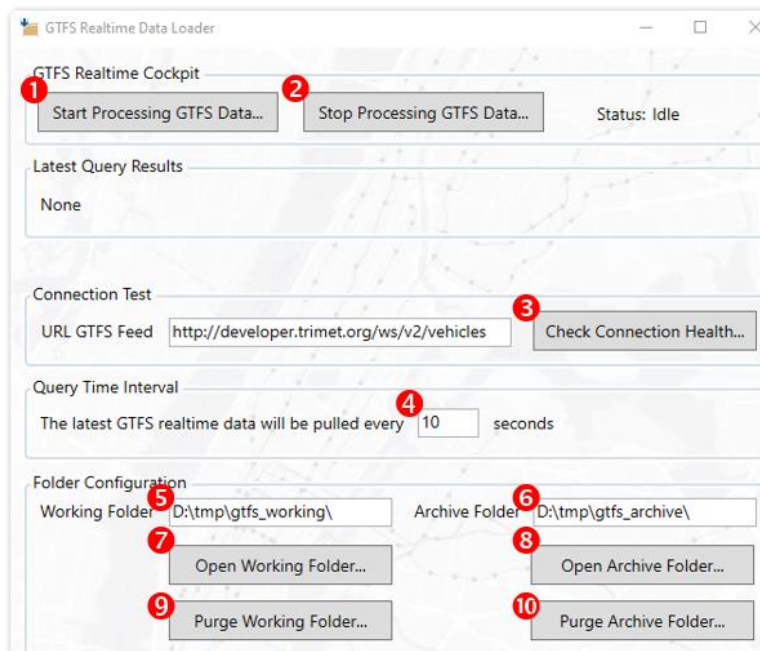


Figure 64: UI elements of GTFS Data Loader with events attached to them

With the UI related events summarized I will continue with giving an overview of the methods that actually do the heavy lifting under the hood.

General methods used in <i>MainWindow.xaml.cs</i>	
<code>MainWindow()</code>	Roughly comparable to C's <code>main()</code> function <code>MainWindow()</code> is the program entry point when building a WPF application with C#
<code>initializeRestClient()</code>	Sets up the REST client based on the RestSharp library
<code>buildRequest()</code>	Takes care of building the REST request
<code>getAndSaveVehicleData()</code>	Overarching method to bundle everything – receives GTFS JSON data and saves them to local disk
<code>getNumberOfVehicles()</code>	Get number of vehicles in operation to display them in “Latest Query Results” section
<code>getQueryTimeStamp()</code>	Reads time stamp from GTFS JSON data
<code>epochTimeToPacificTime()</code>	Converts UNIX epoch time to Pacific time
<code>localTimeToPacificTime()</code>	Converts local time to Pacific time
<code>buildTimeStampForJsonFile()</code>	Build string with time stamp to be included in file name of local GTFS JSON file
<code>writeGtfsFileToWorkingFolder()</code>	Write GTFS JSON file to local working folder
<code>writeGtfsFileToArchiveFolder()</code>	Write GTFS JSON file to local archive folder
<code>writeGtfsCompleteFileToWorkingFolder()</code>	Write *.complete file to indicate that writing of GTFS JSON file has been completed
<code>Timer_Tick()</code>	Event triggered by timer. It is called every x seconds depending on the configured time interval
<code>printVehicleProperties()</code>	Debug method to output vehicle properties to console

Table 12: GTFS Data Loader method overview

With this overview in place I guess the time is right to discuss how things work in general with GTFS Data Loader. Obviously there are a couple of event related methods that are used to bind certain actions to certain events.

With a quick recap of the user interface two major events are wired into the application. These two events are linked to the buttons `Start Processing GTFS Data...` and `Stop Processing GTFS Data...`. As soon as a user hits `Start Processing GTFS Data...` a REST client is set up to prepare for calling the GTFS Realtime web service. A timer is set up too to manage the recurring web service calls.

The timer plays a crucial part when accessing the web service. Closely associated with the timer is a method called `Timer_Tick()` which can be used to link whatever method needs to be executed on a regular basis. This is because the `Timer_Tick()` method is called every x seconds depending on the timer settings. Here the `Timer_Tick()` method calls a core method named `getAndSaveVehicleData()` that is run every 10 seconds by default. This method takes care of building the web request and running it against the GTFS Realtime web service. Once the web service call has been successfully completed the JSON response is parsed, formatted and saved to local folders. Other than that this method takes care of computing the number of vehicles in operation and displays them in the UI section “Latest Query Results”. This data collection happens continuously as long until the user hits the `Stop Processing GTFS Data...` button to finish the data collection. Clicking this button means that the timer is killed and the `Timer_Tick()` method won't be called any more.

The following diagram might help to get an overall impression on all methods working together. But please note that for the sake of clarity I slightly modified some method names.

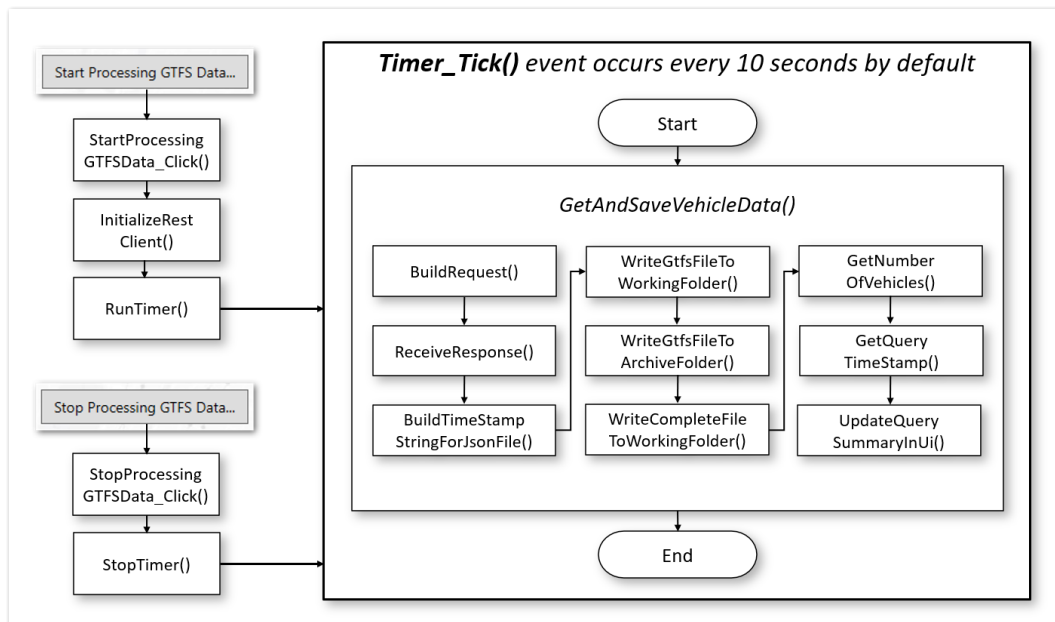


Figure 65: The inner workings of GTFS Data Loader

Although the diagram above outlines the program flow in general there also a couple of “helper” methods that are responsible for transforming time related artifacts. Due to Portland’s location being in a different time zone a transformation from Central European Time (CET) to Pacific Standard Time (PST) was required. This has been taken care of by method `localTimeToPacificTime()`. And since time related data as part of the GTFS Realtime feed had been provided in Unix Epoch Time there was also the need to have a transformation for this. A method named `epochTimeToPacificTime()` did the trick here.

The complete source code for this application can be found in appendix section A. *GTFS_Data_Loader*. While the file `MainWindow.xaml.cs` contains C# code to define the application’s behaviour the file `MainWindow.xaml` describes its user interface in XAML notation.

7.3. GTFS Realtime Data Simulator

Obviously a precondition for accessing a GTFS Realtime feed over the internet is to have a working internet connection in place. When sitting at the office desk under usual circumstances there is nothing to worry about. But I also imagined that I might want to show this digital twin in environments with poor or no internet access at all. In this case I wanted the application to have offline capabilities that would allow for showcasing the digital twin even without having live data. So the idea of a GTFS Realtime Data Simulator developed when thinking about this...

Since this simulator is only meant as a fallback in the rare situation of having no internet connection I kept the programming effort limited and fleshed this out as a plain console application.

To prepare data for simulation I captured data from the GTFS Realtime feed and set them aside in a dedicated folder. What the simulator then basically does is to copy files from the

simulation folder to the working folder every ten seconds thus mimicking the behaviour of the GTFS Data Loader from the previous section. Again the source code can be found in appendix B. *GTFS_Data_Simulator*.

7.4. Unity Project TriMet_Portland_Digital_Twin

This Unity project can be seen as the “engine room” of the Digital Twin application. It is its pounding heart that brings the digital twin to life. But to get a better idea what a Unity project looks like it might help to have a glimpse into the user interface of Unity.

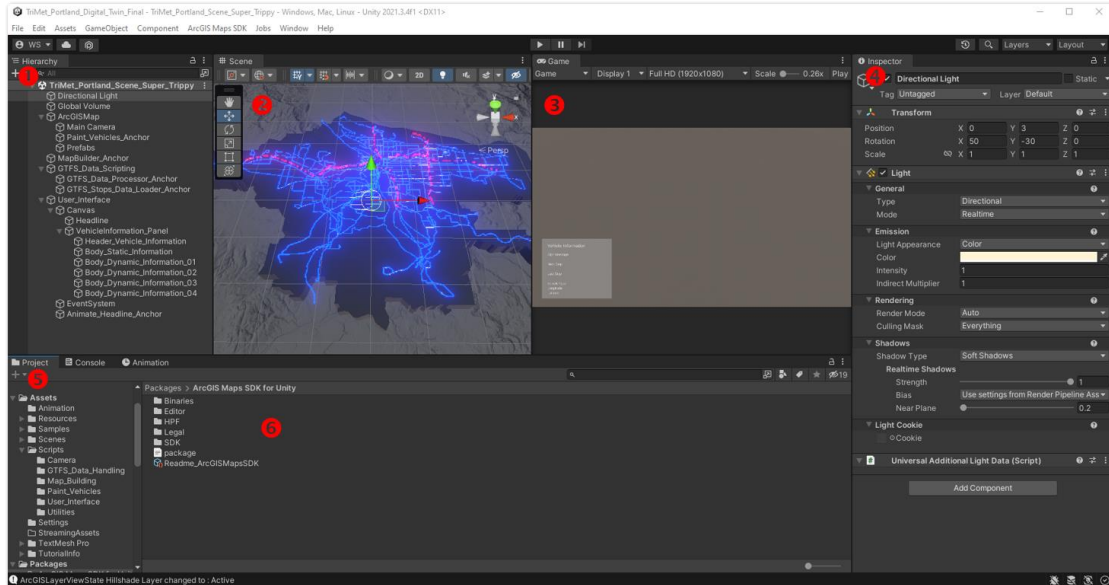


Figure 66: Working with the Unity Editor (Source: Own picture)

Because the user interface of the Unity Editor can be a little bit overwhelming at first I will try to dissect its different areas.

Unity Editor Views		
Hierarchy View	1	The Hierarchy View contains all elements a Unity scene consists of
Scene View	2	The Scene View is meant for building out the scene i.e. set up the camera, take care of the lighting, placing objects, assign materials
Game View	3	The Game View allows previewing the scene at runtime. It looks the same as a finished application with all scripts executing
Inspector View	4	The Inspector View allows for viewing and modifying the components of GameObjects that the scene is composed of
Project View 1	5	The Project View enables an Explorer like look at a Unity project and all of its files
Project View 2	6	Again similar to Windows Explorer in this view project files can be examined in more detail

Table 13: Unity Editor views

With a general breakdown of the UI now settled I will focus more specifically on the editor’s project view. The project view in the lower half of the Editor is very similar to the Windows Explorer pattern when working with files and folders. And it is good practice after starting a new Unity project to create a default set of folders within the project. Some of the folders that are usually created include Scenes, Scripts, Animation, Resources and so on.

And this is what I did to keep the Unity project organized. I set up a couple of folders to hold the different project elements which are also generally known as “Assets” in Unity speak.

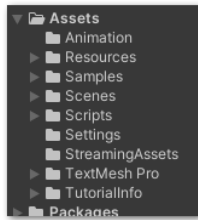


Figure 67: Folder layout in Unity project

Obviously the Scripts folder holds all C# scripts and I refined the file organization some more by creating different subfolders. This enabled me to group scripts by topics.

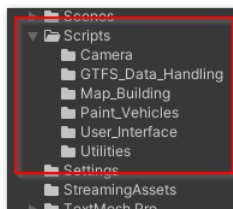


Figure 68: Subfolders in Scripts folder

The following folder layout made sense to me so I grouped the scripts as described in the table below.

Script Subfolders in Unity Project <i>TriMet_Portland_Digital_Twin</i>	
Camera	CustomArcGISCameraControllerComponent.cs This script takes care of the scene camera and allows to zoom and pan the camera within the scene
GTFS_Data_Handling	GTFS_Data_Processor.cs This script reads and shares incoming GTFS data as provided by the GTFS_Data_Loader app GTFS_Stops_Data_Loader.cs This scripts reads the stops.txt file to have stop names ready for mapping to stop ids
Map_Building	Portland_Map_Builder_Regular.cs This script creates the map that is displayed in the scene
Paint_Vehicles	Paint_Vehicles_Regular.cs This script is responsible for displaying vehicle locations on the map
User_Interface	Animate_Headline_Regular.cs This script takes care of displaying a scene title Display_Vehicle_Information.cs This script creates and updates an info panel that contains details on a selected vehicle Escape_Key_Handler.cs This script checks if the Escape key has been pressed for exiting the application
Utilities	VehicleIndex.cs This script is a helper script to be able to index vehicles for easier access

Table 14: Script subfolders in Unity project

With the `Scripts` folder examined in more detail I will also cover the remaining project folders. Again a table might help to get an overview.

Project folders in Unity Project <i>TriMet_Portland_Digital_Twin</i>	
Animation	This folder contains “animations”. Animations in Unity are everything that changes a <code>GameObject</code> ’s properties over time. In this project animations have been used to change the light intensity of the dots representing the vehicles on the map.
Resources	By convention a folder that is used during runtime needs to be called <code>Resources</code> . It had been used to load materials for <code>GameObjects</code> during runtime.
Scenes	This folder name is rather self-explanatory. It contains Unity scenes which are saved in a Unity native file format.
StreamingAssets	This folder is a Unity default folder that can contain any files that are needed at runtime. It has been used to host the Scene Layer Packages from ArcGIS Pro.
TextMesh Pro	TextMesh Pro is the name of Unity’s library to help with user interfaces including text. It has been used to prepare the scene title and the info panel.

Table 15: Project folders in Unity project

With the folder layout of the Unity project out of the way I will move on by examining the scene hierarchy.

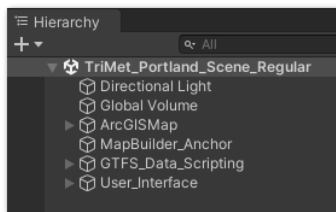


Figure 69: Scene hierarchy in Unity project

On top of the scene hierarchy the scene name can be found which happens to be `TriMet_Portland_Scene_Regular` in this screenshot.

All other entities in this scene are `GameObjects` with Components attached. `GameObjects` can be renamed to have a more meaningful name. It is also possible to have empty but properly named `GameObjects` to get the scene more organized. This is helpful when it comes to hiding complexity in a scene. In the next screenshot for example `User_Interface` is an empty `GameObject` which as a parent holds quite a few child `GameObjects` dealing with UI related tasks.

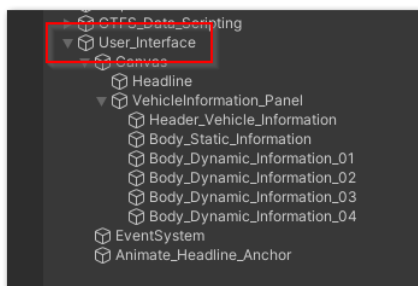


Figure 70: Usage of empty `GameObject` in Unity project

I also developed the habit to name `GameObjects` that have a `C#` script attached to them with the suffix `_Anchor`. If I see something that is named like this in a scene I know at once that there is a script attached to this `GameObject`.

With all those basics settled now I will try to sort the hierarchy one by one.

The first two `GameObjects` `Directional Light` and `Global Volume` are related to scene lighting. The `Directional Light` can be considered the “sun” of the scene that takes care of flooding the scene with light. The `Global Volume` can be used for light effects that affect the whole scene such as fog or bloom. Volumes are a feature of URP and HDRP but are not present in the Built-in Render Pipeline.

The next `GameObject` named `ArcGISMap` is already something that comes from the ArcGIS Maps SDK for Unity. Any `GameObject` that is meant to use features of the SDK needs to be a child of this `ArcGISMap` `GameObject` because it inherits essential information like longitude, latitude, altitude and a spatial reference WKID from the parent.

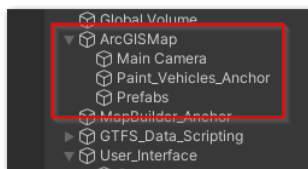


Figure 71: `ArcGISMap` `GameObject` and its children

Obviously the `ArcGISMap` `GameObject` has three children. The first one is the scene camera that determines what is shown in Game View. Unlike any regular Unity project this special “GIS” camera has coordinates in terms of longitude, latitude and altitude to define its position. Considering how the Unity coordinate system works for any other non-ESRI project this is no small feat.

The second one is an empty `GameObject` that holds a reference to the `C#` script that is responsible for painting the vehicle “dots” on the map. Since it also works with longitude/latitude data it needs to be a child `GameObject` of `ArcGISMap` too.

The last `GameObject` found here is called `Prefabs`. For now a prefab can be considered a special type of `GameObject` that is derived from a kind of `GameObject` template. Although it is also possible to build a sophisticated `GameObject` from scratch just by coding in `C#` it is much easier to build a complicated `GameObject` once, set it aside as a prefab and, if needed, just instantiate the `GameObject` from this previously prepared prefab. I guess the notion of a “game object factory” comes pretty close to how prefabs can be understood. Every time many `GameObjects` of the same type are needed in a scene it’s worth thinking about using prefabs.

And this is what I did when it came to bringing dots i.e. vehicle positions to the screen. I used prefabs for visualizing the vehicle positions and at runtime all these instantiated prefabs are attached to the empty `Prefabs` `GameObject`. This is important because with `Prefabs` being a child of `ArcGISMap` there is a spatial reference present which is needed to place them at the right spot.

Coming up next is the `MapBuilder_Anchor`. Its only purpose is to host the `Portland_Map_Builder_Regular.cs` script. This script will be discussed in one of the next sections.

There is another `GameObject` in the scene called `GTFS_Data_Scripting`. This is also an empty `GameObject` serving as a kind of container to bundle everything related to GTFS data processing. It has two children named `GTFS_Data_Processor_Anchor` and `GTFS_Stops_Data_Loader_Anchor`. Again both `GameObjects` have the two scripts `GTFS_Data_Processor.cs` and `GTFS_Stops_Data_Loader.cs` attached to them respectively. These scripts will be also discussed while moving forward.

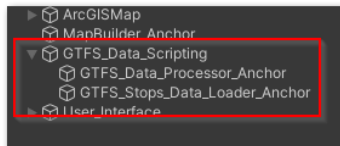


Figure 72: `GameObjects` responsible for GTFS data processing

Last but not least the `GameObject` `User_Interface`, as the name implies, does focus on the user interfacing side of the scene.

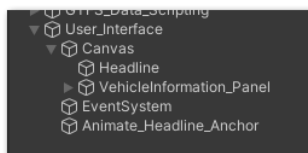


Figure 73: `User_Interface` `GameObject` and its children

It is responsible for painting the scene title but also builds the info panel in the lower left of the scene that is populated with vehicle info once a vehicle is selected.

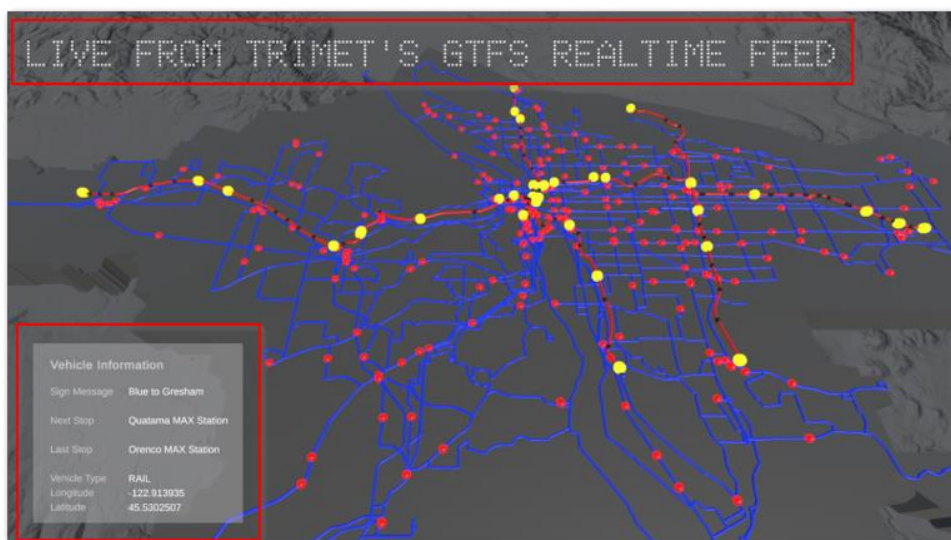


Figure 74: Scene headline and info panel (Source: Own picture)

Now that the most important parts of the digital twin scene in Unity have been discussed I will dive deeper into the scripts that do the work behind the scenes (no pun intended).

Since we already discussed the `GTFS_Data_Loader` standalone application outside of the Unity project it may make sense to look into how Unity does process the data that

GTFS_Data_Loader has collected beforehand. This is what the script `GTFS_Data_Processor.cs` is tasked with: Processing the collected GTFS data...

7.4.1. GTFS Data Processor

The script `GTFS_Data_Processor.cs` takes care of processing GTFS Realtime data that have been acquired earlier by the `GTFS_Data_Loader` application. It does so by monitoring the folder `D:\tmp\gtfs_working\` for any new files. As soon as a new GTFS Realtime file is found the file gets processed and its data are shared Unity project-wide by a public `Vehicle List`. I also set up a class named `Vehicle` to have a convenient data type to build the `List` with. Instances of this class can hold all information associated with a vehicle.

I made a conscious decision to implement this data processing task as an independent thread within the Unity project. Although there are methods within Unity such as `StartCoroutine()` or `InvokeRepeating()` that are also meant to perform tasks in parallel their downside is that they are competing with the rendering thread for computing resources which can lead to hiccups in terms of frame rate.

I already discussed the pivotal role of the `Update()` method in any Unity related C# script. This method is called each time before a new frame is rendered to the screen. Another default method provided by Unity's engine is `Start()` which is called once and only once after a scene has been loaded. It can be seen as a kind of `autoexec.bat` file from the MS-DOS days of old within a Unity scene.

I used this `Start()` method to get the data processing thread going. There is also a method called `OnApplicationQuit()` that can be used to call any method as seen fit when leaving the application. By using this method I killed the data processing thread when the digital twin gets closed. That means the data processing thread is run as soon as the scene is loaded and terminates when the application is left.

Another key mechanism is how the folder `D:\tmp\gtfs_working\` is observed to detect any new GTFS files. More specifically the thread is on the lookout for any new files with a `*.complete` extension. Those files indicate when a file has been successfully copied to the working folder. To detect any new `*.complete` file I leverage a class called `FileSystemWatcher` which is part of the library `System.IO`. It can be configured in a way that an `OnCreated()` event is triggered each time a new `*.complete` file has been created. I used this `OnCreated()` event to trigger processing the file. In fact the only method that is called from within the `OnCreated()` method is called `processGtfsFiles()`. This method is the workhorse that takes care of all things GTFS file processing. The next diagram might help to get a feel for the chain of methods that are executed by `processGtfsFiles()`.

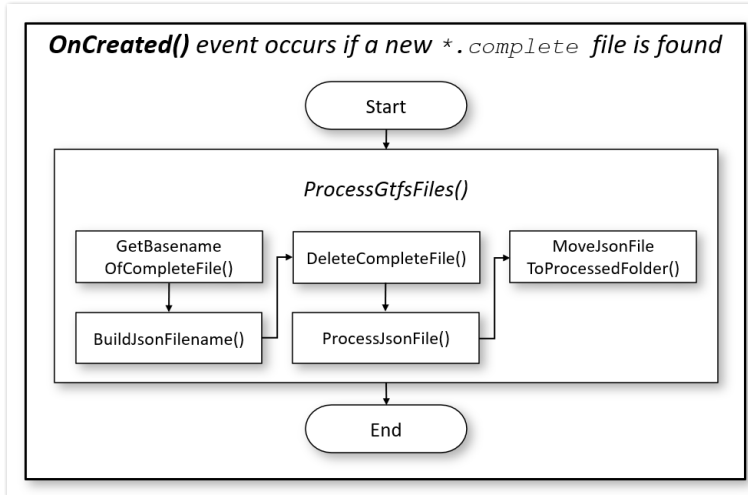


Figure 75: The inner workings of ProcessGtfsFiles()

Since the `processGtfsFiles()` method relies on a couple of submethods again I will try to explain them one by one in what they're doing. First of all I try to get the basename of the found `*.complete` file. That means if the file is called `someName.complete` I want to extract the `someName` part of the filename. After that this basename part is used to build the filename that needs to be read which is `someName.json`. Since the `*.complete` file is only needed to indicate the arrival of a fully transmitted file it is deleted afterwards because it has served its purpose. Coming up next is processing the contents of the JSON file. The file `someName.json` is read and all its JSON contents is assigned to a List of type `Vehicle` that holds all vehicle information in a structured fashion. This List is dynamically built out of `Vehicle` objects each time a new JSON file is read. That means at any point in time there is a vehicle List containing the current vehicles in operation along with all vehicle properties including latitude/longitude. Since this is a public list any other script in the Unity project can access the complete vehicle data at any time. The following picture might help to grasp the idea.

```

static public List<Vehicle> vehicleList = new List<Vehicle>();

public class Vehicle
{
    public string expires      { get; set; }
    public string signMessage  { get; set; }
    public string serviceDate  { get; set; }
    public string loadPercentage { get; set; }
    public string latitude     { get; set; }
    public string nextStopSeq  { get; set; }
    ...
    // many more vehicle properties...
    ...
    public string source       { get; set; }
}
  
```

Figure 76: The Vehicle List as a means of sharing vehicle data across the Unity project

Once the Vehicle List has been updated with the latest vehicle information the file `someName.json` will be moved from the `working` folder to the `processed` folder. It is no longer needed for processing but is kept in the `processed` folder for reference.

Taking a step back might help to get the bigger picture of script `GTFS_Data_Processor.cs`. Maybe it could be summarized like this: This script makes sure that at any point in time the latest vehicle information is available across the entire Unity project.

The C# source code of `GTFS_Data_Processor.cs` can be found in appendix C. *Unity Project TriMet_Portland_Digital_Twin*

7.4.2. GTFS Stops Data Loader

In section 4.2.2 *TriMet Stops Data* I already pointed out that GTFS Realtime data does not carry any stop names as plain text. What the feed does provide though is stop ids in numerical form. But with GTFS Realtime data alone it wouldn't be possible to resolve any stop id to a stop name. Luckily there is a file `stops.txt` as part of the GTFS Schedule spec that can be used for mapping stop ids to stop names.

So the GTFS Stops Data Loader script is responsible for reading the `stops.txt` file from disk and putting it in a C# Dictionary that can be accessed by other scripts of the Unity project. The core method of this script is called `getStopName()` which browses the dictionary for any given stop id and returns the stop name in plain text.

```
public static string getStopName(int stopId)
{
    return gtfsStopsDictionary[stopId];
}
```

Figure 77: Resolving stop ids to stop names

This service is leveraged when the vehicle info panel gets populated with stop names.

The C# source code of `GTFS_Stops_Data_Loader.cs` can be found in appendix C. *Unity Project TriMet_Portland_Digital_Twin*

7.4.3. Map Builder

This section sets out to discuss the script that deals with painting the map onto the Unity scene. Its name is `Portland_Map_Builder_Regular.cs`. This script borrows heavily from sample code provided by ESRI as part of the Maps SDK for Unity. In fact it is a customized version of ESRI's code to match the particular needs of the Portland TriMet network digital twin.

ESRI provides plenty of tutorials and code samples to make it easier for any developer wanting to deep dive into the Maps SDK for Unity. I found one of the tutorials especially helpful because it was about building out a map by using the C# API of the SDK. ESRI's sample code is also well documented which helped a lot to ease me in when learning the ropes of the SDK. Because the ESRI tutorial covers all the steps needed to create a complete map based Unity scene it might not be necessary to discuss every line of code of

`Portland_Map_Builder_Regular.cs` because a comprehensive explanation can be found at ESRI's developer site (ESRI 2022D). Most of the code hasn't been changed at all. I only injected changes on an as needed basis. So instead of discussing the whole code I will focus on the code parts that have been modified.

As always in a Unity scene a camera position needs to be set up. In a default Unity project the coordinate system would have its origin at an imagined position somewhere in the scene. All coordinates would be relative to this made up origin. But here, thanks to ESRI's plugin, the camera is positioned with longitude, latitude and elevation data. A spatial reference is provided as well. The data type `ArcGISPoint` from the SDK is the foundation to position `GameObjects` like the scene camera precisely.

```
private ArcGISPoint cameraCoordinates =
    new ArcGISPoint(-122.679775, 45.057815, 84750.982019,
        ArcGISSpatialReference.WGS84());
```

The camera's longitude/latitude data describe a position in the midst of the Portland area although there is some skew because the camera looks at Portland from above with a certain angle. The camera elevation is in fact about 85 kilometers above the ground which seems about right when looking at the scene.

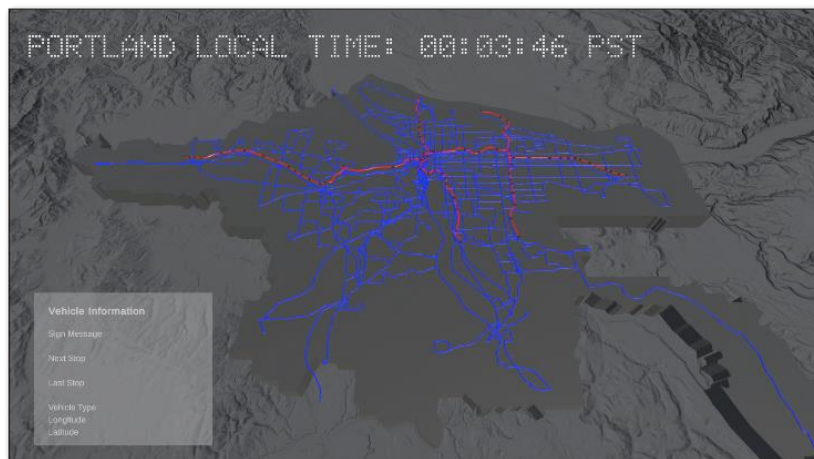


Figure 78: Portland area as seen by the scene camera (Source: Own picture)

The Portland area of interest's origin is also defined by leveraging the `ArcGISPoint` data type.

```
private ArcGISPoint originCoordinates =
    new ArcGISPoint(-122.6883204, 45.4639347, 0,
        ArcGISSpatialReference.WGS84());
```

The extent of the map and its center can be set up by using the following code snippets which basically creates a window into the greater Portland area that is 400 by 400 kilometers.

```
ArcGISPoint extentCenter = new ArcGISPoint
    (-122.713204, 45.4639347, 0, ArcGISSpatialReference.WGS84());

ArcGISExtentRectangle extent = new ArcGISExtentRectangle
    (extentCenter, 400000, 400000);
```

With the basic scene and camera set up the scene now needs refining by using layers. I included an elevation layer that can be pulled from ESRI's online resources. Again the modified code snippet might help to get the idea.

```
arcGISMap.Elevation = new ArcGISMapElevation
    (new ArcGISImageElevationSource
        ("https://elevation3d.arcgis.com/arcgis/rest/services/
        WorldElevation3D/Terrain3D/ImageServer",
        "Elevation", ""));
```

It basically reads the elevation layer provided by ESRI and assigns the name `Elevation` to it.

I also embedded a hillshaded basemap layer into the scene to give it a more interesting texture. To me that felt more appealing visually than having an imagery layer which would have been pretty flat and un-3D. This layer is given the name `Hillshade Layer`.

```
ArcGISImageLayer hillshadeLayer = new ArcGISImageLayer
    ("https://ibasemaps-api.arcgis.com/arcgis/rest/services/
    Elevation/World_Hillshade_Dark/MapServer/", "Hillshade Layer",
    1.0f, true, APIKey)
```

In an earlier step described in *4.3 TriMet GIS Data* I prepared scene layer package files or `*.slpk` files from TriMet's shape files. These `*.slpk` files will be also embedded into the scene. The following code fragment shows how this can be achieved.

```
string tmBoundariesSlpkPath =
    Path.Combine(Application.streamingAssetsPath, "tm_boundary.slpk");

ArcGIS3DObjectSceneLayer tmBoundariesLayer = new
    ArcGIS3DObjectSceneLayer(tmBoundariesSlpkPath,
    "TriMet Boundary Layer", 1.0f, true, "");
```

These statements are responsible for loading the local `*.slpk` file named `TriMet Boundary Layer`. To change the appearance of the layer a material has been assigned to the layer. Just to recap a material is used to control how a `GameObject` is rendered in Unity with color being the most obvious material property. The following code takes care of giving the layer a dark grey color. The material is loaded at runtime from a special Unity folder within the project named `Resources`.

```
tmBoundariesLayer.MaterialReference = new Material
    (Resources.Load<Material>("Materials/DarkGreyMat_Regular"));
```

The same pattern is used to load all remaining `*.slpk` files such as `tm_routes_ALL.slpk`, `tm_routes_MAX.slpk` and `tm_stops_MAX.slpk`. I consciously omitted `tm_stops_ALL.slpk` from the scene because there are way too many stops in the TriMet service area. I pondered that so many stops would cause visual confusion and hurt the scene's clarity.

To make it easier to identify the different layers in the scene, the `tm_boundary` layer has a dark grey color assigned to it, the `tm_routes_ALL` layer is shown in blue, the `tm_routes_MAX` layer is the red one and the `tm_stops_MAX` layer is colored black. But I admit that the black stops might be a little bit hard to spot in the scene.

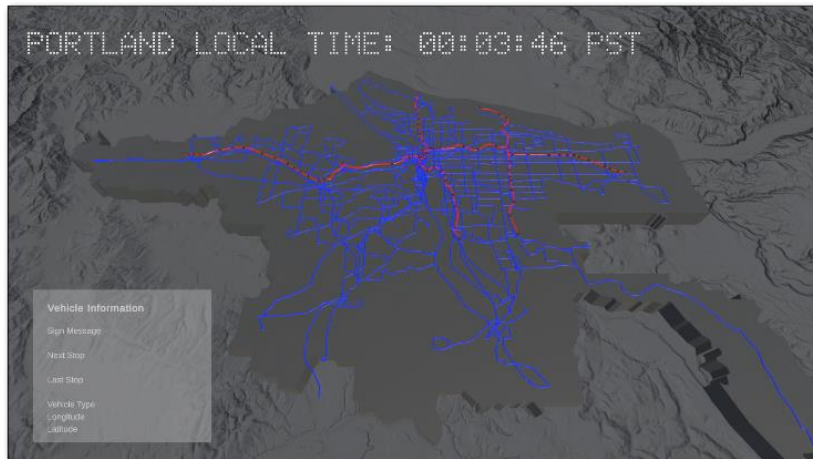


Figure 79: The Portland scene and its coloured layers (Source: Own picture)

To summarize all this I will fast forward through the entire script `Portland_Map_Builder_Regular.cs` once again. The scene camera is set up and the map is created along with center, extent, basemap and elevation. Different layers based on offline *.slpk files are loaded into the scene and have a material assigned to them. The material takes care of coloring the different layers. The scene as is it can be seen in the previous figure is now ready to serve as a backdrop where all vehicle positions can be visualized against.

The map builder's complete source can be found at appendix section *D. Customized ESRI Code*.

7.4.4. Camera Controller

This section describes how the controller of the scene camera gets set up. The camera controller is the scene component that captures user input and translates this input into camera movements. The primary input device to interact with the digital twin scene is the computer mouse.

For example by scrolling the mouse wheel it's possible to zoom in and out of the Portland area. The script also supports panning the camera to the right or left by moving the mouse with the left mouse button pressed. These basic functions to interact with the scene come already with the ESRI provided script `ArcGISCameraControllerComponent.cs`. Similar to the previous script this had been provided by ESRI as a sample script to learn from. A general discussion on cameras and how to tweak them can be found at ESRI's developer documentation (ESRI 2022E).

I also wanted to have certain restrictions on movements when navigating the scene. This is why I used this ESRI script as a template and built a custom camera controller tailored to the Portland area. This custom script is called `CustomArcGISCameraControllerComponent.cs`.

One of the restrictions I had in mind was to prevent the camera from zooming out ad infinitum. It made no sense to me to be able to zoom out of the scene outside of a

reasonable range. So I limited the range in which zooming out is allowed. The following screenshot shows the maximum zoom out permitted by the custom camera controller.

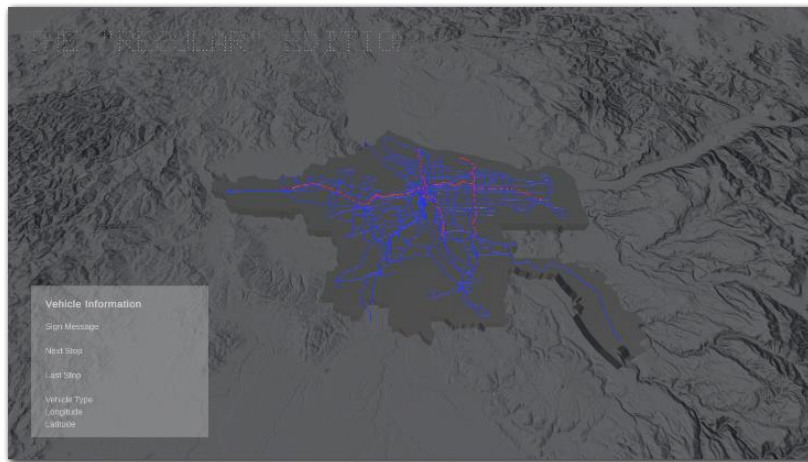


Figure 80: Zooming out limited by camera controller (Source: Own picture)

I also felt that limiting the range in which the camera is able to pan would lead to a better user experience. I didn't want the camera to leave the Portland area so I made the camera controller recognize this and stop panning at a certain point. The following pictures show how panning is restricted in different directions.

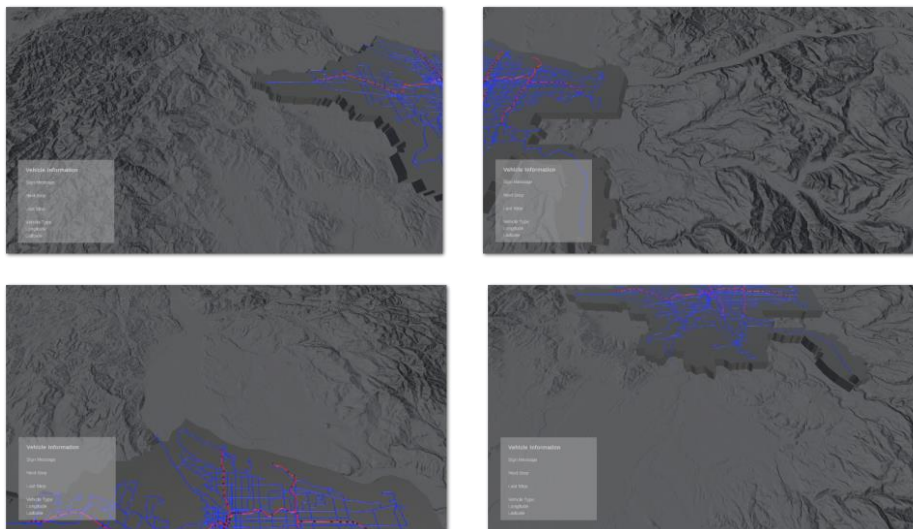


Figure 81: Panning restrictions by camera controller (Source: Own picture)

The last change I did to the camera controller was to disable camera rotation. By default the ESRI sample camera controller allows to rotate the camera by moving the mouse with the right mouse button pressed. But this didn't seem like a good fit for a local scene as the Portland area. It would make perfect sense to have camera rotation enabled on a global scene but with this project I pondered this would be confusing.

The challenge with tweaking the camera controller was to identify the code pieces that needed to be customized. Thanks to the well documented ESRI sample code this could be done with a medium effort.

To start with the suppressed camera rotation this could be achieved by commenting out the code that is responsible for rotating the camera.

```
if (IsMouseRightClicked())
{
    if (!deltaMouse.Equals(Vector3.zero))
    {
        // It's not desirable to have map rotation enabled

        // RotateAround(ref cartesianPosition, ref cartesianRotation,
        // deltaMouse);
    }
}
```

Limiting the panning abilities of the camera had to be tested first by working interactively with the scene to get the coordinates for limiting the pan. Then another code section needed to be extended to make this work.

```
// It's not desirable to have map panning
// beyond reasonable values

if (cartesianPosition.x > -13718136.5779012 &&
    cartesianPosition.x < -13591647.8562627 &&
    cartesianPosition.z < 5700730.93491859 &&
    cartesianPosition.z > 5623991.4877625)
{
    Position = cartesianPosition;
}
```

Finally the code piece needed to be found that could restrict zooming out of the scene. The code needed to be extended a little bit to reflect the maximum camera height allowed. The next code snippet shows how the camera height can be limited at 80000 meters. As soon as the Y coordinate of the camera exceeds 80000 it is reset to 80000 again.

```
if (cameraPosition.y > 80000.0)
{
    cameraPosition.x = savedCameraPosition.x;
    cameraPosition.z = savedCameraPosition.z;

    cameraPosition.y = 80000.0;

    savedCameraPosition = cameraPosition;
}
else
{
    savedCameraPosition = cameraPosition;
}
```

Although it took some effort to find the right spots in the camera controller code I finally got it working as imagined.

So this has been the customized camera controller script whose source code can be found in the appendix at section *D. Customized ESRI Code*.

7.4.5. Paint Vehicles

This section discusses the script that takes care of painting the vehicle locations onto the scene. Each light rail vehicle is visualized as a yellow sphere, all other vehicles like buses are shown as a red sphere.

In a previous chapter I discussed how I prepared a C# List of type `Vehicle` that is continuously updated to always have the latest vehicle information available. The `Paint_Vehicles_Regular.cs` script does access this vehicle list continuously to extract each vehicle's longitude and latitude properties from the list.

Since the source Vehicle list is updated every ten seconds by default the same time interval is used in this script to fetch the latest data from the list. I used the Unity builtin method `InvokeRepeating()` to get this done.

```
InvokeRepeating("paintVehicles", 0, queryInterval);
```

The invoked `paintVehicles` method is called every 10 seconds depending on the value of `queryInterval`. At the core of this method is a loop that walks through the `Vehicle List`, extracts longitude/latitude from each vehicle in the list, checks if a light rail or bus vehicle is processed and depending on the outcome yellow or red sphere prefabs are instantiated.

I already discussed the notion of prefabs in Unity earlier with pointing out that prefabs are a kind of template `GameObject` that can be easily instantiated without taking care of any of its components because they are already part of the template. And this is what I leveraged to paint vehicle symbols on the screen. I prepared two types of prefabs upfront, a somewhat larger yellow sphere and a red sphere. At the beginning of the script these two prefabs are preloaded into the scene but not yet instantiated.

The `paintVehicles` script also takes care of making the prefabs i.e. spheres children of the empty `GameObject` called `Prefabs` once they are instantiated. The `Prefabs GameObject` itself in turn is a child of `GameObject ArcGISMap`.

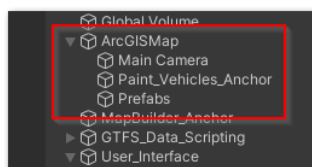


Figure 82: Prefabs `GameObject` as a child of `ArcGISMap`

This is rather important because by being a child of the `ArcGISMap GameObject` its spatial reference is inherited. This is a precondition when it comes to positioning the yellow or red spheres on the map exactly. Positioning itself is enabled by data type

ArcGISPoint as part of the ESRI SDK. But ArcGISPoint does work only when the ArcGISMap GameObject can be found among its parents. The next code snippet shows how the vehicle's location data is used to prepare the variable `vehiclePosition`.

```
vehiclePosition = new ArcGISPoint(vehicleLongitude, vehicleLatitude,
    4000, new ArcGISSpatialReference(4326));
```

What's happening next is that the values from `vehiclePosition` are fed into the prefab's ArcGISLocationComponent.

```
ArcGISLocationComponent locationComponent =
    prefabs[i].GetComponent<ArcGISLocationComponent>();

locationComponent.Position = vehiclePosition;
```

Speaking of the ArcGISLocationComponent this is finally the prefab's component that takes care of painting the vehicle symbols to the screen by using their longitude/latitude data.

To put a long story short: The script `Paint_Vehicles_Regular.cs` sets up a forever loop that is responsible for getting the latest vehicle positions, for instantiating the prefabs and for placing them on the screen at a precise location grounded with longitude/latitude data.

The complete source code of `Paint_Vehicles_Regular.cs` can be explored in appendix section C. *Unity Project TriMet_Portland_Digital_Twin*.

7.4.6. Display Vehicle Information

This script deals with the user interface and manages the vehicle information panel in the lower left of the scene. It is updated each time the mouse pointer hovers over one of the spheres that are representing vehicle locations.

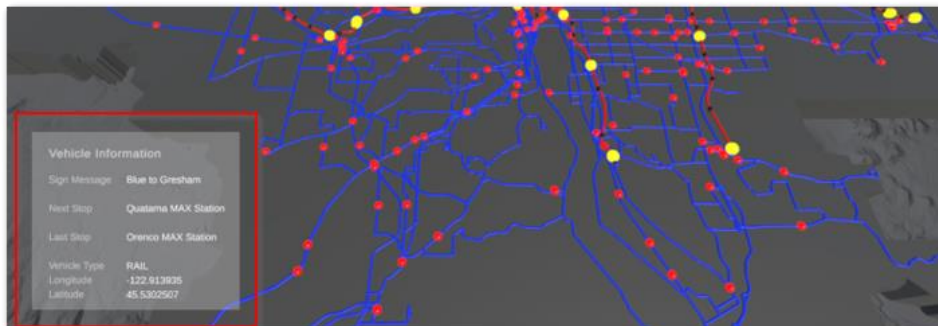


Figure 83: Scene info panel (Source: Own picture)

It is important to note that this script is one of the components the vehicle prefabs are made of. Each of the red and yellow spheres in the scene above represents an instantiated prefab that has a script component `Display_Vehicle_Information.cs`.

To get the info panel updated with vehicle specific information each time when the mouse pointer hovers over a particular vehicle I leverage two events that are built into the Unity engine. Those events are named `OnMouseOver()` and `OnMouseExit()`. As

their names already tell these events are triggered by the engine when the mouse pointer hovers over a point or leaves the hovering area of a point. These two events come in handy to capture if the mouse pointer is hovering over a vehicle or not. Specifically the `OnMouseOver()` event is used to populate the panel with data of the selected vehicle while the `OnMouseExit()` event is used to clear the info panel to make room for the data of the next vehicle that might be selected.

Thanks to the Vehicle List that is provided by script `GTFS_Data_Processor.cs` all information on any given vehicle is instantly available for populating the info panel. A code snippet that shows how vehicle type and longitude/latitude information can be retrieved from the list is shown below.

```
textDisplayMessage04.text =  
    GTFS_Data_Processor.vehicleList[vehicleIndex].type.ToUpper() +  
    "<br>" +  
    GTFS_Data_Processor.vehicleList[vehicleIndex].longitude +  
    "<br>" +  
    GTFS_Data_Processor.vehicleList[vehicleIndex].latitude;
```

The highlighted section of the info panel below shows the panel part that is populated by this piece of code.

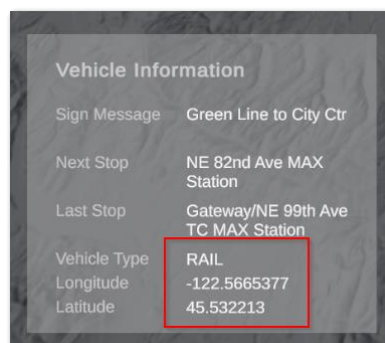


Figure 84: Populating the info panel

As mentioned before the user interface including the info panel is built by using the Unity package TextMesh Pro. This is the de facto standard with Unity to deal with text elements in a scene. It has rich text formatting capabilities that cover most of the needs when building a text based interface. Since TextMesh Pro is a well known staple when working with text information in Unity I will just refer to Unity's TextMesh Pro documentation for all the details (UNITY 2022c).

But one thing worth considering is how the 3D scene and the 2D text based interface live together within a 3D scene. I'd like to introduce a key element with TextMesh Pro called "canvas". This canvas can be imagined as a kind of transparent film sitting in front of the scene camera. All UI elements are placed on this transparent object that has a fixed position in front of the camera. So when the scene camera moves the position of all UI elements in front of the camera stays the same. This might help to get the picture how 2D elements do coexist with 3D objects in a scene.

Source code of `Display_Vehicle_Information.cs` can be reviewed in appendix section C. *Unity Project TriMet_Portland_Digital_Twin*.

7.4.7. Animate Headline

This is the final piece of script that manages the scene title at the top of the scene.



Figure 85: Changing the scene title (Source: Own picture)

Displaying the scene title is also based on the TextMesh Pro package. In fact there are four titles that are displayed in a rotating fashion. The first of the four titles fades in, is displayed for eight seconds and fades out again. Then the first title is replaced by the second title and the fade in/out sequence starts again. As soon as the fourth and last title has been displayed the scripts continues with displaying the first title again. This is done as long as the scene is running.

The four titles that are displayed one after the other endlessly are:

The TriMet Network in Portland
Live from TriMet's GTFS Realtime Feed
The "Vanilla" Edition OR The "Trippy Edition" (depending on the version)
Portland Local Time <PacificDateTime>

The `Update()` section of this script that is called each time before a frame is rendered does contain a timer. It checks if eight seconds have already passed. If so the script switches to the next title and the timer is reset. This is how I got the scene title steadily changing in a kind of billboard manner.

Picking the typeface from Unity's font library was pretty much my personal taste because I liked the matrix like feel of the font but the fade in and out effect has been achieved by a method provided by Unity's Shader Utilities. The following piece of code shows how this had been done.

```
dilationValue = messageMaterial.GetFloat(ShaderUtilities.ID_FaceDilate);
dilationValue += 0.18f * Time.deltaTime;
messageMaterial.SetFloat(ShaderUtilities.ID_FaceDilate, dilationValue);
```

The Shader Utilities contain a field named `ID_FaceDilate` that can be used to change the appearance of the type face. It can be made slimmer or fatter depending on the dilation value. The snippet above takes care of the title fading in so the dilation value gets greater each time the `Update()` method is called. This is done as long until a dilation value threshold is reached. Fading out the title works very similar but instead of increasing the dilation value it is decreased.

Perhaps the `Time.deltaTime` variable that is multiplied with the 0.18 fixed value needs some further discussion. The `Time` class is provided by the engine and it offers a way to measure the time between rendering two frames at runtime. This is what `deltaTime` contains. If the engine is running at 60 fps the value of `deltaTime` would be roughly 0.017. If the engine would be only capable of 30 fps due to a complicated scene `deltaTime` would be 0.033. By multiplying the `Time.deltaTime` value with another value which needs to be constantly increased/decreased Unity manages to get the dilation effect almost independent from frame rates. This ensures that movements in a scene feel the same no matter what the current frame rate is.

The full source code of `Animate_Headline_Regular.cs` can be found in appendix section C. *Unity Project TriMet_Portland_Digital_Twin*.

7.4.8. Vehicle Index

This is just a piece of “glue” code that helps with identifying vehicles based on an index number. The script `Vehicle_Index.cs` is an integral part of the Prefab that is responsible for displaying vehicles on the screen. This way each of the Prefabs can be easily identified by an index.

Source code of `Vehicle_Index.cs` can be found in appendix section C. *Unity Project TriMet_Portland_Digital_Twin*.

7.4.9. Escape Key Handler

This is another tiny bit of code helping with exiting the application gracefully. At its core this script is on the lookout for a pressed Escape key on the keyboard. If this happens the application will be closed.

The source of `Escape_Key_Handler.cs` can be found again in appendix section C. *Unity Project TriMet_Portland_Digital_Twin*.

8. Portland Public Transport Traffic Live – Digital Twin Concepts Applied...

8.1. Overview

In this section I will try to give an impression of the Digital Twin application representing the movements of TriMet vehicles across the Portland public transport network. Of course the best way to experience this Digital Twin is by setting up the complete application because the interactive features and the visuals let this Digital Twin really shine. Please refer to the GitHub section for instructions on how to get the Digital Twin installed.

8.2. „Vanilla“ Edition vs. „Trippy“ Edition

I have built out this Digital Twin in two versions: The first one is nicknamed the “Vanilla” edition and sports more unobtrusive visuals. The second one, dubbed the “Trippy” edition, plays with neon like visual effects and is somewhat more daring in terms of visuals.

Both versions are almost identical except their visuals. It almost boils down to having different materials in place and bits of animation to get the “glow and fade” effect working. But at its core both versions are the same.

This is also an explanation for certain file names that might be confusing at first sight. For example I referred to the script `Portland_Map_Builder_Regular.cs` in a previous chapter. This script is responsible for building out the “Vanilla” map. The script that takes care of building the “Trippy” map is called `Portland_Map_Builder_Trippy.cs`. This is how I tried to separate the visuals for each version.

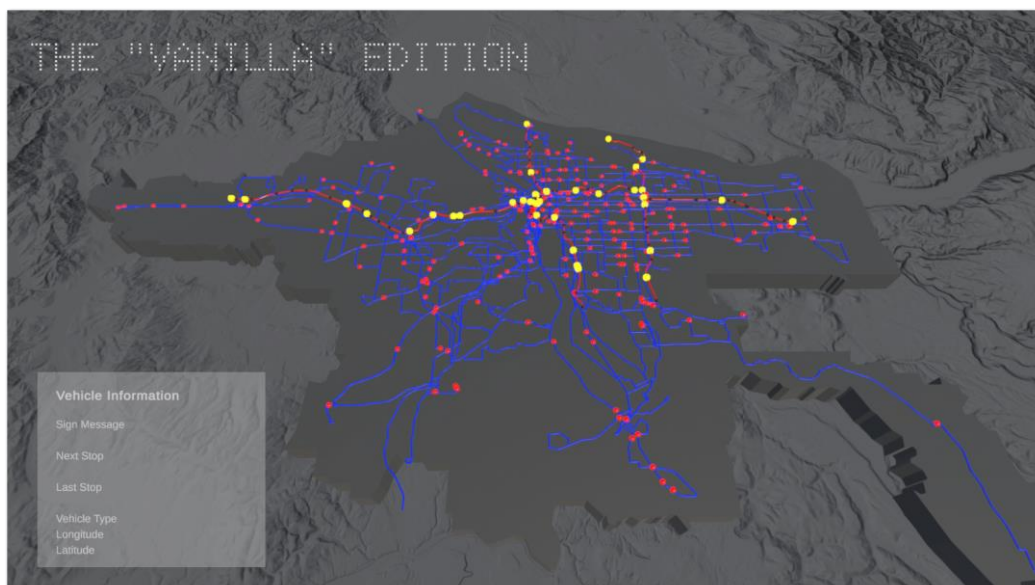


Figure 86: Digital Twin “Vanilla” Edition (Source: Own picture)

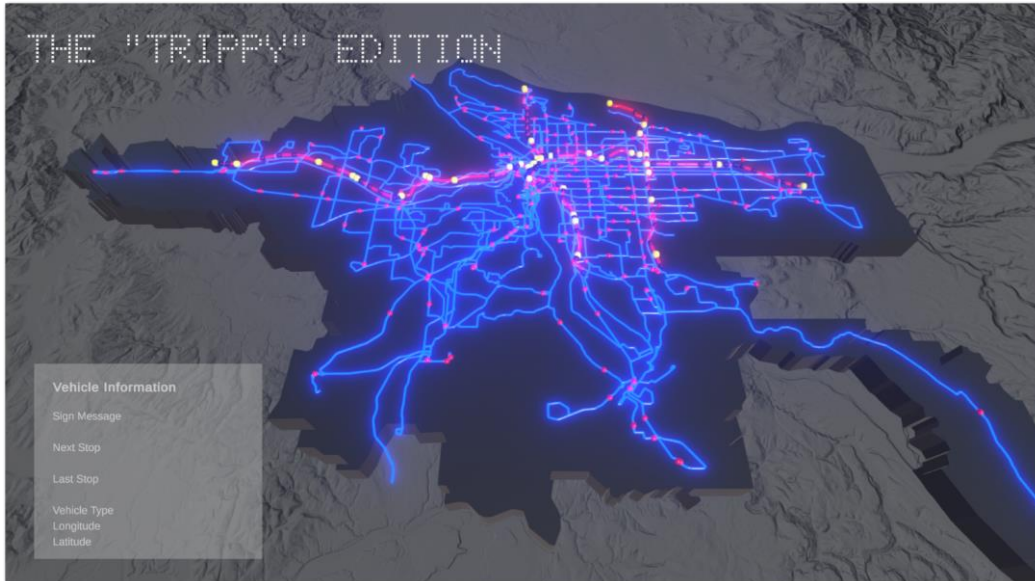


Figure 87: Digital Twin "Trippy" Edition (Source: Own picture)

8.3. Screenshots

In this section there are two more screenshots from both versions. They have been captured early in the morning at about 10.00 am. The vehicle count at this time had been in the range of 300 vehicles that had been operational and moved across the screen.

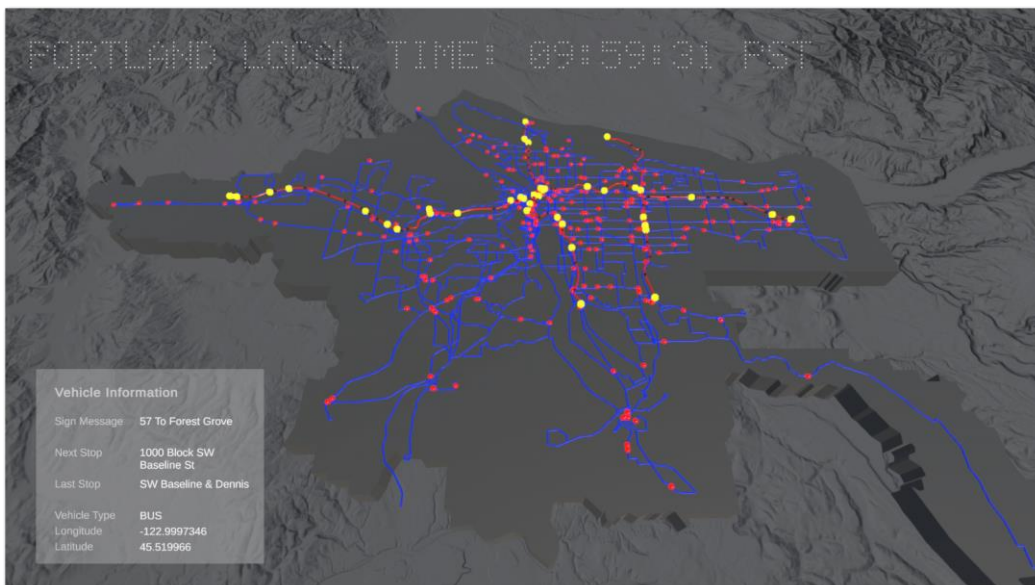


Figure 88: Random screenshot Digital Twin "Vanilla" Edition (Source: Own picture)

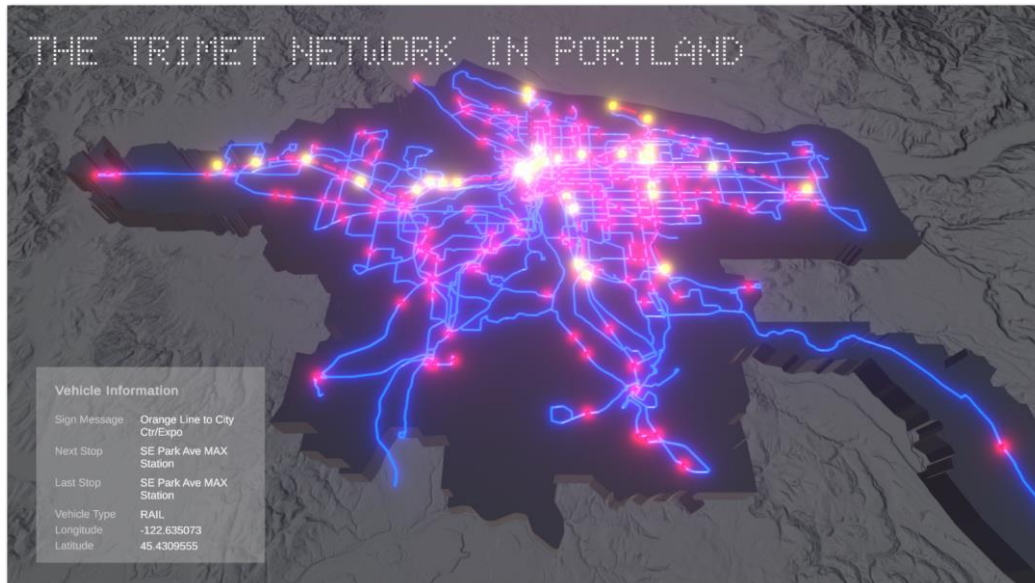


Figure 89: Random screenshot Digital Twin “Trippy” Edition (Source: Own picture)

8.4. Videos

It’s difficult to showcase this Digital Twin in a medium like a Microsoft Word document. At least screenshots might give an idea how this Digital Twin looks like. But a more appropriate medium for an interactive application like that is a video. Zooming in and out of the scene, panning the camera, selecting vehicles and exploring the info panel: All these interactions cannot be shown by screenshots. The same is true for watching the vehicle movements that happen all the time or the “glow and fade” effect of the vehicles.

So I created two videos to bring this Digital Twin to life some more. The first video captures the feel of the “Vanilla” edition (YOUTUBE 2022A), the second one does focus on the “Trippy” edition (YOUTUBE 2022B).

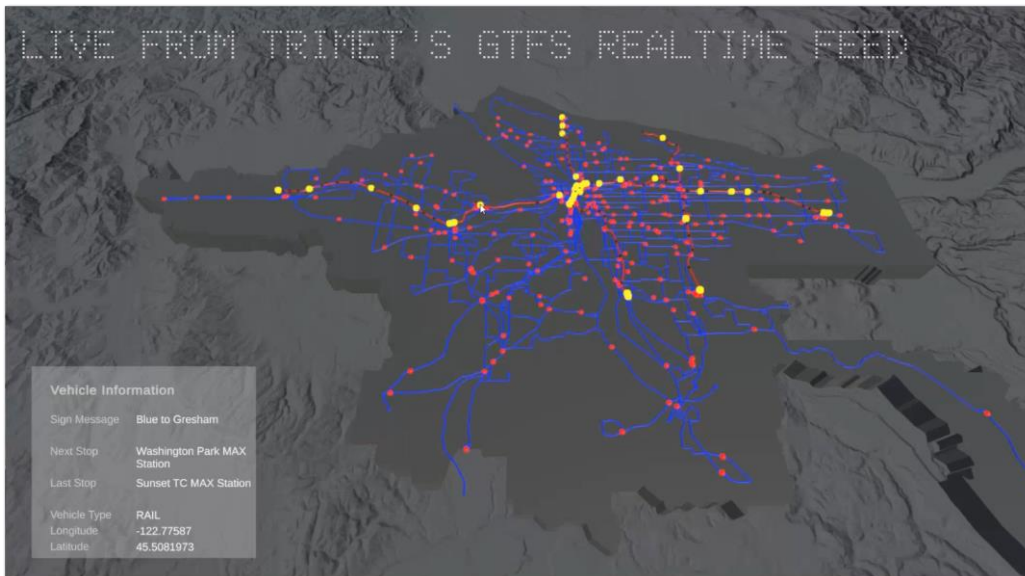


Figure 90: Screenshot from video Digital Twin “Vanilla” Edition (Source: Own picture)

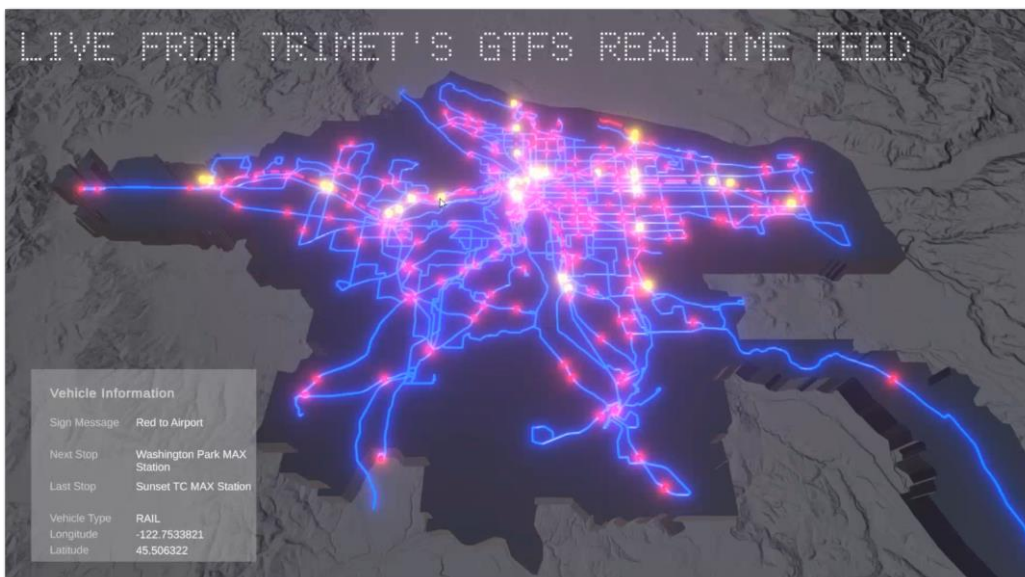


Figure 91: Screenshot from video Digital Twin “Trippy” Edition (Source: Own picture)

8.5. Into the Vault...

Although the appendix contains the complete source code of this Digital Twin I also shared the source code on GitHub. Besides the source code there are also executables available in the `Binaries` folder that should be ready to use. The GitHub repository can be found at https://github.com/fussgaenger/Portland_Digital_Twin (GITHUB 2022A).

9. Lookout Mountain...

You may wonder why this chapter is headlined “Lookout Mountain”. I called it like that because I felt this could be a nice metaphor. A nice metaphor because it takes quite some effort to climb a mountain which is also true for working on this project. But once the summit is reached the reward is to have a splendid view from the top. And with this technical journey coming to a close I also thought that looking from the top would allow me to reflect somewhat on this endeavour. I admit I was also fond of the powerful connotations of the term “Lookout Mountain” because this is also the name of a film studio that had been operated by U.S. Air Force to work on secret film projects. The studio was called “Lookout Mountain Film Laboratory” at the time and has a rich history that would deserve to be revisited just for film history reasons. But for now just take a look at the film credits that had been used throughout their movies in the 50s of the last century. The credits are somewhat ambiguous but they also have a unique feel somehow...



Figure 92: Movie credits from Lookout Mountain Laboratory (Source: Wikipedia)

Another distant relative to “Lookout Mountain Studios” is the term “skunkworks project”. While Skunkworks is also the official brand of American aircraft company Lockheed Martin for their most advanced and/or secret projects sometimes this term is colloquially used to refer to any project that is done just for the sake of passion and innovation. And this is what I did most of the time when working on this project. I wanted to get this as good as possible but as always time was a scarce resource. So many ideas had to be put aside to get the project finished in time. But at least some of the ideas will be put in this closing chapter in order to prevent them from being forgotten.

Movie history shenanigans aside I want to start with one thing that had room for improvement. At least this is what I felt. In the ArcGIS Pro section 6.2.2 *Workflows* I described a workflow to build *.slpk files for later import into Unity. Three tools from ArcGIS Pro contributed to getting an *.slpk file from a shapefile which have been the “Feature To 3D By Attribute”, “Layer 3D To Feature Class” and “Create 3D Object Scene Layer Content” tools. Although the number of shapefiles that needed to be prepared had been limited I thought about building a custom Python tool based on ESRI’s `arcpy` libraries. This custom tool would take a plain shapefile as input and create an *.slpk file as output thus combining 3 out of the box ArcGIS Pro tools into 1 custom tool. Sometimes I found running these three tools one after the other a tedious task and I pondered that

this custom tool could speed up the workflow and make it less error prone. Maybe I will give this a try later on...

Another area of extension might be the digital twin's visuals as rendered by Unity. Since I had already prepared the digital twin in two visual flavors dubbed the "vanilla" and "trippy" edition I wanted to take this one step further. With the "vanilla" edition having a more conventional look and the "trippy" edition having a more playful and neon-like feel to it I found the notion of having "application skins" as a separate entity within the Unity project appealing. Such a "skin" would bundle all the visuals of the project. It felt tempting to be able to easily exchange and expand the look and feel of the twin by having something like a "skin package" that could be plugged into the Unity project. Perhaps this would have been roughly comparable to ArcGIS Pro style files which also change a project's appearance by modifying its symbology.

I also came across different weather related APIs by HERE Technologies (HERE 2022A) which are capable of providing weather data. I thought it would be a nice addition to this digital twin to have live weather information by adding another info panel dealing with the local weather. Even a weather forecast would have been feasible. Not to mention the option to visualize weather conditions like rain or sunshine which Unity would be perfectly capable of. But again time had been a constraint that prevented me from exploring this some more.

Another thing that bothered me in retrospect is how I managed all the dots that are representing vehicles in a Unity scene by using C#. My approach of having all vehicles painted on the scene by looping through a vehicle List worked well enough to have decent performance. But to be honest I was worried if this would scale if the application had to cope with, say, 1000 GameObjects. The hidden issue here is that all vehicles i.e. GameObjects need to be instantiated and destroyed during runtime. This puts a substantial burden on performance even if it turned out to be good enough in the current twin version. But a couple of weeks after the code had been finished I came across a design pattern named "Object Pooling". And this would have been a perfect fit for dealing with large amounts of GameObjects on screen. At the core of "Object Pooling" is to instantiate GameObjects upfront before the application is running and keep them inactive until they are needed. As soon as fresh vehicle location data are in it would be only a matter of "activating" the GameObjects to get them on screen. GameObjects no longer needed to be destroyed but just pushed back into the object pool for later reuse. This approach would offload the burden of instantiating and destroying GameObjects at runtime from the CPU. For sure this is a technique to keep in mind for future projects when working with large amounts of GameObjects.

Talking about platforms: While this digital twin is targeting the Windows 10 platform, it would have been interesting to bring this to mobile platforms as well. I already had some experience with bringing Unity based applications to Android smartphones and I found the process quite seamless. Unity takes care of embedding Android Studio into the building toolchain so there is no need to tinker around manually. However, some kind of "holy grail" would have been to bring this twin to iOS platforms like Apple iPhone oder iPad which Unity is perfectly capable of. But in this case it would have been mandatory for Unity to run on Apple Mac hardware to develop for iOS so this was not an option due to missing hardware.

And talking about platforms some more: I had temporary access to a Microsoft HoloLens device so I thought about pushing the multiplatform capabilities of Unity even more by targeting this augmented reality device with the TriMet digital twin. When I read up on HoloLens documentation I

found that Unity's Universal Render Pipeline would support deployment to HoloLens and soon I imagined a holographic tabletop model of the Portland network with glowing dots that are moving within a Portland area hologram. I'm pretty sure that this would have been an astonishing view...

I would have loved to build on all these ideas but because I had to finish the project in time I was left with a somewhat bittersweet feeling when I completed the project as it is. For one there's a working digital twin that has quite a few technologies folded into it. From my point of view I feel quite satisfied how it turned out technically and visually. But on the other hand there have been ideas that just couldn't be explored due to time and other constraints. But anyway I'm pretty confident that this is not terminal station on my journey across 3D land. And to come full circle again, Ed Catmull, the Pixar mastermind that had been introduced in the beginning will have the final say with a quote from his book "Creativity, Inc." ...

Don't wait for things to be perfect before you share them with others.

Show early and show often.

It'll be pretty when we get there, but it won't be pretty along the way.

Ed Catmull

And this sounds quite comforting to me after all.....

Source Code Appendix

A. GTFS_Data_Loader

C# file "MainWindow.xaml.cs"

```
/*
 * Project           GTFS Data Loader
 * Supervisor       Christoph Traun
 * Author           Winfried Schwan
 * Participant ID    107023
 * Filename          MainWindow.xaml.cs
 * Version          1.0
 * Summary          This script collects GTFS
 *                  realtime data from the
 *                  Portland TriMet GTFS feed and
 *                  stores all data in local
 *                  folders for further
 *                  processing
 *
 * Created          2022-06-26 15:00:00
 * Last modified    2022-08-16 19:20:00
 */

using System;
using System.IO;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;
using System.Net;
using System.Diagnostics;

using RestSharp;

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace TriMetDigitalTwin
{
    public partial class MainWindow : Window
    {
        /*
        * Initialize class variables
        */

        static public string triMetWebServiceUrl =
            "http://developer.trimet.org/ws/v2/vehicles";

        static public string archiveFolderPath;
        static public string workingFolderPath;

        static public string triMetDeveloperKey = "developerKeyByEsri";

        static public int queryIntervalInMilliseconds = 10000;

        static public RestClient connectionTestRestClient;
        static public RestClient triMetRestClient;

        static public RestRequest request;
        static public RestResponse response;
    }
}
```



```

static public DateTime currentDateTime;

static public string vehiclesSummary;

static DispatcherTimer timer;

public MainWindow()
{
    InitializeComponent();

    //
    // set up working and archive folders
    // to hold incoming GTFS data
    //

    archiveFolderPath = ArchiveFolder.Text;
    workingFolderPath = WorkingFolder.Text;

    //
    // set up the timer that is controlling
    // the time interval between requests
    //

    timer = new DispatcherTimer();

    //
    // get time interval from UI
    // and pass QueryResultLabel to Timer_Tick method
    //

    timer.Interval = TimeSpan.FromSeconds(Convert.ToInt32(QueryInterval.Text));
    timer.Tick += (s, args) => Timer_Tick(QueryResultLabel);
}

/*****
 *
 * Method name      StartProcessingGTFSData_Click *
 * Arguments        object sender                *
 *                  RoutedEventArgs e           *
 * Return value     none                        *
 * Summary          Triggers an event as soon as *
 *                  user clicks the button "Start *
 *                  Processing GTFS Data". Event *
 *                  initializes REST client and   *
 *                  sets up a timer that controls *
 *                  the data processing thread   *
 * *****/

private void StartProcessingGTFSData_Click(object sender, RoutedEventArgs e)
{
    StatusLabel.Content = "Processing";

    //
    // set up the REST client for accessing the GTFS feed
    //

    triMetRestClient = initializeRestClient();

    //
    // start timer and adapt time interval if necessary

```

```
//

timer.Interval = TimeSpan.FromSeconds(Convert.ToInt32(QueryInterval.Text));
timer.Start();
}

/*****
 *
 * Method name      StopProcessingGTFSData_Click
 * Arguments        object sender
 *                  RoutedEventArgs e
 * Return value     none
 * Summary          Stops the timer and updates
 *                  status message
 *
 *****/

private void StopProcessingGTFSData_Click(object sender, RoutedEventArgs e)
{
    StatusLabel.Content = "Idle";
    timer.Stop();
}

/*****
 *
 * Method name      Timer_Tick
 * Arguments        Label result
 * Return value     none
 * Summary          Associated timer event that
 *                  will be called every x seconds
 *                  to get latest vehicle
 *                  information including location
 *
 *****/

public static void Timer_Tick(Label result)
{
    getAndSaveVehicleData();

    result.Content = vehiclesSummary;
}

/*****
 *
 * Method name      ConnectionCheck_Click
 * Arguments        object sender
 *                  RoutedEventArgs e
 * Return value     none
 * Summary          Checks if connection to GTFS
 *                  is alive
 *
 *****/

private void ConnectionCheck_Click(object sender, RoutedEventArgs e)
{
    connectionTestRestClient = initializeRestClient();

    //
    // build request to check if TriMet GTFS feed is working
    //

    buildRequest();
}
```

```

//
// fire request and hope for an OK response from TriMet server
//

try
{
    response = connectionTestRestClient.Get(request);

    if (response.StatusCode == HttpStatusCode.OK)
    {
        MessageBox.Show("  GTFS Feed online...", "GTFS Data Realtime Loader",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    else
    {
        MessageBox.Show("  GTFS Feed offline...", "GTFS Data Realtime Loader",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
catch (Exception ex)
{
    MessageBox.Show("  Exception caught... " + ex.Message,
        "GTFS Data Realtime Loader", MessageBoxButton.OK, MessageBoxImage.Error);
}

connectionTestRestClient.Dispose();
}

/*****
*
* Method name      QueryInterval_Changed      *
* Arguments        object sender              *
*                  TextChangedEventArgs e      *
* Return value     none                       *
* Summary          This event is fired as soon as *
*                  user is changing the query *
*                  interval in UI (which is by *
*                  default set to 10 seconds) *
*
*****/

private void QueryInterval_Changed(object sender, TextChangedEventArgs e)
{
    queryIntervalInMilliseconds = Convert.ToInt32(QueryInterval.Text) * 1000;
}

/*****
*
* Method name      WorkingFolder_Changed      *
* Arguments        object sender              *
*                  TextChangedEventArgs e      *
* Return value     none                       *
* Summary          This event is fired as soon as *
*                  user is changing the working *
*                  folder in UI (which is by *
*                  default set to *
*                  D:\tmp\gtfs\working) *
*
*****/

private void WorkingFolder_Changed(object sender, TextChangedEventArgs e)
{
    workingFolderPath = WorkingFolder.Text;
}

```

```

/*****
 *
 * Method name      ArchiveFolder_Changed      *
 * Arguments       object sender              *
 *                 TextChangedEventArgs e       *
 * Return value    none                       *
 * Summary         This event is fired as soon as *
 *                 user is changing the archive *
 *                 folder in UI (which is by   *
 *                 default set to              *
 *                 D:\tmp\gtfs\archive)        *
 *
 *****/

private void ArchiveFolder_Changed(object sender, TextChangedEventArgs e)
{
    archiveFolderPath = ArchiveFolder.Text;
}

/*****
 *
 * Method name      OpenWorkingFolder_Click    *
 * Arguments       object sender              *
 *                 RoutedEventArgs e           *
 * Return value    none                       *
 * Summary         Runs Windows Explorer to   *
 *                 take a look at the working *
 *                 folder                      *
 *
 *****/

private void OpenWorkingFolder_Click(object sender, RoutedEventArgs e)
{
    Process.Start("explorer.exe", workingFolderPath);
}

/*****
 *
 * Method name      OpenArchiveFolder_Click    *
 * Arguments       object sender              *
 *                 RoutedEventArgs e           *
 * Return value    none                       *
 * Summary         Runs Windows Explorer to   *
 *                 take a look at the archive *
 *                 folder                      *
 *
 *****/

private void OpenArchiveFolder_Click(object sender, RoutedEventArgs e)
{
    Process.Start("explorer.exe", archiveFolderPath);
}

/*****
 *
 * Method name      PurgeArchiveFolder_Click   *
 * Arguments       object sender              *
 *                 RoutedEventArgs e           *
 * Return value    none                       *
 * Summary         Deletes all files in archive *
 *                 folder and notifies user   *
 *                 once deletion of files is  *
 *                 complete                    *
 *
 *****/
```

```
*
*****/

private void PurgeArchiveFolder_Click(object sender, RoutedEventArgs e)
{
    if (Directory.Exists(archiveFolderPath))
    {
        DirectoryInfo directoryInfo = new DirectoryInfo(archiveFolderPath);
        FileInfo[] files = directoryInfo.GetFiles();
        foreach (FileInfo file in files)
        {
            file.Delete();
        }
    }

    Thread.Sleep(2000);

    MessageBox.Show("    Archive Folder purged...", "GTFS Data Loader",
        MessageBoxButton.OK, MessageBoxImage.Information);
}

/*****
*
* Method name      PurgeWorkingFolder_Click
* Arguments        object sender
*                  RoutedEventArgs e
* Return value     none
* Summary          Deletes all files in working
*                  folder and notifies user
*                  once deletion of files is
*                  complete
*
*****/

private void PurgeWorkingFolder_Click(object sender, RoutedEventArgs e)
{
    if (Directory.Exists(workingFolderPath))
    {
        DirectoryInfo directoryInfo = new DirectoryInfo(workingFolderPath);
        FileInfo[] files = directoryInfo.GetFiles();
        foreach (FileInfo file in files)
        {
            file.Delete();
        }
    }

    Thread.Sleep(2000);

    MessageBox.Show("    Working Folder purged...", "GTFS Data Loader",
        MessageBoxButton.OK, MessageBoxImage.Information);
}

/*****
*
* Method name      initializeRestClient
* Arguments        none
* Return value     RestClient
* Summary          Initializes the REST client
*                  to prepare for access to
*                  GTFS realtime feed
*
*****/

public static RestClient initializeRestClient()
{
    RestClientOptions options = new RestClientOptions()
```

```

    {
        ThrowOnAnyError = true,
        MaxTimeout = 10000
    };

    RestClient client = new RestClient(options);

    return client;
}

/*****
 *
 * Method name      getAndSaveVehicleData      *
 * Arguments        none                       *
 * Return value     none                       *
 * Summary          Sets up REST request to get *
                   vehicle data from GTFS      *
                   feed, send request, format  *
                   REST response and write to  *
                   working folder              *
 *
 *****/

public static void getAndSaveVehicleData()
{
    //
    // build request to get vehicle positions from TriMet server
    //

    buildRequest();

    //
    // fire request to get vehicle positions from TriMet GTFS feed
    //

    try
    {
        response = triMetRestClient.Get(request);
    }
    catch (Exception e)
    {
        Console.WriteLine("Cannot access TriMet web services, exiting now...");
        Console.WriteLine("Exception: " + e.Message);
    }

    //
    // put prettily formatted JSON data in string formattedVehicleList
    //

    object vehicleList = JsonConvert.DeserializeObject(response.Content);
    string formattedVehicleList = JsonConvert.SerializeObject(vehicleList,
        Formatting.Indented);

    //
    // build the timestamp to be added to JSON file name
    //

    string timeStampJsonFile = buildTimeStampForJsonFile();

    //
    // write formatted GTFS to file in "archive" folder along with timestamp
    //

```

```

writeGtfsFileToArchiveFolder(formattedVehicleList, timeStampJsonFile);

//
// write formatted GTFS to file in "working" folder along with timestamp
//

writeGtfsFileToWorkingFolder(formattedVehicleList, timeStampJsonFile);

//
// write file with .complete extension to indicate that saving of JSON file
// has been completed in "working" folder
//

writeGtfsCompleteFileToWorkingFolder(timeStampJsonFile);

//
// get the number of vehicles in operation from latest service call
//

dynamic allVehicles = JsonConvert.DeserializeObject(formattedVehicleList);

int numberOfVehicles = 0;

numberOfVehicles = getNumberOfVehicles(formattedVehicleList);

//
// get the query's time and date according to received data
//

string queryTimeStamp = getQueryTimeStamp(formattedVehicleList);

//
// output each vehicle's properties
//

printVehicleProperties(allVehicles, numberOfVehicles);

//
// get current local time CEST
//

string currentDateTime = MainWindow.currentDateTime.ToString
    ("dd/MM/yyyy HH:mm:ss");

Console.WriteLine();
Console.WriteLine(numberOfVehicles + " TriMet vehicle location(s) found on " +
    currentDateTime + " CEST");

string pstDateTime = localTimeToPacificTime(MainWindow.currentDateTime);

vehiclesSummary = numberOfVehicles + " TriMet vehicle location(s) found on " +
    currentDateTime + " CEST" + " / " + pstDateTime + " PST";
}

/*****
*
* Method name      buildRequest
* Arguments        none
* Return value     none
* Summary          Build REST request to get data
*                  GTFS live feed
*
*****/

```

```

*
*****/

private static void buildRequest()
{
    request = new RestRequest(triMetWebServiceUrl);
    response = new RestResponse();

    request.AddHeader("User-Agent", "ESRI Unity prototype app");
    request.AddHeader("Accept", "*/*");
    request.AddHeader("Accept-Encoding", "gzip, deflate, br");
    request.AddHeader("Connection", "keep-alive");

    request.AddParameter("appID", "triMetApplicationId");
}

/*****
*
* Method name      getNumberOfVehicles      *
* Arguments        string vehicleList      *
* Return value     int numberOfVehicles    *
* Summary          Get currentnumber of vehicles *
*                 in operation             *
*
*****/

public static int getNumberOfVehicles(string vehicleList)
{
    dynamic allVehicles = JsonConvert.DeserializeObject(vehicleList);

    JSONArray vehicleArray;

    int numberOfVehicles = 0;

    try
    {
        vehicleArray = allVehicles.resultSet.vehicle;
        numberOfVehicles = vehicleArray.Count;
    }
    catch (Exception exception)
    {
        Console.WriteLine("No vehicles operational");
        Console.WriteLine(exception.Message);

        numberOfVehicles = 0;
    }

    return numberOfVehicles;
}

/*****
*
* Method name      getQueryTimeStamp      *
* Arguments        string vehicleList      *
* Return value     string queryTimeStamp  *
* Summary          Get timestamp from GTFS data *
*                 and convert time format from *
*                 UNIX epoch time to local *
*                 Portland time           *
*
*****/

public static string getQueryTimeStamp(string vehicleList)
{
    dynamic allVehicles = JsonConvert.DeserializeObject(vehicleList);

```



```

    string queryTime;

    queryTime = allVehicles.resultSet.queryTime;

    string queryTimeStamp = epochTimeToPacificTime(queryTime);

    return queryTimeStamp;
}

/*****
 *
 * Method name      epochTimeToPacificTime
 * Arguments        string epochTime
 * Return value     string portlandDateTime
 * Summary          Get UTC time and date from
 *                  UNIX epoch time and transform
 *                  to "Pacific Standard Time"
 *                  which happens to be the time
 *                  where the city of Portland,
 *                  Oregon is located in the US
 *
 *                  All Portland time values are
 *                  in milliseconds since UNIX
 *                  epoch
 *
 *                  Portland local time is UTC-7
 *
 *****/

public static string epochTimeToPacificTime(string epochTime)
{
    //
    // cut the last three digits, we need seconds not milliseconds
    //

    string tmpTimeStamp = epochTime.Substring(0, epochTime.Length - 3);

    //
    // change type from string to long
    //

    long timeStamp = Convert.ToInt64(tmpTimeStamp);

    //
    // unix timestamp is seconds past epoch
    //

    DateTime triMetUtcDateTime =
        new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc);

    triMetUtcDateTime = triMetUtcDateTime.AddSeconds(timeStamp);

    DateTime utcTime = triMetUtcDateTime;

    //
    // prepare for getting "Pacific Standard Time"
    //

    string pacificZoneId = "Pacific Standard Time";

    TimeZoneInfo pacificZone = TimeZoneInfo.FindSystemTimeZoneById(pacificZoneId);

    DateTime pacificTime = TimeZoneInfo.ConvertTimeFromUtc(utcTime, pacificZone);

```

```

    string portlandDateTime = string.Format("{0:u}", pacificTime);

    return portlandDateTime;
}

/*****
 *
 * Method name      buildTimeStampForJsonFile
 * Arguments        none
 * Return value     string formattedDateTime
 * Summary          Build a time stamp with
 *                  current time to be used for
 *                  naming of working JSON file
 *
 *****/

public static string buildTimeStampForJsonFile()
{
    currentDateTime = DateTime.Now;

    string formattedDateTime = currentDateTime.ToString("_yyyy_MM_dd_HH_mm_ss");

    return formattedDateTime;
}

/*****
 *
 * Method name      writeGtfsFileToArchiveFolder
 * Arguments        string formattedVehicleList
 *                  string timeStampJsonFile
 * Return value     none
 * Summary          Writes JSON file to archive
 *                  folder
 *
 *****/

private static void writeGtfsFileToArchiveFolder
    (string formattedVehicleList, string timeStampJsonFile)
{
    //
    // write formatted JSON to file along with timestamp
    //

    File.WriteAllText(archiveFolderPath + "vehicle_positions" +
        timeStampJsonFile + ".json", formattedVehicleList);
}

/*****
 *
 * Method name      writeGtfsFileToWorkingFolder
 * Arguments        string formattedVehicleList
 *                  string timeStampJsonFile
 * Return value     none
 * Summary          Writes JSON file to working
 *                  folder
 *
 *****/

private static void writeGtfsFileToWorkingFolder
    (string formattedVehicleList, string timeStampJsonFile)
{
    //
    // write formatted JSON to file along with timestamp
    //

```

```

        File.WriteAllText(workingFolderPath + "vehicle_positions" +
            timeStampJsonFile + ".json", formattedVehicleList);
    }

    /*****
    *
    * Method name      writeGtfsCompleteFileToWorkingFolder *
    * Arguments        string timeStampJsonFile             *
    * Return value     none                                 *
    * Summary          Writes .complete file to working    *
    *                  folder as a signal that writing of    *
    *                  file has finished                    *
    *
    *****/

    private static void writeGtfsCompleteFileToWorkingFolder(string timeStampJsonFile)
    {
        File.WriteAllText(workingFolderPath + "vehicle_positions" +
            timeStampJsonFile + ".complete", "");
    }

    /*****
    *
    * Method name      printVehicleProperties               *
    * Arguments        dynamic allVehicles                 *
    *                  int numberOfVehicles                *
    * Return value     none                                 *
    * Summary          Loops through all captured          *
    *                  vehicles and outputs vehicle       *
    *                  at console (for debugging          *
    *                  purposes)                           *
    *
    *****/

    public static void printVehicleProperties(dynamic allVehicles, int numberOfVehicles)
    {
        string expires;
        string signMessage;
        string serviceDate;
        string loadPercentage;

        string latitude;
        string nextStopSeq;
        string source;
        string type;

        string blockID;
        string signMessageLong;
        string lastLocID;
        string nextLocID;

        string locationInScheduleDay;
        string newTrip;
        string longitude;
        string direction;

        string inCongestion;
        string routeNumber;
        string bearing;
        string garage;

        string tripID;
        string delay;
        string extraBlockID;
        string messageCode;
    }

```

```
string lastStopSeq;
string vehicleId;
string time;
string offRoute;

string portlandDateExpires;
string portlandServiceDate;
string portlandTime;

//
// loop through all vehicles
// and output properties
//

for (int i = 0; i < numberOfVehicles; i++)
{
    expires          = allVehicles.resultSet.vehicle[i].expires;
    signMessage      = allVehicles.resultSet.vehicle[i].signMessage;
    serviceDate      = allVehicles.resultSet.vehicle[i].serviceDate;
    loadPercentage   = allVehicles.resultSet.vehicle[i].loadPercentage;

    latitude         = allVehicles.resultSet.vehicle[i].latitude;
    nextStopSeq      = allVehicles.resultSet.vehicle[i].nextStopSeq;
    source           = allVehicles.resultSet.vehicle[i].source;
    type             = allVehicles.resultSet.vehicle[i].type;

    blockID          = allVehicles.resultSet.vehicle[i].blockID;
    signMessageLong  = allVehicles.resultSet.vehicle[i].signMessageLong;
    lastLocID        = allVehicles.resultSet.vehicle[i].lastLocID;
    nextLocID        = allVehicles.resultSet.vehicle[i].nextLocID;

    locationInScheduleDay =
        allVehicles.resultSet.vehicle[i].locationInScheduleDay;

    newTrip          = allVehicles.resultSet.vehicle[i].newTrip;
    longitude         = allVehicles.resultSet.vehicle[i].longitude;
    direction         = allVehicles.resultSet.vehicle[i].direction;

    inCongestion     = allVehicles.resultSet.vehicle[i].inCongestion;
    routeNumber       = allVehicles.resultSet.vehicle[i].routeNumber;
    bearing           = allVehicles.resultSet.vehicle[i].bearing;
    garage            = allVehicles.resultSet.vehicle[i].garage;

    tripID           = allVehicles.resultSet.vehicle[i].tripID;
    delay             = allVehicles.resultSet.vehicle[i].delay;
    extraBlockID     = allVehicles.resultSet.vehicle[i].extraBlockID;
    messageCode       = allVehicles.resultSet.vehicle[i].messageCode;

    lastStopSeq      = allVehicles.resultSet.vehicle[i].lastStopSeq;
    vehicleId        = allVehicles.resultSet.vehicle[i].vehicleId;
    time              = allVehicles.resultSet.vehicle[i].time;
    offRoute          = allVehicles.resultSet.vehicle[i].offRoute;

    Console.WriteLine("expires:           " + expires);
    Console.WriteLine("signMessage:        " + signMessage);
    Console.WriteLine("serviceDate:       " + serviceDate);
    Console.WriteLine("loadPercentage:    " + loadPercentage);

    Console.WriteLine("latitude:          " + latitude);
    Console.WriteLine("nextStopSeq:       " + nextStopSeq);
    Console.WriteLine("source:            " + source);
    Console.WriteLine("type:              " + type);

    Console.WriteLine("blockID:           " + blockID);
    Console.WriteLine("signMessageLong:   " + signMessageLong);
    Console.WriteLine("lastLocID:         " + lastLocID);
```

```

        Console.WriteLine("nextLocID:           " + nextLocID);

        Console.WriteLine("locationInScheduleDay: " + locationInScheduleDay);
        Console.WriteLine("newTrip:           " + newTrip);
        Console.WriteLine("longitude:         " + longitude);
        Console.WriteLine("direction:         " + direction);

        Console.WriteLine("inCongestion:       " + inCongestion);
        Console.WriteLine("routeNumber:       " + routeNumber);
        Console.WriteLine("bearing:           " + bearing);
        Console.WriteLine("garage:            " + garage);

        Console.WriteLine("tripID:            " + tripID);
        Console.WriteLine("delay:             " + delay);
        Console.WriteLine("extraBlockID:      " + extraBlockID);
        Console.WriteLine("messageCode:       " + messageCode);

        Console.WriteLine("lastStopSeq:       " + lastStopSeq);
        Console.WriteLine("vehicleId:         " + vehicleId);
        Console.WriteLine("time:              " + time);
        Console.WriteLine("offRoute:          " + offRoute);

        portlandDateExpires = epochTimeToPacificTime(expires);
        portlandServiceDate = epochTimeToPacificTime(serviceDate);
        portlandTime = epochTimeToPacificTime(time);

        Console.WriteLine();
        Console.WriteLine("portlandDateExpires PDT: " + portlandDateExpires);
        Console.WriteLine("portlandServiceDate PDT: " + portlandServiceDate);
        Console.WriteLine("portlandTime PDT: " + portlandTime);
    }
}

/*****
*
* Method name      localTimeToPacificTime
* Arguments        DateTime localTime
* Return value     string pacificDateTime
* Summary          Converts Cologne local time
*                  to Portland local time
*
*****/

public static string localTimeToPacificTime(DateTime localTime)
{
    string pacificZoneId = "Pacific Standard Time";

    TimeZoneInfo pacificZone = TimeZoneInfo.FindSystemTimeZoneById(pacificZoneId);

    DateTime pacificTime = TimeZoneInfo.ConvertTime(localTime, pacificZone);

    string pacificDateTime = pacificTime.ToString("dd/MM/yyyy HH:mm:ss");

    return pacificDateTime;
}
}

/*****
* End of file MainWindow.xaml.cs
*****/

```

XAML file "MainWindow.xaml"

```
<!--*****-->
```

```
<!-- Project          GTFS Data Loader          -->
<!-- Supervisor      Christoph Traun           -->
<!-- Author          Winfried Schwan           -->
<!-- Participant ID  107023                   -->
<!-- Filename        MainWindow.xaml           -->
<!-- Version         1.0                      -->
<!-- Summary         This file describes the   -->
<!--                 window layout including  -->
<!--                 different UI controls to -->
<!--                 make it easier to interact -->
<!--                 with the GTFS Data Loader -->
<!--                 -->
<!-- Created         2022-06-26 15:00:00      -->
<!-- Last modified   2022-08-16 12:20:00      -->
<!--*****-->

<!-- Window Properties -->

<Window x:Name="GTFSWindow" x:Class="TriMetDigitalTwin.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:TriMetDigitalTwin"
  mc:Ignorable="d"
  Title="GTFS Realtime Data Loader" Height="557" Width="659" Icon="download_icon.png">

  <!-- Set up a grid to host all controls -->

  <Grid Margin="0,0,2,0" RenderTransformOrigin="0.47,0.63">

    <Grid.RowDefinitions>
      <RowDefinition Height="240*" />
    </Grid.RowDefinitions>

    <!-- Have some nice background image based on New York City subway map -->

    <Image x:Name="NYC_Map" HorizontalAlignment="Left" Height="865"
      VerticalAlignment="Top" Width="649" Source="nyc_transparent.jpg" Margin="0,1,0,-
      336" />

    <!-- Mission control to start/stop collecting GTFS data -->

    <GroupBox Header="GTFS Realtime Cockpit" HorizontalAlignment="Left" Height="73"
      Margin="10,11,0,0" VerticalAlignment="Top" Width="624" BorderThickness="2"
      FontSize="14" />
    <Button x:Name="StartProcessingGTFSData" Content="Start Processing GTFS Data..."
      HorizontalAlignment="Left" Margin="23,38,0,0" VerticalAlignment="Top" Width="200"
      IsDefault="True" Height="33" Click="StartProcessingGTFSData_Click" FontSize="14" />
    <Button x:Name="StopProcessingGTFSData" Content="Stop Processing GTFS Data..."
      HorizontalAlignment="Left" Margin="244,38,0,0" VerticalAlignment="Top" Width="200"
      IsDefault="True" Height="33" Click="StopProcessingGTFSData_Click" FontSize="14" />
    <Label Content="Status:" HorizontalAlignment="Left" Margin="483,41,0,0"
      VerticalAlignment="Top" Width="51" FontSize="14" Height="32" />
    <Label x:Name="StatusLabel" Content="Idle" HorizontalAlignment="Left"
      Margin="529,41,0,0" VerticalAlignment="Top" FontSize="14" />

    <!-- Area to output latest GTFS query results -->

    <GroupBox Header="Latest Query Results" HorizontalAlignment="Left" Height="73"
      Margin="10,88,0,0" VerticalAlignment="Top" Width="624" BorderThickness="2"
      FontSize="14" />
    <Label x:Name="QueryResultLabel" Content="None" HorizontalAlignment="Left"
      Margin="20,116,0,0" VerticalAlignment="Top" Width="584" FontSize="14" />
```

```
<!-- Connection test area -->

<GroupBox Header="Connection Test" HorizontalAlignment="Left" Height="73"
  Margin="10,194,0,0" VerticalAlignment="Top" Width="624" BorderThickness="2"
  FontSize="14"/>
<Label Content="URL GTFS Feed" HorizontalAlignment="Left" Margin="21,221,0,0"
  VerticalAlignment="Top" Width="104" FontSize="14"/>
<TextBox HorizontalAlignment="Left" Height="24" Margin="132,225,0,0"
  TextWrapping="Wrap" Text="http://developer.trimet.org/ws/v2/vehicles"
  VerticalAlignment="Top" Width="285" FontSize="14"/>
<Button x:Name="CheckConnectionHealth" Content="Check Connection Health..."
  HorizontalAlignment="Left" Margin="435,219,0,0" VerticalAlignment="Top"
  Width="186" IsDefault="True" Height="33" Click="ConnectionCheck_Click"
  FontSize="14"/>

<!-- Query time interval settings -->

<GroupBox Header="Query Time Interval" HorizontalAlignment="Left" Height="73"
  Margin="10,271,0,0" VerticalAlignment="Top" Width="624" BorderThickness="2"
  FontSize="14"/>
<Label Content="The latest GTFS realtime data will be pulled every"
  HorizontalAlignment="Left" Margin="20,297,0,0" VerticalAlignment="Top" Width="317"
  FontSize="14"/>
<TextBox x:Name="QueryInterval" HorizontalAlignment="Left" Height="24"
  Margin="339,300,0,0" TextWrapping="Wrap" Text="10" VerticalAlignment="Top"
  Width="51" FontSize="14" TextChanged="QueryInterval_Changed"/>
<Label Content="seconds" HorizontalAlignment="Left" Margin="397,297,0,0"
  VerticalAlignment="Top" Width="82" FontSize="14"/>

<!-- Folder configuration -->

<GroupBox Header="Folder Configuration" HorizontalAlignment="Left" Height="158"
  Margin="10,350,0,0" VerticalAlignment="Top" Width="624" BorderThickness="2"
  FontSize="14"/>
<Label Content="Working Folder" HorizontalAlignment="Left" Margin="20,370,0,0"
  VerticalAlignment="Top" Width="104" FontSize="14"/>
<TextBox x:Name="WorkingFolder" HorizontalAlignment="Left" Height="24"
  Margin="132,374,0,0" TextWrapping="Wrap" Text="D:\tmp\gtfs_working\"
  VerticalAlignment="Top" Width="186" FontSize="14"
  TextChanged="WorkingFolder_Changed"/>
<Button x:Name="OpenWorkingFolder" Content="Open Working Folder..."
  HorizontalAlignment="Left" Margin="132,413,0,0" VerticalAlignment="Top"
  Width="186" IsDefault="True" Height="33" Click="OpenWorkingFolder_Click"
  FontSize="14"/>
<Button x:Name="PurgeWorkingFolder" Content="Purge Working Folder..."
  HorizontalAlignment="Left" Margin="132,459,0,0" VerticalAlignment="Top"
  Width="186" IsDefault="True" Height="33" Click="PurgeWorkingFolder_Click"
  FontSize="14"/>

<Label Content="Archive Folder" HorizontalAlignment="Left" Margin="331,370,0,0"
  VerticalAlignment="Top" Width="104" FontSize="14"/>
<TextBox x:Name="ArchiveFolder" HorizontalAlignment="Left" Height="24"
  Margin="435,374,0,0" TextWrapping="Wrap" Text="D:\tmp\gtfs_archive\"
  VerticalAlignment="Top" Width="186" FontSize="14"
  TextChanged="ArchiveFolder_Changed"/>
<Button x:Name="OpenArchiveFolder" Content="Open Archive Folder..."
  HorizontalAlignment="Left" Margin="433,413,0,0" VerticalAlignment="Top"
  Width="186" IsDefault="True" Height="33" Click="OpenArchiveFolder_Click"
  FontSize="14"/>
<Button x:Name="PurgeArchiveFolder_Copy1" Content="Purge Archive Folder..."
  HorizontalAlignment="Left" Margin="433,459,0,0" VerticalAlignment="Top"
  Width="186" IsDefault="True" Height="33" Click="PurgeArchiveFolder_Click"
  FontSize="14" UseLayoutRounding="True"/>

</Grid>
```

```
</Window>
```

```
<!-- End of file MainWindow.xaml -->
```

B. GTFS_Data_Simulator

C# file "GTFS_Data_Simulator.cs"

```

/*****
*
* Project          TriMet_Portland_Digital_Twin  *
* Supervisor      Christoph Traun              *
* Author          Winfried Schwan              *
* Participant ID   107023                      *
* Filename        GTFS_Data_Simulator.cs      *
* Version         1.0                          *
* Summary         This script simulates a GTFS *
*                 feed by copying previously   *
*                 captured GTFS files from    *
*                 one local folder to another *
*                 to mimick incoming live data *
*
*                 This script is meant to be  *
*                 a "safety net" when no     *
*                 internet connection is     *
*                 available to capture a "live" *
*                 GTFS feed                  *
*
* Created at      2022-09-11 12:00:00         *
* Last modified   2022-09-11 17:00:00         *
*
*****/

using System;
using System.IO;
using System.Threading;

namespace TriMetDigitalTwin
{
    class TriMetDataSimulator
    {
        /*****
        * Set up class variables *
        *****/

        const string SOURCE_FOLDER      = @"D:\tmp\gtfs_data_offline\";
        const string DESTINATION_FOLDER = @"D:\tmp\gtfs_working_test\";

        static string gtfsOfflineDataFolderPath;
        static string gtfsWorkingFolderPath;

        static int copyIntervalInMilliseconds = 10000;

        /*****
        * Main class *
        *****/

        static void Main(string[] args)
        {
            //
            // print welcome message
            //

```



```
printWelcomeMessage();

//
// check number of command line arguments if any
//

int numberOfCommandLineArguments = checkCommandLineArguments(args);

//
// If number of command line arguments is 2
// we assume that custom input/output folders
// should be used otherwise we take the
// default values for input/output folders
//

if(numberOfCommandLineArguments == 2)
{
    gtfsOfflineDataFolderPath = args[0];
    gtfsWorkingFolderPath     = args[1];
}
else
{
    gtfsOfflineDataFolderPath = SOURCE_FOLDER;
    gtfsWorkingFolderPath     = DESTINATION_FOLDER;
}

//
// Read all GTFS files in input folder
//

string[] sortedGtfsFiles = readAndSortGtfsFiles();

//
// Let's see how many GTFS files we've got
//

int numberOfGtfsFiles = sortedGtfsFiles.Length;

//
// Loop through file name array and copy one file
// to working folder every 10 seconds
//

for(int i = 0; i < numberOfGtfsFiles; i++)
{
    string gtfsFileName = Path.GetFileName(sortedGtfsFiles[i]);

    Console.WriteLine("Copying file \"" + gtfsFileName +
                      "\" from \""      + gtfsOfflineDataFolderPath +
                      "\" to \""      + gtfsWorkingFolderPath +
                      "\"");

    File.Copy(sortedGtfsFiles[i],
              Path.Combine(gtfsWorkingFolderPath, gtfsFileName),
              true);

    //
    // Wait 10 seconds until the next file is copied
    // mimicking the query interval with the live GTFS feed
    //

    Thread.Sleep(copyIntervalInMilliseconds);
}
```

```

//
// check if any key has been pressed at console window
// if so break out from loop and exit program
//

if (Console.KeyAvailable)
{
    printExitMessage();
    break;
}
}

/*****
*
* Method name      printWelcomeMessage
* Arguments        none
* Return value     none
* Summary          Prints a welcome message that
*                  indicates that program has
*                  successfully started
*
*****/

static void printWelcomeMessage()
{
    Console.WriteLine("*****");
    Console.WriteLine("*");
    Console.WriteLine("*   Welcome to TriMet Data Simulator   *");
    Console.WriteLine("*");
    Console.WriteLine("*****");
}

/*****
*
* Method name      printExitMessage
* Arguments        none
* Return value     none
* Summary          Prints an exit message that
*                  indicates that program has
*                  been terminated
*
*****/

static void printExitMessage()
{
    Console.WriteLine();
    Console.WriteLine("*****");
    Console.WriteLine("*");
    Console.WriteLine("*   Leaving TriMet Data Simulator...   *");
    Console.WriteLine("*");
    Console.WriteLine("*****");
    Console.WriteLine();
}

/*****
*
* Method name      checkCommandLineArguments
* Arguments        string[] arguments
* Return value     int numberOfArguments
* Summary          Checks if there are any
*                  command line arguments and
*                  if so how many
*
*****/

```

```

*
*****/

static int checkCommandLineArguments(string[] arguments)
{
    if (arguments.Length == 0)
    {
        Console.WriteLine("No command line arguments found.");
    }
    else
    {
        for (int i = 0; i < arguments.Length; i++)
        {
            Console.WriteLine("Command line argument " + i + ": " + arguments[i]);
        }
    }

    Console.WriteLine();

    int numberOfArguments = arguments.Length;

    return numberOfArguments;
}

/*****
*
* Method name      readAndSortGtfsFiles
* Arguments        none
* Return value     string[] gtfsOfflineFiles
* Summary          Checks given folder and
*                  collects all GTFS file names
*                  in folder to prepare for file
*                  copy
*
*****/

static string[] readAndSortGtfsFiles()
{
    string[] gtfsOfflineFiles = Directory.GetFiles(gtfsOfflineDataFolderPath,
        "*.json");

    Array.Sort(gtfsOfflineFiles);

    return gtfsOfflineFiles;
}
}

/*****
* End of file GTFS_Data_Simulator.cs
*****/

```

C. Unity Project TriMet_Portland_Digital_Twin

C# file "GTFS_Data_Processor.cs"

```

/*****
*
* Project          TriMet_Portland_Digital_Twin
* Supervisor      Christoph Traun
* Author          Winfried Schwan
* Participant ID   107023
* Filename        GTFS_Data_Processor.cs
*****/

```

```
* Version          1.0          *
* Summary          This script reads locally *
*                  stored realtime data from the *
*                  Portland TriMet GTFS feed and *
*                  organizes vehicle data *
*                  Vehicle data are then put in *
*                  a List for consumption by *
*                  Unity *
*                  *
* Created          2022-07-22 15:00:00 *
* Last modified    2022-08-16 19:20:00 *
*                  *
*****/

using System;
using System.IO;
using System.Collections.Generic;
using System.Threading;

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

using UnityEngine;

public class GTFS_Data_Processor : MonoBehaviour
{
    //
    // Initializing the class variables...
    //

    static public bool debugMode = false;

    static public string workingFolderPath = @"D:\tmp\gtfs_working\";
    static public string processedFolderPath = @"D:\tmp\gtfs_processed\";

    static public List<Vehicle> vehicleList = new List<Vehicle>();

    static public Thread gtfsProcessorThread;

    static public FileSystemWatcher watcher;

    /*****
    *
    * Method name      Start
    * Arguments        none
    * Return value     none
    * Summary          Start will be executed once
    *                  the scene is loaded
    *
    *                  Here it is used to launch the
    *                  that consumes GTFS data in the
    *                  background
    *
    *****/

    void Start()
    {
        Debug.Log("Running gtfsProcessorThread...");

        gtfsProcessorThread = new Thread(gtfsProcessor);
        gtfsProcessorThread.Start();
    }
}
```

```

/*****
 *
 * Method name      OnApplicationQuit
 * Arguments        none
 * Return value     none
 * Summary          This method will be executed
 *                  when the application will be
 *                  shut down
 *
 *                  Here it is used to kill the
 *                  thread that is responsible for
 *                  processing GTFS data in the
 *                  background
 *
 *                  The file system watcher is
 *                  also shut down here
 *
 *****/

void OnApplicationQuit()
{
    Debug.Log("Killing gtfsProcessorThread...");

    gtfsProcessorThread.Abort();
    watcher.Dispose();
}

/*****
 *
 * Method name      gtfsProcessor
 * Arguments        none
 * Return value     none
 * Summary          This method is responsible for
 *                  watching the working folder
 *                  where newly collected GTFS
 *                  data arrive
 *
 *                  Every time a new file with the
 *                  extension *.complete has been
 *                  created an "OnCreated" event
 *                  will be triggered
 *
 *                  This event is then captured in
 *                  the "OnCreated" method and
 *                  GTFS processing begins
 *
 *****/

private static void gtfsProcessor()
{
    if (debugMode)
    {
        Debug.Log("Initializing FileSystemWatcher targeting folder: " +
            workingFolderPath + "...");
    }

    watcher = new FileSystemWatcher(workingFolderPath);

    watcher.Created += OnCreated;

    watcher.Filter = "*.complete";
    watcher.IncludeSubdirectories = false;
    watcher.EnableRaisingEvents = true;
}

```

```
/*
 *
 * Method name      OnCreated
 * Arguments        object Sender
 *                  FileSystemEventArgs e
 * Return value     none
 * Summary          This method is executed each
 *                  time a new GTFS file has been
 *                  found
 *
 *                  It calls the method
 *                  responsible for GTFS parsing
 *
 */
public static void OnCreated(object sender, FileSystemEventArgs e)
{
    if (debugMode)
    {
        Debug.Log("Event FileCreated captured...");
    }

    processGtfsFiles(e);
}

/*
 *
 * Method name      processGtfsFiles
 * Arguments        FileSystemEventArgs e
 * Return value     none
 * Summary          GTFS file name is built here
 *                  then file will be processed
 *                  and file will be moved from
 *                  "working" to "processed"
 *                  folder
 *
 */
public static void processGtfsFiles(FileSystemEventArgs e)
{
    if (debugMode)
    {
        Debug.Log("File " + e.Name + " has been created.");
    }

    //
    // get filename w/o extension
    //

    string fileNameWithoutExtension = Path.GetFileNameWithoutExtension(e.Name);

    //
    // build json file name
    //

    string fileNameWithJsonExtension = fileNameWithoutExtension + ".json";

    //
    // delete *.complete file
    //

    deleteCompleteFile(e.FullPath);
}
```

```

//
// process json file
//

processJsonFile(workingFolderPath + fileNameWithJsonExtension);

//
// move json file to processed folder
//

moveJsonFileToProcessedFolder(fileNameWithJsonExtension);
}

/*****
*
* Method name      deleteCompleteFile
* Arguments        string completeFileToBeDeleted
* Return value     none
* Summary          Signaling file with complete
*                  extension will be deleted
*                  after processing the GTFS file
*
*****/

public static void deleteCompleteFile(string completeFileToBeDeleted)
{
    if (debugMode)
    {
        Debug.Log(completeFileToBeDeleted + " will be deleted");
    }

    File.Delete(completeFileToBeDeleted);
}

/*****
*
* Method name      processJsonFile
* Arguments        string jsonFileToBeProcessed
* Return value     none
* Summary          Reads GTFS JSON file and
*                  extracts information about
*                  vehicles, finally put all
*                  vehicles into a list
*
*****/

public static void processJsonFile(string jsonFileToBeProcessed)
{
    if (debugMode)
    {
        Debug.Log(jsonFileToBeProcessed + " will be processed");
    }

    string vehicleJsonString;

    using (StreamReader streamReader = new StreamReader(jsonFileToBeProcessed))
    {
        vehicleJsonString = streamReader.ReadToEnd();
    }

//
// put vehicles in List
//

```

```

        pushVehiclesInList(vehicleJsonString);
    }

/*****
*
* Method name      moveJsonFileToProcessedFolder *
* Arguments        string jsonFileToBeMoved     *
* Return value     none                          *
* Summary          After GTFS JSON processing has *
*                  finished processed file will  *
*                  moved to "processed" folder   *
*
*****/

public static void moveJsonFileToProcessedFolder(string jsonFileToBeMoved)
{
    if (debugMode)
    {
        Debug.Log(jsonFileToBeMoved + " will be moved");
    }

    string sourceFileName = workingFolderPath + jsonFileToBeMoved;
    string destFileName = processedFolderPath + jsonFileToBeMoved;

    File.Move(sourceFileName, destFileName);
}

/*****
*
* Method name      pushVehiclesInList           *
* Arguments        string vehicleJsonString     *
* Return value     none                          *
* Summary          Build a list of type Vehicle *
*                  that contains all vehicle    *
*                  information from latest GTFS *
*                  JSON file                     *
*
*****/

public static void pushVehiclesInList(string vehicleJsonString)
{
    dynamic allVehicles = JsonConvert.DeserializeObject(vehicleJsonString);

    int numberOfVehicles = 0;

    //
    // find out how many vehicles are in operation
    // currently
    //

    numberOfVehicles = getNumberOfVehicles(vehicleJsonString);

    vehicleList.Clear();

    string expires;
    string signMessage;
    string serviceDate;
    string loadPercentage;

    string latitude;
    string nextStopSeq;
    string source;
    string type;

```



```
string blockID;
string signMessageLong;
string lastLocID;
string nextLocID;

string locationInScheduleDay;
string newTrip;
string longitude;
string direction;

string inCongestion;
string routeNumber;
string bearing;
string garage;

string tripID;
string delay;
string extraBlockID;
string messageCode;

string lastStopSeq;
string vehicleID;
string time;
string offRoute;

for (int i = 0; i < numberOfVehicles; i++)
{
    //
    // collect all vehicle properties
    //

    expires           = allVehicles.resultSet.vehicle[i].expires;
    signMessage       = allVehicles.resultSet.vehicle[i].signMessage;
    serviceDate       = allVehicles.resultSet.vehicle[i].serviceDate;
    loadPercentage    = allVehicles.resultSet.vehicle[i].loadPercentage;

    latitude          = allVehicles.resultSet.vehicle[i].latitude;
    nextStopSeq       = allVehicles.resultSet.vehicle[i].nextStopSeq;
    source             = allVehicles.resultSet.vehicle[i].source;
    type              = allVehicles.resultSet.vehicle[i].type;

    blockID           = allVehicles.resultSet.vehicle[i].blockID;
    signMessageLong   = allVehicles.resultSet.vehicle[i].signMessageLong;
    lastLocID         = allVehicles.resultSet.vehicle[i].lastLocID;
    nextLocID         = allVehicles.resultSet.vehicle[i].nextLocID;

    locationInScheduleDay = allVehicles.resultSet.vehicle[i].locationInScheduleDay;
    newTrip           = allVehicles.resultSet.vehicle[i].newTrip;
    longitude         = allVehicles.resultSet.vehicle[i].longitude;
    direction         = allVehicles.resultSet.vehicle[i].direction;

    inCongestion      = allVehicles.resultSet.vehicle[i].inCongestion;
    routeNumber       = allVehicles.resultSet.vehicle[i].routeNumber;
    bearing           = allVehicles.resultSet.vehicle[i].bearing;
    garage            = allVehicles.resultSet.vehicle[i].garage;

    tripID            = allVehicles.resultSet.vehicle[i].tripID;
    delay             = allVehicles.resultSet.vehicle[i].delay;
    extraBlockID      = allVehicles.resultSet.vehicle[i].extraBlockID;
    messageCode       = allVehicles.resultSet.vehicle[i].messageCode;

    lastStopSeq       = allVehicles.resultSet.vehicle[i].lastStopSeq;
    vehicleID         = allVehicles.resultSet.vehicle[i].vehicleID;
    time              = allVehicles.resultSet.vehicle[i].time;
    offRoute          = allVehicles.resultSet.vehicle[i].offRoute;
```

```
//
// build a new, temporary Vehicle object
//

Vehicle tmpVehicle = new Vehicle();

//
// assign properties to Vehicle object
//

tmpVehicle.expires           = expires;
tmpVehicle.signMessage      = signMessage;
tmpVehicle.serviceDate      = serviceDate;
tmpVehicle.loadPercentage    = loadPercentage;

tmpVehicle.latitude         = latitude;
tmpVehicle.nextStopSeq      = nextStopSeq;
tmpVehicle.source           = source;
tmpVehicle.type             = type;

tmpVehicle.blockID          = blockID;
tmpVehicle.signMessageLong  = signMessageLong;
tmpVehicle.lastLocID        = lastLocID;
tmpVehicle.nextLocID        = nextLocID;

tmpVehicle.locationInScheduleDay = locationInScheduleDay;
tmpVehicle.newTrip          = newTrip;
tmpVehicle.longitude         = longitude;
tmpVehicle.direction         = direction;

tmpVehicle.inCongestion     = inCongestion;
tmpVehicle.routeNumber      = routeNumber;
tmpVehicle.bearing          = bearing;
tmpVehicle.garage           = garage;

tmpVehicle.tripID           = tripID;
tmpVehicle.delay            = delay;
tmpVehicle.extraBlockID     = extraBlockID;
tmpVehicle.messageCode      = messageCode;

tmpVehicle.lastStopSeq      = lastStopSeq;
tmpVehicle.vehicleID        = vehicleID;
tmpVehicle.time             = time;
tmpVehicle.offRoute         = offRoute;

//
// add newly created Vehicle object to list vehicleList
//

vehicleList.Add(tmpVehicle);
}

//
// if debug flag is enabled then output
// vehicle properties to Unity console
// for verification
//

if (debugMode)
{
    for (int i = 0; i < vehicleList.Count; i++)
    {
        Debug.Log("expires: " + vehicleList[i].expires);
        Debug.Log("signMessage: " + vehicleList[i].signMessage);
    }
}
```

```

        Debug.Log("serviceDate: " + vehicleList[i].serviceDate);
        Debug.Log("loadPercentage: " + vehicleList[i].loadPercentage);
        Debug.Log("latitude: " + vehicleList[i].latitude);
        Debug.Log("nextStopSeq: " + vehicleList[i].nextStopSeq);
        Debug.Log("source: " + vehicleList[i].source);
        Debug.Log("type: " + vehicleList[i].type);
        Debug.Log("blockID: " + vehicleList[i].blockID);
        Debug.Log("signMessageLong: " + vehicleList[i].signMessageLong);
        Debug.Log("lastLocID: " + vehicleList[i].lastLocID);
        Debug.Log("nextLocID: " + vehicleList[i].nextLocID);
        Debug.Log("locationInScheduleDay: " + vehicleList[i].locationInScheduleDay);
        Debug.Log("newTrip: " + vehicleList[i].newTrip);
        Debug.Log("longitude: " + vehicleList[i].longitude);
        Debug.Log("direction: " + vehicleList[i].direction);
        Debug.Log("inCongestion: " + vehicleList[i].inCongestion);
        Debug.Log("routeNumber: " + vehicleList[i].routeNumber);
        Debug.Log("bearing: " + vehicleList[i].bearing);
        Debug.Log("garage: " + vehicleList[i].garage);
        Debug.Log("tripID: " + vehicleList[i].tripID);
        Debug.Log("delay: " + vehicleList[i].delay);
        Debug.Log("extraBlockID: " + vehicleList[i].extraBlockID);
        Debug.Log("messageCode: " + vehicleList[i].messageCode);
        Debug.Log("lastStopSeq: " + vehicleList[i].lastStopSeq);
        Debug.Log("vehicleID: " + vehicleList[i].vehicleID);
        Debug.Log("time: " + vehicleList[i].time);
        Debug.Log("offRoute: " + vehicleList[i].offRoute);
    }
}

}

/*****
*
* Method name      getNumberOfVehicles
* Arguments        string vehicleList
* Return value     int numberOfVehicles
* Summary          Gets number of vehicles
*                  in operation
*
* *****/

public static int getNumberOfVehicles(string vehicleList)
{
    dynamic allVehicles = JsonConvert.DeserializeObject(vehicleList);

    JSONArray vehicleArray;

    int numberOfVehicles = 0;

    try
    {
        vehicleArray = allVehicles.resultSet.vehicle;
        numberOfVehicles = vehicleArray.Count;
    }
    catch (Exception exception)
    {
        Debug.Log("No vehicles operational");
        Debug.Log(exception.Message);
    }

    return numberOfVehicles;
}

/*****
*
* Method name      getVehicleList
* Arguments        string vehicleList
* *****/

```

```
* Return value      List<Vehicle> vehicleList      *
* Summary          Exposes list vehicleList to  *
*                  other scripts in Unity project *
*                  *                               *
*****/

public static List<Vehicle> getVehicleList()
{
    return vehicleList;
}

/*****
*
* Class name      Vehicle
* Summary        Defines a class Vehicle
*                according to the vehicle info
*                pulled from Portland TriMet
*                developer site
*
*                Comments to vehicle properties
*                have been pulled directly from
*                TriMet developer documentation
*
*****/

public class Vehicle
{
    // Time this vehicle's entry should be discarded
    // if no new position information is received from the vehicle

    public string expires { get; set; }

    // Vehicle's over head sign text message

    public string signMessage { get; set; }

    // Midnight of the service day the vehicle is performing service for

    public string serviceDate { get; set; }

    public string loadPercentage { get; set; }

    // Latitude of the vehicle

    public string latitude { get; set; }

    public string nextStopSeq { get; set; }

    public string source { get; set; }

    // Identifies the type of vehicle. Can be "bus" or "rail"

    public string type { get; set; }

    public string blockID { get; set; }

    // Vehicle's full over head sign text message

    public string signMessageLong { get; set; }

    public string lastLocID { get; set; }

    // Location ID (or stopID) of the next stop
    // this vehicle is scheduled to serve

    public string nextLocID { get; set; }
```

```

// Number of seconds since midnight from the scheduleDate
// that the vehicle is positioned at along its schedule

public string locationInScheduleDay { get; set; }

// Will be true when the trip the vehicle is servicing is new
// and was not part of the published schedule

public string newTrip { get; set; }

// Longitude of the vehicle

public string longitude { get; set; }

// Direction of the route the vehicle is servicing

public string direction { get; set; }

public string inCongestion { get; set; }

// Route number the vehicle is servicing

public string routeNumber { get; set; }

// Bearing of the vehicle if available
// 0 is north, 180 is south

public string bearing { get; set; }
public string garage { get; set; }

// TripID the vehicle is servicing

public string tripID { get; set; }

// Delay of the vehicle along its schedule
// Negative is late. Positive is early

public string delay { get; set; }
public string extraBlockID { get; set; }

// Identifier for the over head sign message
// This is used internally at TriMet and can be ignored

public string messageCode { get; set; }

public string lastStopSeq { get; set; }

// Identifies the vehicle

public string vehicleID { get; set; }

// Time this position was initially recorded

public string time { get; set; }

public string offRoute { get; set; }    }

}

/*****
* End of file GTFS_Data_Processor.cs          *
*****/

```

C# file "GTFS_Stops_Data_Loader.cs"

```

/*****
*
*

```

```

* Project          TriMet_Portland_Digital_Twin  *
* Supervisor      Christoph Traun              *
* Author          Winfried Schwan              *
* Participant ID   107023                      *
* Filename        GTFS_Stops_Data_Loader.cs    *
* Version         1.0                          *
* Summary         This script reads the locally *
*                stored static GTFS file      *
*                stops.txt, a CSV file that   *
*                contains stop ids and names  *
*                *                             *
*                Stops ids and names are then *
*                exposed as a Dictionary for  *
*                consumption by other Unity   *
*                scripts                      *
*                *                             *
* Created         2022-07-26 08:15:00         *
* Last modified   2022-08-16 19:20:00         *
*                *                             *
*****/

using System.Collections.Generic;
using System.IO;

using UnityEngine;

public class GTFS_Stops_Data_Loader : MonoBehaviour
{
    //
    // Initializing the class variables...
    //

    //
    // Path to static GTFS file stops.txt that contains
    // stop ids and stop names in a csv format
    //

    private static string gtfsStopsFilePath = @"D:\tmp\gtfs_stops\stops.txt";

    //
    // set up a Dictionary that holds all the stop names
    // indexed by the stop id
    //

    public static Dictionary<int, string> gtfsStopsDictionary;

    /*****
    *
    * Method name      Start
    * Arguments        none
    * Return value     none
    * Summary          Start will be executed once
    *                 the scene is loaded
    *
    *                 It reads the static GTFS file
    *                 stops.txt and pushes stops ids
    *                 and names into a Dictionary
    *
    *****/

    void Start()
    {
        gtfsStopsDictionary = new Dictionary<int, string>();
    }
}

```

```

//
// read the stops.txt file and extract
// stop id (field 0) and stop name (field 2)
//

using (var gtfsStopsReader = new StreamReader(gtfsStopsFilePath))
{
    while (!gtfsStopsReader.EndOfStream)
    {
        string stopsFileLine = gtfsStopsReader.ReadLine();
        string[] stopsFileLineFields = stopsFileLine.Split(',');

        //
        // check if first field contains really a number
        // thus excluding anything that contains a string
        // in the first field which happened on some occasions
        //

        int integerKey;

        bool isKeyNumerical = int.TryParse(stopsFileLineFields[0], out integerKey);

        if(isKeyNumerical)
        {
            gtfsStopsDictionary.Add(integerKey, stopsFileLineFields[2]);
        }
    }
}

/*****
 *
 * Method name      getStopName
 * Arguments        int stopId
 * Return value     gtfsStopsDictionary[stopId]
 * Summary          This method returns the stop
 *                  name in plain language as a
 *                  string when it is called with
 *                  the stop id from other scripts
 *
 *****/

public static string getStopName(int stopId)
{
    return gtfsStopsDictionary[stopId];
}

/*****
 * End of file GTFS_Stops_Data_Loader.cs
 *****/

```

C# file "Paint_Vehicles_Regular.cs"

```

/*****
 *
 * Project          TriMet_Portland_Digital_Twin
 * Supervisor       Christoph Traun
 * Author           ESRI
 * Customizing      Winfried Schwan
 * Participant ID   107023
 * Filename         Paint_Vehicles_Regular.cs
 *
 *****/

```

```

* Version          1.0          *
* Summary          This script is responsible *
*                  for painting the vehicles *
*                  in the scene           *
*                  *               *
*                  Yellow and red sphere prefabs *
*                  symbolize vehicle positions *
*                  for MAX and bus vehicles *
*                  respectively           *
*                  *               *
*                  Part of the prefab is the *
*                  ArcGISLocationComponent that *
*                  expects longitude/latitude *
*                  values to place the vehicles *
*                  correctly in the scene     *
*                  *               *
* Created          2022-08-10 15:30:00      *
* Last modified    2022-08-17 11:30:00      *
*                  *               *
*****/

using System;
using System.Collections.Generic;
using System.Globalization;

using Esri.ArcGISMapsSDK.Components;
using Esri.GameEngine.Geometry;

using UnityEngine;

public class Paint_Vehicles_Regular : MonoBehaviour
{
    //
    // Initializing the class variables...
    //

    public static List<GTFS_Data_Processor.Vehicle> vehicleList;

    public static GameObject[] prefabs = new GameObject[500];

    public static GameObject arcGISMap;

    public static GameObject redSpherePrefab;
    public static GameObject yellowSpherePrefab;

    public static GameObject prefabsGameObject;

    public static int queryInterval = 10;

    /*****
    *
    * Method name      Start          *
    * Arguments        none           *
    * Return value     none           *
    * Summary          Start will be executed once *
    *                  the scene is loaded         *
    *                  *               *
    *                  Initializes the various *
    *                  prefabs used for drawing the *
    *                  vehicles on the map *
    *                  Takes also care of updating *
    *                  vehicle positions every 10 *
    *                  seconds           *
    *                  *               *
    *****/
}

```



```

void Start()
{
    arcGISMap = GameObject.Find("ArcGISMap");

    redSpherePrefab    = (GameObject)Resources.Load("Prefabs/RedSphere_Regular");
    yellowSpherePrefab = (GameObject)Resources.Load("Prefabs/YellowSphere_Regular");

    prefabsGameObject = GameObject.Find("Prefabs");

    //
    // call function paintVehicles every 10 seconds
    // according to value of queryInterval
    //

    InvokeRepeating("paintVehicles", 0, queryInterval);
}

/*****
*
* Method name      paintVehicles
* Arguments        none
* Return value     none
* Summary          Consumes the vehicle List
*                  that is built by
*                  GTFS_Data_Processor script
*                  Extracts longitude and
*                  latitude to use it for
*                  vehicle placement on the map
*                  Different prefabs are used
*                  for MAX vehicles and buses
*
*****/

void paintVehicles()
{
    //
    // pull vehicle data from List
    //

    vehicleList = GTFS_Data_Processor.vehicleList;

    ArcGISPoint vehiclePosition;

    //
    // loop through List of vehicles
    //

    for (int i = 0; i < vehicleList.Count; i++)
    {
        //
        // get vehicle longitude and latitude from List
        //

        string vehicleLongitudeString = vehicleList[i].longitude;
        string vehicleLatitudeString = vehicleList[i].latitude;

        double vehicleLongitude = Double.Parse(vehicleLongitudeString,
            CultureInfo.GetCultureInfo("en-US"));
        double vehicleLatitude  = Double.Parse(vehicleLatitudeString,
            CultureInfo.GetCultureInfo("en-US"));

        //
        // if we deal with a MAX vehicle use the yellow sphere prefab
        // assign a vehicle index for later reference
    }
}

```

```
//  
  
if (vehicleList[i].type.Equals("rail") &&  
    vehicleList[i].signMessageLong.Contains("MAX"))  
{  
    //  
    // instantiate the yellow sphere prefab for MAX vehicles  
    //  
  
    prefabs[i] = Instantiate<GameObject>(yellowSpherePrefab, arcGISMap.transform);  
  
    //  
    // make the Prefabs game object the parent of all instantiated prefabs  
    //  
  
    prefabs[i].transform.parent = prefabsGameObject.transform;  
  
    //  
    // assign a vehicle index for later reference  
    //  
  
    prefabs[i].GetComponent<VehicleIndex>().vehicleIndex = i;  
  
    //  
    // make yellow sphere prefab visible  
    //  
  
    prefabs[i].SetActive(true);  
  
    //  
    // let instantiated prefabs disappear after 10 seconds  
    //  
  
    Destroy(prefabs[i].gameObject, queryInterval);  
  
    //  
    // assign longitude and latitude to vehicle  
    //  
  
    vehiclePosition = new ArcGISPoint(vehicleLongitude, vehicleLatitude,  
        4000, new ArcGISSpatialReference(4326));  
}  
else  
{  
    //  
    // instantiate the red sphere prefab for buses and other vehicles except MAX  
    //  
  
    prefabs[i] = Instantiate<GameObject>(redSpherePrefab, arcGISMap.transform);  
  
    //  
    // make the Prefabs game object the parent of all instantiated prefabs  
    //  
  
    prefabs[i].transform.parent = prefabsGameObject.transform;  
  
    //  
    // assign a vehicle index for later reference  
    //  
  
    prefabs[i].GetComponent<VehicleIndex>().vehicleIndex = i;
```

```

//
// make red sphere prefab visible
//

prefabs[i].SetActive(true);

//
// let instantiated prefabs disappear after 10 seconds
//

Destroy(prefabs[i].gameObject, queryInterval);

//
// assign longitude and latitude to vehicle
//

vehiclePosition = new ArcGISPoint(vehicleLongitude, vehicleLatitude,
    3000, new ArcGISSpatialReference(4326));
}

//
// get the location component of prefab
//

ArcGISLocationComponent locationComponent =
    prefabs[i].GetComponent<ArcGISLocationComponent>();

locationComponent.enabled = true;

//
// assign the vehicle position to location component
//

locationComponent.Position = vehiclePosition;
}
}

/*****
* End of file Paint_Vehicles_Regular.cs      *
*****/

```

C# file "Animate_Headline_Regular.cs"

```

/*****
*
* Project           TriMet_Portland_Digital_Twin  *
* Supervisor       Christoph Traun              *
* Author           Winfried Schwan              *
* Participant ID   107023                       *
* Filename         Animate_Headline_Regular.cs  *
* Version         1.0                           *
* Summary         This script is responsible    *
*                 for displaying the scene      *
*                 titles that gives some       *
*                 context on this scene        *
*
* Created         2022-07-26 08:15:00           *
* Last modified   2022-08-17 12:30:00          *
*
*****/

```

```
using System;

using UnityEngine;
using TMPro;

public class Animate_Headline_Regular : MonoBehaviour
{
    //
    // Initializing the class variables...
    //

    public static GameObject messageGameObject;

    public static Material messageMaterial;

    private TextMeshProUGUI textDisplayMessage;

    private float dilationValue = -1.0f;

    //
    // Timer responsible for changing titles
    // after waitTime seconds
    //

    private float timer = 0.0f;
    private float waitTime = 8.0f;

    //
    // string array to hold the different messages
    //

    private string[] messageArray;

    bool increasing = true;

    int messageCounter = 0;
    int moduloMessageCounter = 0;

    public static int numberOfMessages = 0;

    //
    // Portland local time is also displayed
    // as one of the messages so these variables
    // will hold Cologne local time and Portland
    // local time
    //

    DateTime localDateTime;
    string pacificDateTime;

    /*****
    *
    * Method name      Start
    * Arguments        none
    * Return value     none
    * Summary          Start will be executed once
    *                  the scene is loaded
    *
    *                  The message array gets
    *                  initialized with message texts
    *                  I use the dilation/thickness
    *                  property of the font to let
    *                  the characters fade in and out
    *
    *****/
}
```

```

void Start()
{
    //
    // get local Cologne time
    //

    localDateTime = DateTime.Now;

    //
    // build a string that holds local Portland time
    //

    string pacificDateTime = localTimeToPacificTime(localDateTime);

    //
    // set up the message array that will
    // contain the headlines
    //

    messageArray = new string[10];

    numberOfMessages = 4;

    messageArray[0] = "The TriMet Network in Portland";
    messageArray[1] = "Live from TriMet's GTFS Realtime Feed";
    messageArray[2] = "The \"Vanilla\" Edition";
    messageArray[3] = "Portland Local Time " + pacificDateTime;

    messageGameObject = GameObject.Find("Headline");

    textDisplayMessage = messageGameObject.GetComponent<TextMeshProUGUI>();
    textDisplayMessage.text = messageArray[messageCounter];

    messageMaterial = textDisplayMessage.fontSharedMaterial;

    //
    // initialize the dilation value with -0.1 that is invisible
    //

    messageMaterial.SetFloat(ShaderUtilities.ID_FaceDilate, -1.0f);
}

/*****
*
* Method name      Update
* Arguments       none
* Return value    none
* Summary         Update is called once per
*                 frame (as a rule of thumb
*                 60 times per seconds)
*
*                 Message to be displayed is
*                 changed every 8 seconds
*                 Dilation value of font is
*                 continously changed to achieve
*                 a fade in/fade out effect
*                 Dilation value swings from
*                 -0.3 to -1.0 and back again
*
*****/

void Update()
{

```

```
//
// set up the timer to handle fade in/out duration
//

timer += Time.deltaTime;

if (timer < waitTime)
{
    if (increasing)
    {
        //
        // increase dilation value of font
        // using Time.deltaTime to make it more
        // GPU performance independent
        // (Time.deltaTime = seconds between frames)
        //

        dilationValue = messageMaterial.GetFloat(ShaderUtilities.ID_FaceDilate);

        dilationValue += 0.18f * Time.deltaTime;

        messageMaterial.SetFloat(ShaderUtilities.ID_FaceDilate, dilationValue);

        if (dilationValue >= -0.3)
        {
            increasing = false;
        }
    }
    else
    {
        //
        // decrease dilation value of font
        // using Time.deltaTime to make it more
        // GPU performance independent
        // (Time.deltaTime = seconds between frames)
        //

        dilationValue = messageMaterial.GetFloat(ShaderUtilities.ID_FaceDilate);

        dilationValue -= 0.18f * Time.deltaTime;

        messageMaterial.SetFloat(ShaderUtilities.ID_FaceDilate, dilationValue);

        if (dilationValue <= -1.0)
        {
            increasing = true;
        }
    }
}
else
{
    //
    // update current local time
    //

    localDateTime = DateTime.Now;

    pacificDateTime = localTimeToPacificTime(localDateTime);

    messageArray[3] = "Portland Local Time: " + pacificDateTime;

    //
    // reset timer and change displayed title
    //

    timer = 0;
    dilationValue = -1.0f;
}
```

```

        messageCounter++;

        moduloMessageCounter = messageCounter % numberOfMessages;

        messageMaterial.SetFloat(ShaderUtilities.ID_FaceDilate, dilationValue);
        textDisplayMessage.text = messageArray[moduloMessageCounter];
    }
}

/*****
 *
 * Method name      localTimeToPacificTime      *
 * Arguments       DateTime localTime          *
 * Return value    string pacificDateTime      *
 * Summary        Converts Cologne local time  *
 *                to Portland local time      *
 *                *                            *
 *****/

public static string localTimeToPacificTime(DateTime localTime)
{
    string pacificZoneId = "Pacific Standard Time";

    TimeZoneInfo pacificZone = TimeZoneInfo.FindSystemTimeZoneById(pacificZoneId);

    DateTime pacificTime = TimeZoneInfo.ConvertTime(localTime, pacificZone);

    string pacificDateTime = pacificTime.ToString("HH:mm:ss" + " PST");

    return pacificDateTime;
}
}

/*****
 * End of file Animate_Headline_Regular.cs      *
 *****/

```

C# file "Display_Vehicle_Information.cs"

```

/*****
 *
 * Project          TriMet_Portland_Digital_Twin  *
 * Supervisor      Christoph Traun              *
 * Author          Winfried Schwan              *
 * Participant ID  107023                      *
 * Filename        Display_Vehicle_Information.cs *
 * Version         1.0                          *
 * Summary        This script is responsible    *
 *                for displaying vehicle info   *
 *                at the vehicle information    *
 *                panel                          *
 *                *                            *
 * Created        2022-08-09 07:30:00          *
 * Last modified  2022-08-17 13:30:00          *
 *                *                            *
 *****/

using System;
using System.Collections.Generic;

using UnityEngine;
using TMPro;

public class Display_Vehicle_Information : MonoBehaviour

```

```
{
    //
    // Initializing the class variables...
    //

    //
    // access the List with information on all current
    // vehicles operating
    //

    public static List<GTFS_Data_Processor.Vehicle> vehicleList;

    //
    // set up the game objects that are part
    // of the vehicle information panel
    //

    public static GameObject messageGameObject01;
    public static GameObject messageGameObject02;
    public static GameObject messageGameObject03;
    public static GameObject messageGameObject04;

    //
    // set up TextMesh Pro objects to be able
    // change the texts in vehicle information
    // panel
    //

    private TextMeshProUGUI textDisplayMessage01;
    private TextMeshProUGUI textDisplayMessage02;
    private TextMeshProUGUI textDisplayMessage03;
    private TextMeshProUGUI textDisplayMessage04;

    /*****
    *
    * Method name      Start
    * Arguments        none
    * Return value     none
    * Summary          Start will be executed once
    *                  the scene is loaded
    *
    *                  Collect the text objects as
    *                  part of the vehicle
    *                  information panel for on the
    *                  fly text changing
    *
    *****/

    void Start()
    {
        //
        // find game objects in project hierarchy
        // that will contain the vehicle information later on
        //

        messageGameObject01 = GameObject.Find("Body_Dynamic_Information_01");
        messageGameObject02 = GameObject.Find("Body_Dynamic_Information_02");
        messageGameObject03 = GameObject.Find("Body_Dynamic_Information_03");
        messageGameObject04 = GameObject.Find("Body_Dynamic_Information_04");

        //
        // get the TextMesh Pro objects to enable
        // text manipulation in panel
        //
    }
}
```



```

textDisplayMessage01 = messageGameObject01.GetComponent<TextMeshProUGUI>();
textDisplayMessage02 = messageGameObject02.GetComponent<TextMeshProUGUI>();
textDisplayMessage03 = messageGameObject03.GetComponent<TextMeshProUGUI>();
textDisplayMessage04 = messageGameObject04.GetComponent<TextMeshProUGUI>();
}

/*****
*
* Method name      OnMouseOver
* Arguments        none
* Return value     none
* Summary          This event will be triggered
*                  during runtime when the mouse
*                  pointer hovers over a vehicle
*                  While hovering several pieces
*                  of vehicle information are
*                  pulled form List vehicleList
*                  and displayed in the vehicle
*                  information panel
*
*                  1) Sign message of vehicle
*                  2) Name of next stop
*                  3) Name of previous stop
*                  4) Vehicle type
*                  5) Longitude of vehicle
*                  6) Latitude of vehicle
*
*****/

void OnMouseOver()
{
    int vehicleIndex = GetComponent<VehicleIndex>().vehicleIndex;

    if (vehicleIndex >= 0)
    {
        textDisplayMessage01.text =
            GTFS_Data_Processor.vehicleList[vehicleIndex].signMessage;

        textDisplayMessage02.text =
            GTFS_Stops_Data_Loader.getStopName(
                Int32.Parse(GTFS_Data_Processor.vehicleList[vehicleIndex].nextLocID));

        textDisplayMessage03.text =
            GTFS_Stops_Data_Loader.getStopName(
                Int32.Parse(GTFS_Data_Processor.vehicleList[vehicleIndex].lastLocID));

        textDisplayMessage04.text =
            GTFS_Data_Processor.vehicleList[vehicleIndex].type.ToUpper() +
            "<br>" +
            GTFS_Data_Processor.vehicleList[vehicleIndex].longitude +
            "<br>" +
            GTFS_Data_Processor.vehicleList[vehicleIndex].latitude;
    }
}

/*****
*
* Method name      OnMouseExit
* Arguments        none
* Return value     none
* Summary          This event will be fired
*                  during runtime if the mouse
*                  pointer stops hovering over a
*                  vehicle. As soon as this
*                  happens all the text in the
*                  vehicle information panel is
*****/

```

```

*                               removed                               *
*                               *                                     *
*****/

void OnMouseExit ()
{
    textDisplayMessage01.text = "";
    textDisplayMessage02.text = "";
    textDisplayMessage03.text = "";
    textDisplayMessage04.text = "";
}
}

/*****
* End of file Display_Vehicle_Information.cs      *
*****/

```

C# file "Vehicle_Index.cs"

```

/*****
*                               *
* Project      TriMet_Portland_Digital_Twin  *
* Supervisor   Christoph Traun              *
* Author       Winfried Schwan              *
* Participant ID 107023                     *
* Filename     Vehicle_Index.cs             *
* Version      1.0                          *
* Summary      This script is meant to be an *
*              attachment for the red and   *
*              yellow spheres that represent *
*              the vehicle location         *
*              This vehicle index helps to  *
*              link vehicle information to   *
*              its sphere representation    *
*              in the scene                 *
*              This link is a precondition to *
*              be able to display vehicle   *
*              info while the mouse pointer *
*              is hovering over the sphere  *
*              representation of the vehicle *
*                               *
* Created      2022-08-12 16:00:00          *
* Last modified 2022-08-17 14:00:00        *
*                               *
*****/

using UnityEngine;

public class Vehicle_Index : MonoBehaviour
{
    public int vehicleIndex = 0;
}

/*****
* End of file VehicleIndex.cs              *
*****/

```

C# file "Escape_Key_Handler.cs"

```

/*****
*                               *
* Project      TriMet_Portland_Digital_Twin  *

```

```

* Supervisor      Christoph Traun      *
* Author          Winfried Schwan      *
* Participant ID  107023                *
* Filename        Escape_Key_Handler.cs *
* Version         1.0                   *
* Summary         This script is responsible *
*                 for exiting the application *
*                 when the ESC key is hit   *
*                 *                       *
* Created         2022-11-09 12:00:00    *
* Last modified   2022-11-17 13:30:00    *
*                 *                       *
*****/

using UnityEngine;

public class Escape_Key_Handler : MonoBehaviour
{
    /*****
    *
    * Method name      Update
    * Arguments        none
    * Return value     none
    * Summary         Update will be executed before
    *                 a frame gets rendered to the
    *                 screen
    *
    *                 If the method detects a hit
    *                 ESC key the app will be left
    *
    *****/

    void Update()
    {
        //
        // Check for Escape key before rendering a new
        // frame
        // Leave application if Escape key is pressed
        //

        if (Input.GetKey(KeyCode.Escape))
        {
            Application.Quit();
        }
    }
}

/*****
* End of file Escape_Key_Handler.cs      *
*****/

```

D. Customized ESRI Code

C# file "Portland_Map_Builder_Regular.cs"

```

/*****
*
* Project          TriMet_Portland_Digital_Twin *
* Supervisor      Christoph Traun            *
* Author          ESRI                       *
* Customizing     Winfried Schwan           *
* Participant ID  107023                    *
* Filename        Portland_Map_Builder_Regular.cs *

```

```

* Version          1.0          *
* Summary          This script is based on ESRI *
*                  sample code that has been   *
*                  provided under an Apache    *
*                  license, please see license *
*                  details in the comments below *
*                  *
*                  The code has been customized to *
*                  meet the project requirements *
*                  *
*                  This includes setting up a   *
*                  local scene of the Portland  *
*                  area with a hillshade layer and *
*                  transformed *.shp files which *
*                  had been prepared in ArcGIS Pro *
*                  and describe the TriMet network *
*                  The *.shp files had been trans- *
*                  formed into scene layer      *
*                  packages that are loaded    *
*                  locally from disk          *
*                  *
* Created          2022-08-02 09:30:00        *
* Last modified    2022-11-17 10:00:00        *
*                  *
*****/

// Copyright 2022 Esri.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at: http://www.apache.org/licenses/LICENSE-2.0
//
// ArcGISMapsSDK

using Esri.ArcGISMapsSDK.Components;
using Esri.ArcGISMapsSDK.Samples.Components;
using Esri.ArcGISMapsSDK.Utils.GeoCoord;

using Esri.GameEngine.Extent;
using Esri.GameEngine.Elevation;
using Esri.GameEngine.Geometry;
using Esri.GameEngine.Layers;
using Esri.GameEngine.Map;

using Esri.Unity;

using UnityEditor;
using UnityEngine;

using System;
using System.IO;

// This sample code demonstrates the essential API calls to set up an ArcGISMap
// It covers generation and initialization for the necessary ArcGISMapsSDK game objects
// and components

// Render-In-Editor Mode
// The ExecuteAlways attribute allows a script to run both in editor and during play mode
// You can disable the run-in-editor mode functions by commenting out this attribute
// and reloading the scene
// NOTE: Hot reloading changes to an editor script doesn't always work. You'll need to
// restart the scene if you want your code changes to take effect
// You could write an editor script to reload the scene for you, but that's beyond
// the scope of this sample script
// See the Unity Hot Reloading documentation to learn more about hot reloading:
// https://docs.unity3d.com/Manual/script-Serialization.html

```

```
[ExecuteAlways]
public class Portland_Map_Builder_Regular : MonoBehaviour
{
    private ArcGISMapComponent arcGISMapComponent;
    private ArcGISCameraComponent cameraComponent;

    //
    // WS Customizing
    //
    // Defined ESRI developer key
    //

    public string APIKey = "developerKeyProvidedByEsri";

    //
    // WS Customizing
    //
    // set up positions for map and camera
    //

    private ArcGISPoint originCoordinates =
        new ArcGISPoint(-122.6883204, 45.4639347, 0,
            ArcGISSpatialReference.WGS84());

    private ArcGISPoint cameraCoordinates =
        new ArcGISPoint(-122.679775, 45.057815, 84750.982019,
            ArcGISSpatialReference.WGS84());

    // This sample event is used in conjunction with a Sample3DAttributes component
    // It passes a layer to a listener to process its attributes
    // The Sample3DAttributes component is not required, so you are free to remove
    // this event and its invocations in both scripts
    // See ArcGISMapsSDK/Samples/Scripts/3DAttributesSample/Sample3DAttributesComponent.cs
    // for more info

    public delegate void SetLayerAttributesEventHandler(ArcGIS3DObjectSceneLayer layer);

    private void Start()
    {
        CreateArcGISMapComponent();
        CreateArcGISCamera();
        CreateViewStateLoggingComponent();
        CreateArcGISMap();
    }

    // The ArcGISMap component is responsible for setting the origin of the map
    // All geographically located objects need to be a parent of this object

    private void CreateArcGISMapComponent()
    {
        arcGISMapComponent = FindObjectOfType<ArcGISMapComponent>();

        if (!arcGISMapComponent)
        {
            GameObject arcGISMapGameObject = new GameObject("ArcGISMap");
            arcGISMapComponent =
                arcGISMapGameObject.AddComponent<ArcGISMapComponent>();
        }

        arcGISMapComponent.OriginPosition = originCoordinates;
        arcGISMapComponent.MapType = ArcGISMapType.Local;

        // To change the Map Type in editor, you can change the Map Type
        // property of the Map component
        // When you change the Map Type, this event will trigger a call to rebuild
    }
}
```

```
// the map
// We only want to subscribe to this event once after the necessary game
// objects are added to the scene

arcGISMapComponent.MapTypeChanged +=
    new ArcGISMapComponent.MapTypeChangedEventHandler(CreateArcGISMap);
}

// ArcGIS Camera and Location components are added to a Camera game object
// to enable map rendering, player movement and tile loading

private void CreateArcGISCamera()
{
    cameraComponent = Camera.main.gameObject.GetComponent<ArcGISCameraComponent>();

    if (!cameraComponent)
    {
        GameObject cameraGameObject = Camera.main.gameObject;

        // The Camera game object needs to be a child of the
        // Map View game object in order for it to be correctly
        // placed in the world

        cameraGameObject.transform.SetParent
            (arcGISMapComponent.transform, false);

        // We need to add an ArcGISCamera component

        cameraComponent =
            cameraGameObject.AddComponent<ArcGISCameraComponent>();

        // The Camera Controller component provides player movement
        // to the Camera game object

        cameraGameObject.AddComponent<ArcGISCameraControllerComponent>();

        // The Rebase component adjusts the world origin to account for 32 bit
        // floating point precision issues as the camera moves around the scene

        cameraGameObject.AddComponent<ArcGISRebaseComponent>();
    }

    ArcGISLocationComponent cameraLocationComponent =
        cameraComponent.GetComponent<ArcGISLocationComponent>();

    if (!cameraLocationComponent)
    {
        // We need to add an ArcGISLocation component...

        cameraLocationComponent =
            cameraComponent.gameObject.AddComponent<ArcGISLocationComponent>();

        // ...and update its position and rotation in geographic coordinates

        cameraLocationComponent.Position = cameraCoordinates;

        // rotate the camera as needed

        cameraLocationComponent.Rotation =
            new ArcGISRotation(359.727499, 32.388506, 359.999979);
    }
}

private void CreateViewStateLoggingComponent()
{
    ArcGISViewStateLoggingComponent viewStateComponent
```

```
        = arcGISMapComponent.GetComponent<ArcGISViewStateLoggingComponent>();

    if (!viewStateComponent)
    {
        viewStateComponent =
            ArcGISMapComponent.gameObject.AddComponent
                <ArcGISViewStateLoggingComponent>();
    }
}

// This function creates the actual ArcGISMap object that will use
// your data to create a map
// This is the only function from this script that will get called again
// when the map type changes

public void CreateArcGISMap()
{
    if (APIKey == "")
    {
        Debug.LogError
            ("An API Key must be set on the SampleAPIMapCreator for content to load");
    }

    // Create the Map Document
    // You need to create a new ArcGISMap whenever you change the map type

    ArcGISMap arcGISMap = new ArcGISMap(arcGISMapComponent.MapType);

    //
    // WS Customizing
    //
    // Using elevation as a basemap replacement
    // because the ESRI imagery that is available
    // as basemap does not meet a more stylized look
    //
    // Create the elevation layer
    //

    arcGISMap.Elevation = new ArcGISMapElevation
        (new ArcGISImageElevationSource
            ("https://elevation3d.arcgis.com/arcgis/rest/services/
            WorldElevation3D/Terrain3D/ImageServer", "Elevation", ""));

    //
    // WS Customizing
    //
    // Create ArcGIS layers based on *.slpk files prepared in ArcGIS Pro
    // then assign materials and add layers to map
    //

    ArcGISImageLayer hillshadeLayer = new ArcGISImageLayer
        ("https://ibasemaps-api.arcgis.com/arcgis/rest/services/Elevation/
        World_Hillshade_Dark/MapServer/", "Hillshade Layer", 1.0f, true, APIKey);
    arcGISMap.Layers.Add(hillshadeLayer);

    //
    // Add scene layer package describing the TriMet service boundaries
    // (polygon feature)
    //

    string tmBoundariesSlpkPath =
        Path.Combine(Application.streamingAssetsPath, "tm_boundary.slpk");

    ArcGIS3DObjectSceneLayer tmBoundariesLayer =
```

```
        new ArcGIS3DObjectSceneLayer(tmBoundariesSlpkPath, "TriMet Boundary Layer",
        1.0f, true, "");

tmBoundariesLayer.MaterialReference = new Material
    (Resources.Load<Material>("Materials/DarkGreyMat_Regular"));
arcGISMap.Layers.Add(tmBoundariesLayer);

//
// Add scene layer package describing all routes of the TriMet
// network (line feature)
//

string tmAllLinesSlpkPath =
    Path.Combine(Application.streamingAssetsPath, "tm_routes_ALL.slpk");

ArcGIS3DObjectSceneLayer tmAllLinesLayer =
    new ArcGIS3DObjectSceneLayer(tmAllLinesSlpkPath, "TriMet All Routes Layer",
    1.0f, true, "");

tmAllLinesLayer.MaterialReference = new Material
    (Resources.Load<Material>("Materials/BlueMat"));

arcGISMap.Layers.Add(tmAllLinesLayer);

//
// Add scene layer package describing MAX routes of the TriMet
// network only (line feature)
//

string tmMaxLinesSlpkPath =
    Path.Combine(Application.streamingAssetsPath, "tm_routes_MAX.slpk");

ArcGIS3DObjectSceneLayer tmMaxLinesLayer =
    new ArcGIS3DObjectSceneLayer(tmMaxLinesSlpkPath, "TriMet MAX Routes Layer",
    1.0f, true, "");

tmMaxLinesLayer.MaterialReference = new Material
    (Resources.Load<Material>("Materials/RedMat"));

arcGISMap.Layers.Add(tmMaxLinesLayer);

//
// Add scene layer package describing MAX stops of the TriMet
// network (point feature)
//

string tmMaxStopsSlpkPath =
    Path.Combine(Application.streamingAssetsPath, "tm_stops_MAX.slpk");

ArcGIS3DObjectSceneLayer tmMaxStopsLayer =
    new ArcGIS3DObjectSceneLayer(tmMaxStopsSlpkPath, "TriMet MAX Stops Layer",
    1.0f, true, "");

tmMaxStopsLayer.MaterialReference = new Material
    (Resources.Load<Material>("Materials/BlackMat"));

arcGISMap.Layers.Add(tmMaxStopsLayer);

// If the map type is local, we will create a rectangle extent
// and attach it to the map's clipping area

if (arcGISMap.MapType == ArcGISMapType.Local)
{
    ArcGISPoint extentCenter = new ArcGISPoint
        (-122.713204, 45.4639347, 0, ArcGISSpatialReference.WGS84());
```



```

        ArcGISExtentRectangle extent =
            new ArcGISExtentRectangle(extentCenter, 400000, 400000);

        try
        {
            arcGISMap.ClippingArea = extent;
        }
        catch (Exception exception)
        {
            Debug.Log(exception.Message);
        }
    }

    // We have completed setup and are ready
    // to assign the ArcGISMap object to the View

    arcGISMapComponent.View.Map = arcGISMap;

#if UNITY_EDITOR
    // The editor camera is moved to the position
    // of the Camera game object when the map type is changed in editor

    if (!Application.isPlaying && SceneView.lastActiveSceneView != null)
    {
        SceneView.lastActiveSceneView.pivot = cameraComponent.transform.position;
        SceneView.lastActiveSceneView.rotation = cameraComponent.transform.rotation;
    }
#endif
}

/*****
 * End of file Portland_Map_Builder_Regular.cs      *
 *****/

```

C# file "CustomArcGISCameraControllerComponent.cs"

```

/*****
 *
 * Project           TriMet_Portland_Digital_Twin    *
 * Supervisor       Christoph Traun                *
 * Author           ESRI                           *
 * Customizing      Winfried Schwan                 *
 * Participant ID   107023                          *
 * Filename         CustomArcGISCameraController   *
 *                 Component.cs                     *
 * Version          1.0                             *
 * Summary          This script is based on ESRI    *
 *                 code as part of the ArcGIS      *
 *                 Maps SDK for Unity               *
 *
 *                 The code has been customized to *
 *                 meet the project requirements    *
 *
 *                 This includes limiting camera    *
 *                 movements if they are beyond    *
 *                 the main area of interest that  *
 *                 is Portland                      *
 *                 Zooming out of the scene has    *
 *                 been also restricted as well as *
 *                 rotating the scene              *
 *                 I felt that removing these     *
 *                 features would help to keep     *
 *                 focuses on the area of interest *
 *
 *****/

```

```
* Created          2022-08-05 11:30:00      *
* Last modified   2022-08-17 10:00:00      *
*                                                         *
*****/

// COPYRIGHT 1995-2022 ESRI
// TRADE SECRETS: ESRI PROPRIETARY AND CONFIDENTIAL
// Unpublished material - all rights reserved under the
// Copyright Laws of the United States and applicable international
// laws, treaties, and conventions.
//
// For additional information, contact:
// Attn: Contracts and Legal Department
// Environmental Systems Research Institute, Inc.
// 380 New York Street
// Redlands, California 92373
// USA
//
// email: legal@esri.com

using Esri.ArcGISMapsSDK.Components;
using Esri.ArcGISMapsSDK.Utils.Math;
using Esri.GameEngine.Geometry;
using Esri.GameEngine.View;
using Esri.HPFramework;
using System;
using Unity.Mathematics;
using UnityEngine;
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem;
#endif

namespace Esri.ArcGISMapsSDK.Samples.Components
{
    [DisallowMultipleComponent]
    [RequireComponent(typeof(HPTransform))]

    //
    // WS Customizing
    //
    // Renamed component to "Custom ArcGIS Camera Controller"
    // making it easier to recognize if the modified
    // camera controller is in use

    [AddComponentMenu("ArcGIS Maps SDK/Samples/Custom ArcGIS Camera Controller")]

    public class CustomArcGISCameraControllerComponent : MonoBehaviour
    {
        private ArcGISMapComponent arcGISMapComponent;
        private HPTransform hpTransform;

#if ENABLE_INPUT_SYSTEM
        public ArcGISCameraControllerComponentActions CameraActions;
        private InputAction UpControls;
        private InputAction ForwardControls;
        private InputAction RightControls;
#endif

        private float TranslationSpeed = 0.0f;
        private float RotationSpeed = 100.0f;
        private double MouseScrollSpeed = 0.1f;

        private static double MaxCameraHeight = 11000000.0;
        private static double MinCameraHeight = 1.8;
        private static double MaxCameraLatitude = 85.0;

        private double3 lastCartesianPoint = double3.zero;
        private ArcGISPoint lastArcGISPoint =
```

```
        new ArcGISPoint(0, 0, 0, ArcGISSpatialReference.WGS84());
private double lastDotVC = 0.0f;
private bool firstDragStep = true;

private Vector3 lastMouseScreenPosition;
private bool firstOnFocus = true;

public double MaxSpeed = 2000000.0;
public double MinSpeed = 1000.0;

private double3 savedCameraPosition;

private void Awake()
{
    lastMouseScreenPosition = GetMousePosition();

    Application.focusChanged += FocusChanged;

#if ENABLE_INPUT_SYSTEM
    CameraActions = new ArcGISCameraControllerComponentActions();
    UpControls = CameraActions.Move.Up;
    ForwardControls = CameraActions.Move.Forward;
    RightControls = CameraActions.Move.Right;
#endif
}

void OnEnable()
{
    arcGISMapComponent = gameObject.GetComponentInParent<ArcGISMapComponent>();
    hpTransform = GetComponent<HPTransform>();

#if ENABLE_INPUT_SYSTEM
    UpControls.Enable();
    ForwardControls.Enable();
    RightControls.Enable();
#endif
}

private void OnDisable()
{
#if ENABLE_INPUT_SYSTEM
    UpControls.Disable();
    ForwardControls.Disable();
    RightControls.Disable();
#endif
}

private Vector3 GetMousePosition()
{
#if ENABLE_INPUT_SYSTEM
    return Mouse.current.position.ReadValue();
#else
    return Input.mousePosition;
#endif
}

private double3 GetTotalTranslation()
{
    var forward = hpTransform.Forward.ToDouble3();
    var right = hpTransform.Right.ToDouble3();
    var up = hpTransform.Up.ToDouble3();

    var totalTranslation = double3.zero;

#if ENABLE_INPUT_SYSTEM
    up *= UpControls.ReadValue<float>() * TranslationSpeed * Time.deltaTime;
    right *= RightControls.ReadValue<float>() * TranslationSpeed * Time.deltaTime;
#endif
}
```

```
        forward *= ForwardControls.ReadValue<float>() * TranslationSpeed * Time.deltaTime;
        totalTranslation += up + right + forward;
#else

        Action<string, double3> handleAxis = (axis, vector) =>
        {
            if (Input.GetAxis(axis) != 0)
            {
                totalTranslation += vector * Input.GetAxis(axis) * TranslationSpeed *
Time.deltaTime;
            }
        };

        handleAxis("Vertical", forward);
        handleAxis("Horizontal", right);
        handleAxis("Jump", up);
        handleAxis("Submit", -up);
#endif

        return totalTranslation;
    }

    private float GetMouseScrollValue()
    {
#if ENABLE_INPUT_SYSTEM
        return Mouse.current.scroll.ReadValue().y;
#else
        return Input.mouseScrollDelta.y;
#endif
    }

    private bool IsMouseLeftClicked()
    {
#if ENABLE_INPUT_SYSTEM
        return Mouse.current.leftButton.ReadValue() == 1;
#else
        return Input.GetMouseButton(0);
#endif
    }

    private bool IsMouseRightClicked()
    {
#if ENABLE_INPUT_SYSTEM
        return Mouse.current.rightButton.ReadValue() == 1;
#else
        return Input.GetMouseButton(1);
#endif
    }

    void Start()
    {
        if (arcGISMapComponent == null)
        {
            Debug.LogError("An ArcGISMapComponent could not be found.
                Please make sure this GameObject is a child of a GameObject
                with an ArcGISMapComponent attached");

            enabled = false;
            return;
        }
    }

    void Update()
    {
        if (arcGISMapComponent == null)
        {
            return;
        }
    }
}
```

```

    if (arcGISMapComponent.View.SpatialReference == null)
    {
        // Not functional until we have a spatial reference
        return;
    }

    DragMouseEvent();

    UpdateNavigation();
}

/// <summary>
/// Move the camera based on user input
/// </summary>
private void UpdateNavigation()
{
    var altitude = arcGISMapComponent.View.AltitudeAtCartesianPosition(Position);
    UpdateSpeed(altitude);

    var totalTranslation = GetTotalTranslation();

    if (GetMouseScollValue() != 0.0)
    {
        var towardsMouse = GetMouseRayCastDirection();
        var delta = Math.Max(1.0, (altitude - MinCameraHeight))
            * MouseScrollSpeed * GetMouseScollValue();
        totalTranslation += towardsMouse * delta;
    }

    if (!totalTranslation.Equals(double3.zero))
    {
        MoveCamera(totalTranslation);
    }
}

/// <summary>
/// Move the camera
/// </summary>
private void MoveCamera(double3 movDir)
{
    var distance = math.length(movDir);
    movDir /= distance;

    var cameraPosition = Position;
    var cameraRotation = Rotation;

    if (arcGISMapComponent.MapType == GameEngine.Map.ArcGISMapType.Global)
    {
        var spheroidData = arcGISMapComponent.View.SpatialReference.SpheroidData;
        var nextArcGISPoint = arcGISMapComponent.View.WorldToGeographic
            (movDir + cameraPosition);

        if (nextArcGISPoint.Z > MaxCameraHeight)
        {
            var point = new ArcGISPoint(nextArcGISPoint.X, nextArcGISPoint.Y,
                MaxCameraHeight, nextArcGISPoint.SpatialReference);
            cameraPosition = arcGISMapComponent.View.GeographicToWorld(point);
        }
        else if (nextArcGISPoint.Z < MinCameraHeight)
        {
            var point = new ArcGISPoint(nextArcGISPoint.X, nextArcGISPoint.Y,
                MinCameraHeight, nextArcGISPoint.SpatialReference);
            cameraPosition = arcGISMapComponent.View.GeographicToWorld(point);
        }
        else
        {

```

```
        cameraPosition += movDir * distance;
    }

    var newENUReference = arcGISMapComponent.View.GetENUReference(cameraPosition);
    var oldENUReference = arcGISMapComponent.View.GetENUReference(Position);

    cameraRotation = math.mul(math.inverse(oldENUReference.GetRotation()),
        cameraRotation);
    cameraRotation = math.mul(newENUReference.GetRotation(), cameraRotation);
}
else
{
    cameraPosition += movDir * distance;

    //
    // WS Customizing
    //
    // prevent camera from zooming out beyond
    // a defined value for cameraPosition.x

    if (cameraPosition.y > 80000.0)
    {
        cameraPosition.x = savedCameraPosition.x;
        cameraPosition.z = savedCameraPosition.z;

        cameraPosition.y = 80000.0;

        savedCameraPosition = cameraPosition;
    }
    else
    {
        savedCameraPosition = cameraPosition;
    }
}

Position = cameraPosition;
Rotation = cameraRotation;
}

void OnTransformParentChanged()
{
    OnEnable();
}

private void DragMouseEvent()
{
    var cartesianPosition = Position;
    var cartesianRotation = Rotation;

    var deltaMouse = GetMousePosition() - lastMouseScreenPosition;

    if (!firstOnFocus)
    {
        if (IsMouseLeftClicked())
        {
            if (deltaMouse != Vector3.zero)
            {
                if (arcGISMapComponent.MapType == GameEngine.Map.ArcGISMapType.Global)
                {
                    GlobalDragging(ref cartesianPosition, ref cartesianRotation);
                }
                else if (arcGISMapComponent.MapType ==
                    GameEngine.Map.ArcGISMapType.Local)
                {
                    LocalDragging(ref cartesianPosition);
                }
            }
        }
    }
}
```

```

else if (IsMouseRightClicked())
{
    if (!deltaMouse.Equals(Vector3.zero))
    {
        //
        // WS Customizing
        //

        // It's not desirable to have map rotation enabled

        // RotateAround(ref cartesianPosition, ref cartesianRotation,
        //     deltaMouse);
    }
}
else
{
    firstDragStep = true;
}
}
else
{
    firstOnFocus = false;
}

//
// WS Customizing
//
// It's not desirable to have map panning
// beyond reasonable values

if (cartesianPosition.x > -13718136.5779012 &&
    cartesianPosition.x < -13591647.8562627 &&
    cartesianPosition.z < 5700730.93491859 &&
    cartesianPosition.z > 5623991.4877625)
{
    Position = cartesianPosition;
}

Rotation = cartesianRotation;

lastMouseScreenPosition = GetMousePosition();
}

private void LocalDragging(ref double3 cartesianPosition)
{
    var worldRayDir = GetMouseRayCastDirection();
    var isIntersected = Geometry.RayPlaneIntersection(cartesianPosition, worldRayDir,
        double3.zero, math.up(), out var intersection);

    if (isIntersected && intersection >= 0)
    {
        double3 cartesianCoord = cartesianPosition + worldRayDir * intersection;

        var delta = firstDragStep ? double3.zero : lastCartesianPoint -
            cartesianCoord;

        lastCartesianPoint = cartesianCoord + delta;
        cartesianPosition += delta;
        firstDragStep = false;
    }
}

private void GlobalDragging(ref double3 cartesianPosition,
    ref quaternion cartesianRotation)
{
    var spheroidData = arcGISMapComponent.View.SpatialReference.SpheroidData;
    var worldRayDir = GetMouseRayCastDirection();
    var isIntersected = Geometry.RayEllipsoidIntersection(spheroidData,

```

```

        cartesianPosition, worldRayDir, 0, out var intersection);
    if (isIntersected && intersection >= 0)
    {
        var oldENUReference = arcGISMapComponent.View.GetENUReference
            (cartesianPosition);

        var geoPosition = arcGISMapComponent.View.WorldToGeographic
            (cartesianPosition);

        double3 cartesianCoord = cartesianPosition + worldRayDir * intersection;
        var currentGeoPosition = arcGISMapComponent.View.WorldToGeographic
            (cartesianCoord);

        var visibleHemisphereDir = math.normalize(arcGISMapComponent.View.
            GeographicToWorld (new ArcGISPoint(geoPosition.X, 0, 0,
            geoPosition.SpatialReference)));

        double dotVC = math.dot(cartesianCoord, visibleHemisphereDir);
        lastDotVC = firstDragStep ? dotVC : lastDotVC;

        double deltaX = firstDragStep ? 0 : lastArcGISPoint.X - currentGeoPosition.X;
        double deltaY = firstDragStep ? 0 : lastArcGISPoint.Y - currentGeoPosition.Y;

        deltaY = Math.Sign(dotVC) != Math.Sign(lastDotVC) ? 0 : deltaY;

        lastArcGISPoint = new ArcGISPoint(currentGeoPosition.X + deltaX,
            currentGeoPosition.Y + deltaY, lastArcGISPoint.Z,
            lastArcGISPoint.SpatialReference);

        var YVal = geoPosition.Y + (dotVC <= 0 ? -deltaY : deltaY);
        YVal = Math.Abs(YVal) < MaxCameraLatitude ? YVal : (YVal > 0 ?
            MaxCameraLatitude : -MaxCameraLatitude);

        geoPosition = new ArcGISPoint(geoPosition.X + deltaX, YVal, geoPosition.Z,
            geoPosition.SpatialReference);

        cartesianPosition = arcGISMapComponent.View.GeographicToWorld(geoPosition);

        var newENUReference = arcGISMapComponent.View.GetENUReference
            (cartesianPosition);
        cartesianRotation = math.mul(math.inverse(oldENUReference.GetRotation()),
            cartesianRotation);
        cartesianRotation = math.mul(newENUReference.GetRotation(),
            cartesianRotation);

        firstDragStep = false;
        lastDotVC = dotVC;
    }
}

private void RotateAround(ref double3 cartesianPosition,
    ref quaternion cartesianRotation, Vector3 deltaMouse)
{
    var ENUReference = arcGISMapComponent.View.GetENUReference
        (cartesianPosition).ToMatrix4x4();

    Vector2 angles;

    angles.x = deltaMouse.x / (float)Screen.width * RotationSpeed;
    angles.y = deltaMouse.y / (float)Screen.height * RotationSpeed;

    angles.y = Mathf.Min(Mathf.Max(angles.y, -90.0f), 90.0f);

    var right = Matrix4x4.Rotate(cartesianRotation).GetColumn(0);

```



```

    var rotationY = Quaternion.AngleAxis(angles.x, ENUReference.GetColumn(1));
    var rotationX = Quaternion.AngleAxis(-angles.y, right);

    cartesianRotation = rotationY * rotationX * cartesianRotation;
}

private double3 GetMouseRayCastDirection()
{
    var forward = hpTransform.Forward.ToDouble3();
    var right = hpTransform.Right.ToDouble3();
    var up = hpTransform.Up.ToDouble3();

    var camera = gameObject.GetComponent<Camera>();

    var view = new double4x4
    (
        math.double4(right, 0),
        math.double4(up, 0),
        math.double4(forward, 0),
        math.double4(double3.zero, 1)
    );

    var proj = camera.projectionMatrix.inverse.ToDouble4x4();

    proj.c2.w *= -1;
    proj.c3.z *= -1;

    var MousePosition = GetMousePosition();
    double3 ndcCoord = new double3(2.0 * (MousePosition.x / Screen.width)
        - 1.0, 2.0 * (MousePosition.y / Screen.height) - 1.0, 1);
    double3 viewRayDir = math.normalize(proj.HomogeneousTransformPoint(ndcCoord));
    return view.HomogeneousTransformVector(viewRayDir);
}

private void FocusChanged(bool isFocus)
{
    firstOnFocus = true;
}

private void UpdateSpeed(double height)
{
    var msMaxSpeed = (MaxSpeed * 1000) / 3600;
    var msMinSpeed = (MinSpeed * 1000) / 3600;
    TranslationSpeed = (float)(Math.Pow(Math.Min((height / 100000.0), 1), 2.0)
        * (msMaxSpeed - msMinSpeed) + msMinSpeed);
}

#region Properties
/// <summary>
/// Get/set the camera position in world coordinates
/// </summary>
private double3 Position
{
    get
    {
        return hpTransform.UniversePosition;
    }
    set
    {
        hpTransform.UniversePosition = value;
    }
}

/// <summary>
/// Get/set the camera rotation
/// </summary>
private quaternion Rotation
{

```

```
        get
        {
            return hpTransform.UniverseRotation;
        }
        set
        {
            hpTransform.UniverseRotation = value;
        }
    }

    #endregion
}

/*****
* End of file CustomArcGISCameraControllerComponent.cs *
*****/
```

Abbreviations

API	Application Programming Interface
AR	Augmented Reality
CAD	Computer Aided Design
CEST	Central European Summer Time
CET	Central European Time
CLR	Common Language Runtime
CPS	Cyber Physical System
CSV	Comma Separated Value
DDR	Double Data Rate
EPSG	European Petroleum Survey Group
ESRI	Environmental Systems Research Institute
FIPS	Federal Information Processing System
FPS	Frames Per Second
GIS	Geographic Information System
GPU	Graphics Processing Unit
GTFS	General Transit Feed Specification
GUI	Graphical User Interface
HARN	High Accuracy Reference Network
HDRP	High Definition Render Pipeline
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
MBTA	Massachusetts Bay Transportation Authority
MDN	Mozilla Developer Network
MIT	Massachusetts Institute of Technology
MS-DOS	Microsoft Disk Operating System
MTBF	Mean Time Between Failures
NAD	North American Datum
NASA	National Aeronautics and Space Administration
PLM	Product Lifecycle Management
PST	Pacific Standard Time
RAM	Random Access Memory
REST	Representational State Transfer
SDK	Software Development Kit
SLPK	Scene Layer Package
SQL	Structured Query Language
SSD	Solid State Drive
TriMet	Tri-County Metropolitan Transportation District of Oregon
UI	User Interface
URL	Uniform Resource Locator
URP	Universal Render Pipeline
UWP	Universal Windows Platform
VR	Virtual Reality
WGS	World Geodetic System
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

References

- BITC 2022A (REALTIME BITCOIN GLOBE): Real-time Bitcoin Globe created by Mike van Rossum as part of the “Experiments with Google” collection <<https://blocks.wizb.it/>> accessed on 2022/20/11.
- EDGE 2022A (EDGE INDUSTRY REVIEW): Using digital twins to improve operations at Vancouver’s airport documentation <<https://www.edgeir.com/using-digital-twins-to-improve-operations-at-vancouver-airport-20221019>> accessed on 2022/20/11.
- ESRI 2021A (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): Session “Introduction to ArcGIS Maps SDK for Game Engines” at 2021 ESRI Developer Summit, Video Position 2:50 <<https://www.youtube.com/watch?v=VWbuSmjNSv0>> accessed on 2022/14/10.
- ESRI 2021B (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): Session “Introduction to ArcGIS Maps SDK for Game Engines” at 2021 ESRI Developer Summit, Video Position 4:00 <<https://www.youtube.com/watch?v=VWbuSmjNSv0>> accessed on 2022/14/10.
- ESRI 2022A (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): ArcGIS Maps SDK for Unity – Layers documentation <<https://developers.arcgis.com/unity/layers/>> accessed on 2022/16/09.
- ESRI 2022B (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): ArcGIS Maps SDK for Unity Overview <<https://developers.arcgis.com/unity/>> accessed on 2022/16/10.
- ESRI 2022C (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): ArcGIS Maps SDK for Unity – Spatial and data analysis documentation <<https://developers.arcgis.com/unity/spatial-and-data-analysis/>> accessed on 2022/17/10.
- ESRI 2022D (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): ArcGIS Maps SDK for Unity – Tutorial: Display a map with C# API <<https://developers.arcgis.com/unity/maps/tutorials/display-a-map-api/>> accessed on 2022/10/11.
- ESRI 2022E (ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE): ArcGIS Maps SDK for Unity – Camera Documentation <<https://developers.arcgis.com/unity/maps/camera/>> accessed on 2022/13/11.
- GITHUB 2022A (GITHUB SOURCE CODE REPOSITORY): Complete source code for Portland Digital Twin Overview <https://github.com/fussgaenger/Portland_Digital_Twin> accessed on 2022/24/11.
- GOOGLE 2022A: Google Transit APIs Overview <<https://developers.google.com/transit>> accessed on 2022/06/09.
- GOOGLE 2022B: GTFS Realtime Overview <<https://developers.google.com/transit/gtfs-realtime>> accessed on 2022/07/09.
- GOOGLE 2022C: GTFS Realtime Revision History <<https://developers.google.com/transit/gtfs-realtime/guides/revision-history>> accessed on 2022/10/09.

- GOOGLE 2022D: Protocol Buffers Overview <<https://developers.google.com/protocol-buffers/docs/overview>> accessed on 2022/10/09.
- HERE 2022A (HERE TECHNOLOGIES): HERE Destination Weather API Overview <https://developer.here.com/documentation/destination-weather/dev_guide/topics/guide.html> accessed on 2022/26/10.
- IEEE 2020A (INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS): Digital Twin: Enabling Technologies, Challenges and Open Research, Volume 8, 2020, Page 108964 <<https://ieeexplore.ieee.org/document/9103025>> accessed on 2022/16/11.
- LG 2022A (LOOKING GLASS FACTORY): Homepage of Looking Glass Factory <<https://lookingglassfactory.com/>> accessed on 2022/13/09.
- MBTA 2022A (MASSACHUSETTS BAY TRANSPORTATION AUTHORITY): MBTA GTFS Realtime Documentation <<https://github.com/mbta/gtfs-documentation/blob/master/reference/gtfs-realtime.md>> accessed on 2022/10/09.
- MDN 2022A (MOZILLA DEVELOPER NETWORK): Overview HTTP Response Status Code 418 “I’m a teapot” <<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>> accessed on 2022/13/09.
- MDN 2022B (MOZILLA DEVELOPER NETWORK): Overview HTTP Response Status Codes <<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418>> accessed on 2022/13/09.
- MSFT 2022A (MICROSOFT): Microsoft Mixed Reality Documentation <<https://docs.microsoft.com/en-us/windows/mixed-reality/develop/choosing-an-engine?tabs=unity>> accessed on 2022/15/09.
- MSFT 2022B (MICROSOFT): Microsoft definition of Mixed Reality <<https://learn.microsoft.com/en-us/windows/mixed-reality/discover/mixed-reality>> accessed on 2022/25/09.
- MSFT 2022C (MICROSOFT): Overview Microsoft HoloLens <<https://www.microsoft.com/de-de/hololens?rtc=1>> accessed on 2022/13/10.
- NS 2022A (NEWTONSOFT): Overview Newtonsoft Json.NET Library <<https://www.newtonsoft.com/json>> accessed on 2022/27/10.
- PM 2022A (POSTMAN API PLATFORM): Overview Postman API Platform <<https://www.postman.com/>> accessed on 2022/13/09.
- PROC 2017A (PROCEDIA MANUFACTURING): A Review of the Roles of Digital Twin in CPS-based Production Systems, Volume 11, 2017, Pages 939-948 <<https://www.sciencedirect.com/science/article/pii/S2351978917304067>> accessed on 2022/15/11.
- RS 2022A (RESTSHARP LIBRARY): Overview RestSharp Library <<https://restsharp.dev/>> accessed on 2022/25/10.
- TRIMET 2022A (TRI-COUNTY METROPOLITAN TRANSPORTATION AGENCY): TriMet Mission Statement <<https://trimet.org/about/index.htm>> accessed on 2022/28/08.

- TRIMET 2022B (TRI-COUNTY METROPOLITAN TRANSPORTATION AGENCY): Section of TriMet's internet presence that supports developers with GTFS data <<https://developer.trimet.org/GTFS.shtml>> accessed on 2022/10/09.
- TRIMET 2022C (TRI-COUNTY METROPOLITAN TRANSPORTATION AGENCY): Section of TriMet's internet presence that supports developers with GIS data <<https://developer.trimet.org/gis/>> accessed on 2022/29/08.
- TRIMET 2022D (TRI-COUNTY METROPOLITAN TRANSPORTATION AGENCY): TriMet's Web Service Documentation <https://developer.trimet.org/ws_docs/> accessed on 2022/13/09.
- UNITY 2022A (UNITY TECHNOLOGIES): Unity Technologies Home Page <<https://unity.com/>> accessed on 2022/18/09.
- UNITY 2022B (UNITY TECHNOLOGIES): Supported platforms by Unity <<https://support.unity.com/hc/en-us/articles/206336795-What-platforms-are-supported-by-Unity->> accessed on 2022/13/10.
- UNITY 2022C (UNITY TECHNOLOGIES): Unity TextMesh Pro Documentation <<https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/index.html>> accessed on 2022/14/11.
- UNITY 2022D (UNITY TECHNOLOGIES): The what, why and how of digital twins, Unity e-book <<https://resources.unity.com/aec-content/what-is-a-digital-twin>> accessed on 2022/16/11.
- WIKIPEDIA 2022A: Overview of TriMet at Wikipedia <<https://en.wikipedia.org/wiki/TriMet>> accessed on 2022/28/08.
- WIKIPEDIA 2022B: Overview of Windows Presentation Foundation framework at Wikipedia <https://en.wikipedia.org/wiki/Windows_Presentation_Foundation> accessed on 2022/14/09.
- WIKIPEDIA 2022C: Overview of C# programming language at Wikipedia <[https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))> accessed on 2022/25/09.
- WORLDBANK 2022A (WORLD BANK GROUP): Open Learning Campus – Introduction to the General Transit Feed Specification (GTFS) and Informal Transit System Mapping <<https://olc.worldbank.org/content/introduction-the-general-transit-feed-specification-gtfs-and-informal-transit-system-mappi-0>> accessed on 2022/06/09.
- YOUTUBE 2022A (ALPHABET INC.): Video Portland Digital Twin “Vanilla” Edition <<https://youtu.be/09L1RJOhGCM>> accessed on 2022/23/11.
- YOUTUBE 2022B (ALPHABET INC.): Video Portland Digital Twin “Tippy” Edition <<https://youtu.be/KDzwME6ZzYI>> accessed on 2022/23/11.

Figures

Figure 1: Anaglyphic 3D glasses back in the 1970s...	8
Figure 2: How it all began – first rough sketch on architecture ideas	11
Figure 3: And how it turned out – random screenshot of final Unity application	11
Figure 4: My travel route across “Digital Twin Land”	15
Figure 5: Having fun with Indiana Jones typefaces – with tongue in cheek	15
Figure 6: Amount of Digital Twin research across different industries	20
Figure 7: Digital Twin of Vancouver International Airport.....	24
Figure 8: Digital Twin of Bitcoin mining activities.....	24
Figure 9: Digital Twin of Wind Turbine	25
Figure 10: Sample GTFS file routes.txt.....	26
Figure 11: Exemplary GTFS file relationship using a shared field	27
Figure 12: TriMet developer site GTFS data	32
Figure 13: How a stop_id can be resolved to a stop_name.....	34
Figure 14: TriMet developer site GIS data	35
Figure 15: Projecting to “WGS 1984 Web Mercator”	36
Figure 16: Feature layer TriMet Boundary.....	37
Figure 17: Feature layer TriMet Routes	37
Figure 18: Feature layer TriMet Stops	38
Figure 19: Separating MAX routes from all routes	39
Figure 20: Feature layer TriMet MAX Routes	39
Figure 21: Separating MAX stops from all stops.....	39
Figure 22: Feature layer TriMet MAX Stops.....	40
Figure 23: Postman workspace with TriMet collection	43
Figure 24: Postman workspace with sample request and response	43
Figure 25: Visual Studio project template for Console Application.....	45
Figure 26: Visual Studio project template for WPF App	45
Figure 27: C# script within the Unity Editor.....	46
Figure 28: Unity Editor and Visual Studio side by side.....	46
Figure 29: Layer tm_boundary in ArcGIS Pro.....	48
Figure 30: Attribute height in attribute table tm_boundary	48
Figure 31: Attribute based extrusion of layer tm_boundary.....	48
Figure 32: First 3D representation of layer tm_boundary.....	49
Figure 33: Using the tool “Feature To 3D By Attribute”	49
Figure 34: Confirming the Polygon Z shape type in attribute table.....	49
Figure 35: Creating a MultiPatch feature.....	50
Figure 36: Confirming the MultiPatch shape type in attribute table.....	50
Figure 37: Creating an offline 3D Scene Layer Package.....	50
Figure 38: Unity Editor with sample scene opened	54
Figure 39: Sphere GameObject in a Unity scene hierarchy	54
Figure 40: Sphere GameObject displayed in a Unity scene	55
Figure 41: Components attached to Sphere GameObject.....	55
Figure 42: Inspector view of C# script attached to Sphere GameObject.....	58
Figure 43: Sample C# script to let a Sphere GameObject rotate	58

Figure 44: The structure of a simple C program	59
Figure 45: How a real-time 3D engine implements a “game loop”	60
Figure 46: Sample Update() function.....	60
Figure 47: User interface of “Map Creator” in Unity Editor	65
Figure 48: Local scene and circle extent enabled by ArcGIS Map component.....	66
Figure 49: Digital twin application’s architecture overview	68
Figure 50: Leveraging the RestSharp library with C#.....	69
Figure 51: Building a request with the RestSharp library	70
Figure 52: Running a request with the RestSharp library	70
Figure 53: Leveraging the Json.NET library with C#.....	70
Figure 54: The Json.NET library hard at work with serializing/deserializing objects.....	71
Figure 55: GTFS Data Loader overview	72
Figure 56: GTFS Data Loader desktop shortcut	72
Figure 57: How to run the GTFS Data Loader	73
Figure 58: GTFS Data Loader configurable feed URL.....	73
Figure 59: GTFS Data Loader connection health check	73
Figure 60: GTFS Data Loader configurable query frequency.....	74
Figure 61: GTFS Data Loader folder configuration.....	74
Figure 62: GTFS Data Loader folder cleanup	75
Figure 63: Sample contents of “working” folder	75
Figure 64: UI elements of GTFS Data Loader with events attached to them	76
Figure 65: The inner workings of GTFS Data Loader.....	78
Figure 66: Working with the Unity Editor	79
Figure 67: Folder layout in Unity project	80
Figure 68: Subfolders in Scripts folder	80
Figure 69: Scene hierarchy in Unity project.....	81
Figure 70: Usage of empty GameObject in Unity project.....	81
Figure 71: ArcGISMap GameObject and its children	82
Figure 72: GameObjects responsible for GTFS data processing	83
Figure 73: User_Interface GameObject and its children	83
Figure 74: Scene headline and info panel	83
Figure 75: The inner workings of ProcessGtfsFiles().....	85
Figure 76: The Vehicle List as a means of sharing vehicle data across the Unity project.....	85
Figure 77: Resolving stop ids to stop names.....	86
Figure 78: Portland area as seen by the scene camera	87
Figure 79: The Portland scene and its coloured layers	89
Figure 80: Zooming out limited by camera controller	90
Figure 81: Panning restrictions by camera controller	90
Figure 82: Prefabs GameObject as a child of ArcGISMap	92
Figure 83: Scene info panel	93
Figure 84: Populating the info panel.....	94
Figure 85: Changing the scene title	95
Figure 86: Digital Twin “Vanilla” Edition	97
Figure 87: Digital Twin “Trippy” Edition	98
Figure 88: Random screenshot Digital Twin “Vanilla” Edition	98
Figure 89: Random screenshot Digital Twin “Trippy” Edition	99

Figure 90: Screenshot from video Digital Twin “Vanilla” Edition 100
Figure 91: Screenshot from video Digital Twin “Trippy” Edition 100
Figure 92: Movie credits from Lookout Mountain Laboratory 101

Tables

<i>Table 1: GTFS Schedule Mandatory Files</i>	28
<i>Table 2: GTFS Schedule Optional Files</i>	28
<i>Table 3: GTFS Realtime Feeds</i>	29
<i>Table 4: Feature layer TriMet Boundary</i>	36
<i>Table 5: Feature layer TriMet Routes</i>	37
<i>Table 6: Feature layer TriMet Stops</i>	38
<i>Table 7: Feature layer TriMet MAX Routes</i>	39
<i>Table 8: Feature layer TriMet MAX Stops</i>	40
<i>Table 9: Technical Data Development Laptop #1</i>	41
<i>Table 10: Technical Data Development Laptop #2</i>	41
<i>Table 11: UI Event related methods in MainWindow.xaml.cs</i>	76
<i>Table 12: GTFS Data Loader method overview</i>	77
<i>Table 13: Unity Editor views</i>	79
<i>Table 14: Script subfolders in Unity project</i>	80
<i>Table 15: Project folders in Unity project</i>	81