

UNIVERSITY OF CALIFORNIA  
SANTA CRUZ

**PRT-SIM: AN OPEN-SOURCE MICROSIMULATOR FOR  
PERSONAL RAPID TRANSIT SYSTEMS**

A project submitted in partial satisfaction of the  
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

**Daniel J. Homerick**

December 2010

The Project of Daniel J. Homerick  
is approved:

---

Professor Gabriel Elkaim, Chair

---

Professor James Davis

---

Professor Renwick Curry

---

Dean Tyrus Miller  
Vice Provost and Dean of Graduate Studies

Copyright © by  
Daniel J. Homerick  
2010

# Table of Contents

<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>viii</b>
<b>Abstract</b>	<b>ix</b>
<b>Dedication</b>	<b>x</b>
<b>Acknowledgments</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Personal Rapid Transit . . . . .	1
1.1.1 Motivation for PRT . . . . .	1
1.1.2 What is PRT? . . . . .	3
1.2 PRT Simulators . . . . .	5
1.2.1 Prior Work . . . . .	5
1.2.2 Motivation . . . . .	6
<b>2 TrackBuilder</b>	<b>8</b>
2.1 Introduction . . . . .	8
2.2 Usage . . . . .	9
2.3 Features . . . . .	9
2.3.1 Online Map & Elevation Data . . . . .	9
2.3.2 Curved Track Segments . . . . .	10
2.3.3 One-Way & Two-Way Track . . . . .	11
2.3.4 Automated Track Placement . . . . .	11
2.3.5 Undo . . . . .	12
2.3.6 Google Transit Feed Import . . . . .	12
2.4 Implementation . . . . .	13
2.4.1 Language and Framework . . . . .	13
2.4.2 Laying Track . . . . .	14

2.4.3	Undo . . . . .	17
2.4.4	Track Preview . . . . .	19
2.4.5	Creating Stations . . . . .	19
2.4.6	Creating Vehicles . . . . .	21
2.4.7	Creating Passengers . . . . .	21
2.4.8	Google Transit Feed Import . . . . .	22
2.4.9	Saving and Loading Files . . . . .	27
<b>3</b>	<b>Simulator</b>	<b>28</b>
3.1	Introduction . . . . .	28
3.2	Level of Detail . . . . .	29
3.3	Implementation . . . . .	30
3.3.1	Separation of Simulation and Controller . . . . .	30
3.3.2	Hybrid Simulator . . . . .	30
3.3.3	Simulation Time . . . . .	31
3.3.4	Vehicles . . . . .	32
3.3.5	Stations . . . . .	34
3.3.6	Passengers . . . . .	36
3.3.7	Statistics . . . . .	37
3.3.8	Energy Usage . . . . .	39
<b>4</b>	<b>Controllers</b>	<b>42</b>
4.1	Overview . . . . .	42
4.2	PRT Controller . . . . .	44
4.2.1	Overview . . . . .	44
4.2.2	Usage . . . . .	44
4.2.3	Routing . . . . .	45
4.2.4	Empty Vehicle Management . . . . .	46
4.2.5	Merging . . . . .	47
4.2.6	Station Behavior . . . . .	55
4.3	GTF Controller . . . . .	57
4.3.1	Vehicle Movement . . . . .	58
4.3.2	Passenger Routing . . . . .	60
<b>5</b>	<b>Trajectory Generation</b>	<b>63</b>
5.1	Splines . . . . .	64
5.1.1	Constraint Equations . . . . .	65
5.2	System of Equations . . . . .	66
5.3	Trajectory Procedures . . . . .	68
5.4	Target Acceleration . . . . .	69
5.5	Target Velocity . . . . .	70
5.6	Target Position . . . . .	73
5.7	Target Time . . . . .	77

<b>6</b>	<b>Discussion</b>	<b>82</b>
6.1	Future Work . . . . .	82
6.1.1	Spiral Curves . . . . .	82
6.1.2	LQR Trajectory Generation . . . . .	83
6.1.3	Safety Checks . . . . .	84
6.1.4	Nonlinear Stations . . . . .	84
6.1.5	Simulated Latency . . . . .	85
6.1.6	Noise and Error . . . . .	86
6.1.7	Multi-lane tracks . . . . .	86
6.1.8	Group Rapid Transit Controller . . . . .	87
6.2	Postmortem . . . . .	87
6.2.1	What Went Right . . . . .	87
6.2.2	What Went Wrong . . . . .	88
6.3	Final Conclusions . . . . .	90
<b>A</b>	<b>Scenario File XML Schema</b>	<b>91</b>
<b>B</b>	<b>Communication</b>	<b>104</b>
B.1	Messsages . . . . .	105
<b>C</b>	<b>Trajectory Calculations</b>	<b>110</b>
	<b>Bibliography</b>	<b>114</b>

# List of Figures

2.1	TrackBuilder’s main window. . . . .	10
2.2	Cutting a Corner . . . . .	15
2.3	An S-Curve created using two constant radius curve segments. . . . .	16
2.4	An intersection of two-way tracks. . . . .	18
3.1	A Station icon within the Simulator. . . . .	35
3.2	Vehicle Report . . . . .	37
3.3	Passenger Report . . . . .	38
3.4	Station Report . . . . .	38
3.5	Power & Energy Report . . . . .	39
4.1	Extent of a Merge’s Zone of Control. . . . .	48
4.2	Merge Slots within a Merge’s Reservation Queue . . . . .	49
4.3	State machine used to govern vehicle behavior within a station. . . . .	56
4.4	Time-Expanded Digraph . . . . .	61
5.1	A spline targeting an acceleration . . . . .	70
5.2	A three polynomial spline targeting a velocity . . . . .	71
5.3	A two segment spline targeting a velocity value . . . . .	72
5.4	A seven segment spline targeting a position . . . . .	73
5.5	A five segment spline targeting a position . . . . .	75
5.6	A seven segment spline targeting a time . . . . .	78
5.7	A seven segment spline targeting a time, with reduced acceleration limit. . . . .	80
A.1	Scenario, the root element. . . . .	91
A.2	TrackSegmentType . . . . .	93
A.3	CoordinateType . . . . .	93
A.4	StationModelType . . . . .	94
A.5	StorageType . . . . .	95
A.6	PlatformType and BerthType . . . . .	95
A.7	StationType . . . . .	96

A.8 VehicleModelType . . . . .	97
A.9 LimitType . . . . .	97
A.10 VehicleType . . . . .	98
A.11 WaypointType . . . . .	99
A.12 MergeType . . . . .	99
A.13 SwitchType . . . . .	99
A.14 MetadataType . . . . .	100
A.15 ImageType . . . . .	100
A.16 WeatherType . . . . .	101
A.17 GoogleTransitFeedType . . . . .	102

# List of Tables

2.1	GTFS files and fields used. . . . .	22
4.1	Station’s vehicle demand under various conditions. . . . .	46
5.1	Number of unknowns and constraints. . . . .	67
B.1	16-byte Message Header . . . . .	106
B.2	Controller Message Types . . . . .	107
B.3	Simulator Message Types – Part 1 of 2 . . . . .	108
B.4	Simulator Message Types – Part 2 of 2 . . . . .	109



## **Abstract**

prt-sim: An open-source microsimulator for Personal Rapid Transit systems

by

Daniel J. Homerick

The prt-sim project is an open-source, cross-platform simulator for Personal Rapid Transit (PRT). PRT is a form of mass transit that uses small, driverless vehicles that operate on-demand, rather than following fixed schedules and routes. PRT vehicles use light-weight, often elevated guideways, which offer a variety of potential advantages. The project includes a GUI-based tool for creating scenarios which incorporates Google Maps data. The simulator is separated from the transit system control, allowing it to be used as a test-bed for a variety of control concepts. The project includes two transit system controllers, one which implements a PRT service model, and another which implements conventional mass transit service. The project is available online at: <http://code.google.com/p/prt-sim/>

Dedicated to all those who have ever asked,

“Why isn’t *the Future* here yet?

Where are our flying cars, our personal robots, our moonbases?”

and then set out to make them all happen, one step at a time.

## Acknowledgments

This project was partially funded by a Center for Information Technology Research in the Interest of Society (CITRIS) Seed Grant #33 “Personal Rapid Transit Energy and Traffic Simulator.”

I want to thank Robert Baertsch for his unwavering support of the project, and his countless bug reports over the course of its development. His feedback has been vital to the shaping of the project, and has always been profoundly helpful.

I’d like to thank my advisor Gabriel Elkaim for his enthusiasm, his sage advice, and his inexhaustible supply of stories that provide context and insight into the wonderful world of engineering.

Thank you also to Ren Curry for stepping in to lead the Autonomous Systems Lab during Gabe’s sabbatical, and for his steady assistance and guidance before and since.

I would like to extend profound thanks to James Davis, for responding to this grad student’s last-minute call for additions to the reading committee.

Finally, thank you to all of the students that have been a part of the Autonomous Systems Lab during my stay. Good luck with the Overbot, and don’t forget to bring the rubber chicken.

# Chapter 1

## Introduction

### 1.1 Personal Rapid Transit

#### 1.1.1 Motivation for PRT

Personal rapid transit (PRT) is an unconventional form of mass transit that avoids many of the drawbacks that exist with traditional public mass transit. Conventional mass transit is characterized by vehicles which travel on a fixed schedule, on a fixed route, and which are shared amongst people with different destinations. Those three traits lead to conventional mass transit being both slow and inconvenient to use, especially in comparison with its principle competitor, the automobile.

Fixed schedule operation forces riders to align their schedules with the transit's schedule, and to suffer waiting periods when they can't. Fixed schedule also creates a high penalty for tardiness—two minutes late with a car may mean being two minutes

late to work, but two minutes late with a bus may cause delays of an hour or more. Even when tardiness is avoided, a fixed-schedule vehicle requires the rider to allocate extra time before every trip as insurance against the risk. Additionally, the transit's schedule must be compatible for both the outbound and return legs of the trip. If the timing of the return trip is uncertain, using mass transit carries the additional risk of stranding the rider for long periods, even overnight.

While fixed schedule operation often lengthens the waiting time, fixed route operation lengthens the riding time, and drastically reduces the number of accessible destinations. For trips that don't happen to align with a transit corridor, a fixed-route system will usually require a trip to a hub station and a transfer. Whenever a transfer is required it introduces additional waiting, and adds undesired complexity to the trip. The need for transfers also reduces flexibility for schedule planners, and can exacerbate problems caused by inevitable delays and by the vehicles getting off-schedule.

Large, shared vehicles in which people do not share the same destination will stop frequently as they service multiple destinations along their route. The periods in which a vehicle is stopped are very detrimental to average speed, even when the in-route speed is quite high. The starting and stopping behavior also decreases passenger comfort and interferes with reading or other attempts at productivity. Shared vehicles also increase the difficulty for passengers who wish to bring groceries or any other such luggage with them on their trip.

The worst of the problems caused by fixed schedules, fixed routes, and shared vehicles can be overcome by providing frequent service, densely covering the service

area with interconnected routes, and including express-service vehicles in the mix. This approach to improving quality of service, however, incurs a prohibitively high operating expense as each additional service route requires both an additional vehicle and driver.

### **1.1.2 What is PRT?**

PRT is a system of small, driverless vehicles, often called pods or podcars. Podcars are light-weight, and run on their own network of “guideways”. Podcars run on demand, do not require transfers, and carry only a small number of passengers<sup>1</sup>. PRT’s use of many small vehicles, made feasible by virtue of their autonomous operation, allows PRT to avoid the problematic traits of conventional mass transit described in the previous section.

PRT retains, and even enhances, many of the good characteristics of mass transit. It allows travel time to be reclaimed for productive and leisure tasks, and it offers mobility to those who don’t wish to or cannot drive, an important consideration as most societies’ average age climbs higher. PRT can readily be made available 24 hours a day due to its low staffing requirements. Most implementations of PRT are electric, and are quieter, less polluting, and more energy efficient than the automobile. Most importantly, PRT has the potential to be vastly safer than the automobile.

---

<sup>1</sup>Typically 2-6 passengers, but larger vehicles are occasionally still considered to be PRT.

## Guideways

What separates a PRT system from autonomous cars is that podcars do not attempt to mix with manually driven vehicles on existing roads. Instead, podcars use “guideways” which are separated from other traffic. Guideways can be rail-like or road-like, elevated or ground-level (or even underground). Vehicles may ride on top, or be suspended underneath, or both at once[2]. By removing the requirement to support heavy vehicles, such as fully loaded trucks, buses, or train cars, an elevated guideway can be vastly smaller, lighter, and cheaper than elevated structures for automobiles or conventional mass transit.

Guideways bring a host of advantages, as well as some not-insignificant disadvantages. The more structured an environment, the easier it is to do automation. Guideways, especially elevated guideway, can be designed to exclude most “unpredictable” things; manually driven vehicles for one, but also pedestrians and wildlife. They also allow for additional infrastructure that just doesn’t exist with roads, such as fiduciary marks indicating position, communications infrastructure, and power delivery to vehicles. For a given level of performance, guideways allow podcars to be simpler in terms of their design, their safety testing, and their manufacture.

The use of guideways brings the clear disadvantage that it’s harder to grow the system swiftly. Like any transit system, the value of a PRT system increases as it connects more destinations, but installing any kind of new infrastructure, especially in urban areas, is not known for being a fast process.

## Related Ideas

PRT is one niche within a cluster of ideas. If we relax the idea that vehicles must always run on guideways, then we get a concept called “Dual Mode” in which vehicles can operate autonomously while on a guideway, but which may also leave the guideway to be driven manually. If we relax the idea that vehicles aren’t shared, then we get something called “Group Rapid Transit” (GRT) which uses somewhat larger vehicles that make multiple stops to pick up and drop off passengers, but which are still demand-responsive for their scheduling and routing<sup>2</sup>. If we remove the concept of using a guideway entirely, then we have autonomous cars running a taxi-like service.

Both PRT and GRT (and to a lesser extent, Dual Mode) fall under the umbrella terms Automated Guideway Transit (AGT) and Automated People Movers (APM).

## 1.2 PRT Simulators

### 1.2.1 Prior Work

More than 30 simulators for PRT have been written since the PRT concept first gained interest in the late 1960’s. The information in this section has been partially informed by Jerry Schneider’s list of PRT simulators<sup>3</sup>. For this paper, PRT simulations are separated into the following broad categories: (i) Unmaintained (ii) Private (iii) Commercially available (iv) Freely available, closed source (v) Open source

---

<sup>2</sup>The “Morgantown Personal Rapid Transit” which connects the campuses of West Virginia University is generally considered to be a GRT, despite its name.

<sup>3</sup><http://faculty.washington.edu/jbs/itrans/simu.htm>



The unmaintained category is, in most cases, synonymous with unavailable. It contains a large number of old simulators, including some that were written for now long-gone machine architectures and operating systems[3]. It also includes newer simulators, often academic projects, that though they may still have promise of being resumed are not maintained at this time[12, 13, 11].

The private category contains simulators that are not publically available. Although some may be available under special circumstances, they are primarily intended for use by their owners. The implementation-specific simulators for the Vectus, UL-Tra, and 2GetThere PRT systems are in this category, as well as Taxi2000’s TrakEdit. PRT Consulting Inc and the engineering firm Kimley-Horn each have their own private simulators capable of modeling PRT systems as well.

Of commercially available simulators, I am only aware of three: Logistic Centrum’s PRTsim (no relation to this project) developed by Ingmar Andrasson, Beamway’s BeamEd PRO[4], and PRT International’s ITNS developed by J.E. Anderson.

Of the known simulators, the following are freely available but closed-source: Hermes[14], BeamEd (non-Pro)[4].

To my knowledge, this project is the only functioning, publically available, and open-source licensed PRT simulator.

### **1.2.2 Motivation**

This simulator has been designed from the start to decouple the simulation aspects from the system control aspects. This separation makes it straight-forward to

model different transit service modes using the same simulator and under the same assumptions. Programs implementing the system control logic (controllers) can be implemented in any programming language that has network support, and can even run on separate hardware. This project includes two example controllers, one that implements a PRT service mode, and another which implements the fixed schedule, fixed route, shared vehicle service mode of conventional mass transit.

The target audience for the simulator is a combination of users who want: (i) an open-source simulation that they can alter and customize; (ii) to compare the effectiveness of competing algorithms; (iii) an open-source system controller that they can alter to test solutions to subproblems such as empty vehicle allocation or congestion management; (iv) and users who want to simulate multimodal systems.

The company Unimodal have been using this simulator for in-house simulation work over the past year of their project's development.

## Chapter 2

# TrackBuilder

### 2.1 Introduction

TrackBuilder is intended to be a user-friendly graphical tool for creating simulation scenarios. It utilizes Google Maps to provide satellite, road, and terrain imagery to the user. The user constructs a scenario by laying out a track network, and adding stations and vehicles. The track network is laid out using a point and click interface which automatically uses a combination of straight and curved track pieces. TrackBuilder's save files (aka scenario files) may be used as input files for the Simulator.

TrackBuilder can also import files that adhere to the Google Transit Feed (GTF) specification. The GTF specification defines a common format for public transportation schedules and associated geographic information. From these files TrackBuilder will automatically lay out a track network, including stations and vehicles, and will include relevant vehicle scheduling information in its save file.

A variety of track, station, vehicle, and scenario attributes may be specified within TrackBuilder. Some of these attributes are used directly by TrackBuilder, such as the track curvature radius, others are used only by the Simulator, such as wind speed and direction.

## **2.2 Usage**

For installation information and the most up to date usage documentation, consult the wiki on the project's website:

`http://code.google.com/p/prt-sim/wiki/TrackBuilder`

## **2.3 Features**

### **2.3.1 Online Map & Elevation Data**

TrackBuilder's main window displays map, aerial, or terrain imagery provided by Google (Figure 2.1). Navigation may be performed using click-and-drag scrolling as is familiar to users of contemporary map websites. Using an online map data provider supplies the user with up-to-date, high-resolution road and terrain data for most countries of the world, and eliminates the often difficult or tedious task of finding georeferenced imagery data.

Elevation data is also provided by Google. As track is laid down, the elevation is sampled at points along its length. The elevation data's primary purpose is to improve the accuracy of the Simulator's energy-usage model.

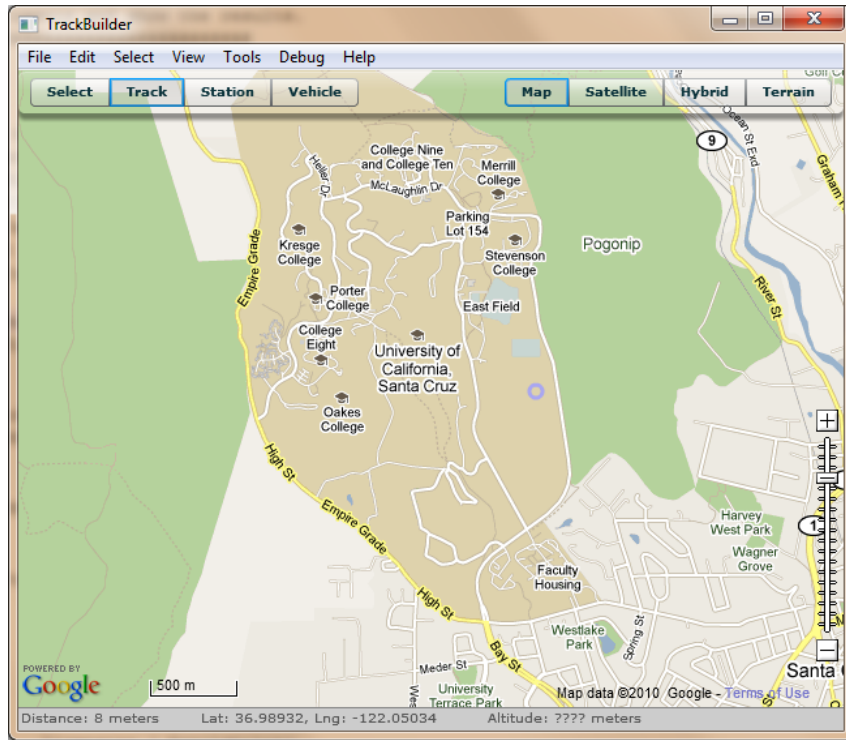


Figure 2.1: TrackBuilder’s main window.

### 2.3.2 Curved Track Segments

TrackBuilder models each segment of track as being either straight or a constant-radius curve. Although curved segments can be abstracted away for low speed PRT systems or low detail scenarios, explicitly modeling curved track segments is important for accurately modeling higher speed systems. Large radius curves suitable for high-speed travel are often difficult to integrate with an urban area’s existing infrastructure. Small radius curves require vehicles to reduce speed or to experience high acceleration loads through the curve<sup>1</sup>. TrackBuilder helps to explore this trade-off by providing ac-

<sup>1</sup>Systems which use banked track, or articulated pods would experience the acceleration as z-axis loading. Others would experience it as lateral acceleration.

celeration estimates for curves, allowing for curve speed limits to be set separately from straight segments, and by providing high detail aerial map data that can show potential obstructions.

Judging what angle a curved segment should be so that a subsequent straight segment will reach a particular point would be a frustrating experience. This realization lead to the development of the automated track placement feature, described in section 2.3.4.

### **2.3.3 One-Way & Two-Way Track**

One-way guideways are generally well suited for PRT; interchanges are simpler and smaller, and a one-way open-loop can provide coverage to greater area than an equivalent pinched-loop configuration would. The use of two-way track can reduce travel times however, and when right of way is expensive in relation to the guideway cost (or simply difficult to acquire) then two-way track is preferable.

TrackBuilder fully supports laying out one-way guideways. Two-way guideways are partially supported at this time<sup>2</sup>.

### **2.3.4 Automated Track Placement**

One approach to track placement would be to have separate “straight track” and “curved track” tools which the user may switch between as they place tracks one segment at a time. The approach taken in TrackBuilder, however, is for the user to

---

<sup>2</sup>Implementation of two-way guideways is complete, but there are significant bugs that affect usability in the current release version (TrackBuilder v0.6.12)

specify an origin point and a destination point and allow the program to handle the task of laying down an appropriate mixture of straight and curved track segments to connect the two points. When the origin point lies on an existing track segment, whether at an endpoint or in the middle, whether one-way or two-way, an appropriate intersection is formed. Likewise when the destination point lies on a track segment. When a destination point lies within the radius of curvature and cannot be reached directly, the program routes the track away from the destination point then loops back toward it. Stations are composed of multiple track segments, and are also automatically assembled and placed with a single click.

Associated with the automated track placement is a “live” preview of where the track will be placed when the user clicks. As the user moves the mouse, the preview is automatically updated. This preview system is also applied to station and vehicle placements.

### **2.3.5 Undo**

TrackBuilder features a robust, infinite undo system for track, station, and vehicle placement.

### **2.3.6 Google Transit Feed Import**

Google Transit Feed (GTF) is a format specification for submitting conventional mass transit routes and schedules to Google for integration with their Google Maps service. The GTF files contains stop locations, route shapes, and scheduling in-

formation. TrackBuilder features the ability to import GTF files and create “tracks” that represent the routes used, with stations at each stop location. From the scheduling information it infers how many buses, trains, etc. would be required to fulfill the schedule. TrackBuilder appends extra schedule information to its save file, which allows a special controller (the “GTF controller”) to perform the shared vehicle, fixed route, fixed schedule operation of conventional mass transit.

## **2.4 Implementation**

### **2.4.1 Language and Framework**

The ease-of-use that online map data can provide the end user was a deciding factor in choosing the language and framework in which to implement TrackBuilder. Google, and many other map providers, offer map APIs that are accessible from Javascript libraries which are meant to be used from within an Internet browser. At the time of TrackBuilder’s initial design, Google began offering a map API for applications written in ActionScript3 (AS3) as well—the language used for Flash. Also around this time, Adobe released their Adobe Integrated Runtime (AIR) platform which allows applications written in AS3 to be locally installed, run independent of any browser, and to have greater access to the file system. The combination of Adobe AIR with an AS3 version of Google’s map library makes for a compelling combination, which resulted in the choice to use them for implementing TrackBuilder.

The Adobe AIR runtime is proprietary, but free. The principle development



tool for Flash or AIR apps is proprietary and non-free, but there is an free, open-source SDK available as well. The use of Google’s map library and data is free below certain usage limits, which are sufficiently high that they are not likely to ever be exceeded.

### **2.4.2 Laying Track**

To lay track for a scenario, the user enters “Track” mode via the menu or the on-screen button. When in track mode the origin marker (displayed as a small blue circle) initially follows the current mouse cursor’s position. After clicking on the map, the origin marker is locked into place and TrackBuilder switches to showing a preview of track that runs from the origin marker to the mouse cursor. With subsequent clicks, the preview track is converted to permanent track and the origin marker is moved to the track’s new endpoint. If the user connects the new track to an existing track segment, the origin marker is freed and returns to following the mouse cursor. The user may free the origin marker at any time by hitting the “Esc” key.

### **Cutting Corners**

When the user is laying down track which “turns a corner”, TrackBuilder will trim the previously placed segment so that the corner is cut rather than extending the track from the previous segment’s end (Figure 2.2). In the common case in which track is being laid down so as to follow a grid, such as streets in a downtown area, TrackBuilder’s behavior allows the user to click once in the center of the intersection then click again at next desired turning point. The corner will be cut, and the pre and

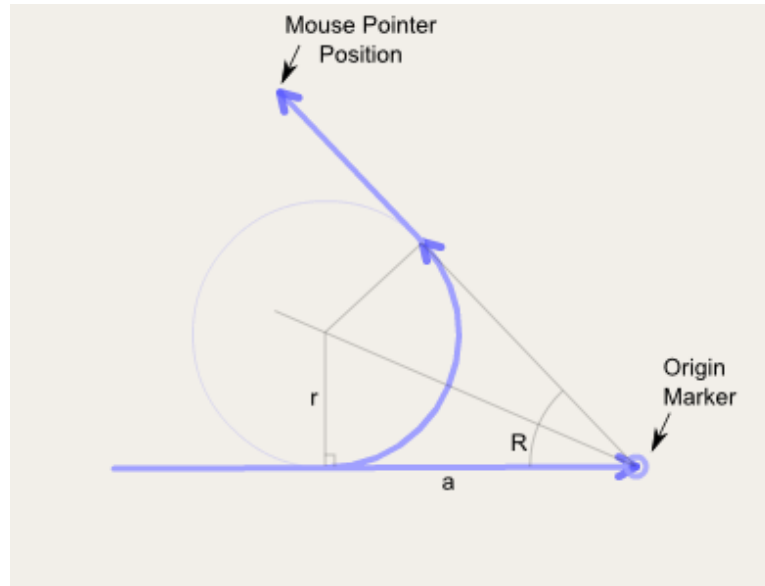


Figure 2.2: An annotated screen capture from TrackBuilder that demonstrates a corner being cut. The thick blue lines are what the user sees while creating a curve, and the light gray lines are annotations. Here, the user has already created a track segment running from West to East and is viewing the preview for creating a curve + straight segment heading Northwest. Once the user clicks, the line segment extending past the curve (line segment **a**) will be trimmed away, and the origin marker will be moved to the end of the NW facing track segment. The curve's center point is calculated from the known angle  $R$  and the radius  $r$  using basic trigonometry.

post-corner tracks will run parallel to the streets.

## S-Curves

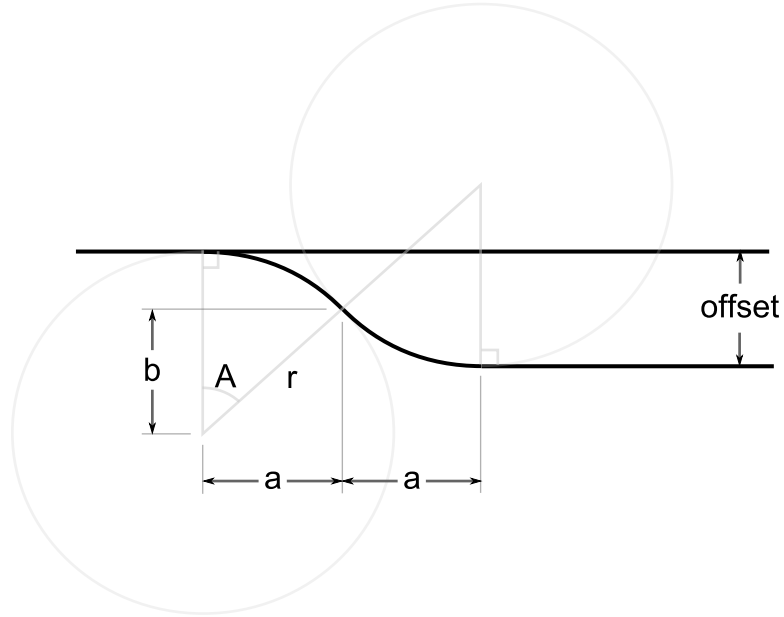


Figure 2.3: An S-Curve created using two constant radius curve segments.

S-Curves are used when a merge or split leads into parallel track segment (Figure 2.3). S-Curves typically use large radius curves, allowing it to be negotiated by a vehicle traveling at full speed.

S-Curves are parametrized by the offset distance and curve radius, and are created from a single starting point. Once distances  $a$ ,  $b$ , and the angle  $A$  are found the locations of the inflection point, the final point, and the centers of the circles can be found in a straightforward manner.

$$b = r - \frac{offset}{2} \quad (2.1)$$

$$A = \arccos\left(\frac{b}{r}\right) \quad (2.2)$$

$$a = \sin(A) r \quad (2.3)$$

S-Curves of this design require that  $\frac{offset}{2}$  distance is less than or equal to the curve radius. Inserting a straight segment between the two curved segments could alleviate this restriction.

## Interchanges

Support for laying down two-way track, modeled as being two anti-parallel tracks on the same right of way, brings more complexity than might naively be supposed. To have non-stop intersections with one-way track requires only that one track curve to meet the other, with perhaps a stretch of parallel track to allow for acceleration or deceleration prior to merging. Intersections between two-way tracks may require interchanges that are akin to a freeway interchange, complete with under/over passes.

### 2.4.3 Undo

The undo system is implemented using a stack, where each element on the stack represents one undo-able user command. When the user initiates a new command, a new array is pushed onto the stack. Each element of the array contains a “micro-step”, which is a structure containing a function reference and function arguments. Any code

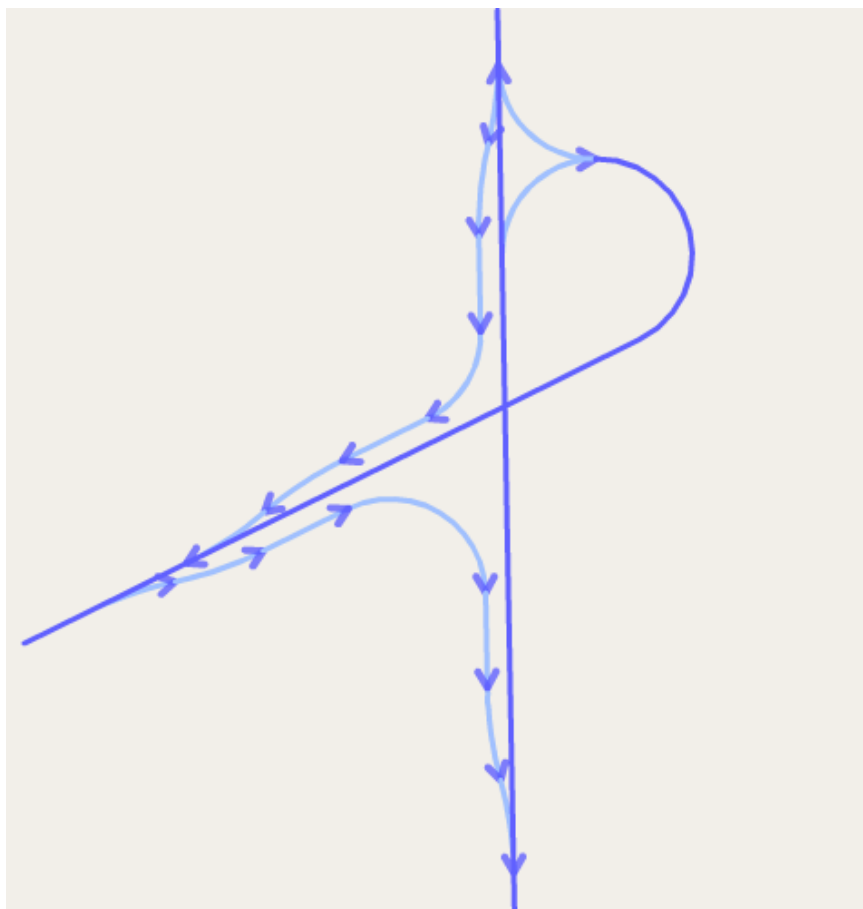


Figure 2.4: An intersection in which a SW-NE running track joins with a N-S running track. The dark blue lines are two-way, and the light blue lines with chevrons are one-way track. The entire interchange is generated procedurally from the point of intersection, two vectors indicating the direction of the intersecting tracks, and user-definable values. Though not rendered as such, the SW-NE line rises above and passes over the N-S line at the apparent point of intersection. In this example, the two-way track is configured to have right hand traffic, matching the driving rules in the USA.

which changes TrackBuilder’s state during a user command is preceded by adding a micro-step to the top element (i.e. array) of the stack.

Each micro-step is written so as to undo the associated change in state. Most micro-steps consist of reverting the value of a single variable or reverting the value of a single element in an array. For these common cases, the Undo module provides some convenience functions.

When the user requests an undo, the top array is popped from the stack and the associated micro-step functions are executed using their stored arguments. The micro-step functions are executed in a reverse order from how they were added.

#### **2.4.4 Track Preview**

The track placement preview is implemented using the undo system. A separate undo stack is used to store the micro-steps that result from creating a preview for a particular positioning of the mouse cursor. When the mouse is moved, the old preview is undone and a new preview created. Unlike the stack used for undoing user actions, the undo stack used for previews does not retain more than one level of history and thus never grows beyond one element.

#### **2.4.5 Creating Stations**

TrackBuilder stations consist of an offramp S-curve which connects to a straight deceleration segment, which is followed by one or more platforms. After the platforms there is a straight acceleration segment, and an onramp S-curve that reconnects the

station line with the main line.

Platforms contain one or more berths. Berths may have any combination of the following capabilities: (i) embark a passenger (ii) disembark a passenger (iii) move a vehicle into storage (iv) move a vehicle out of storage. All berths within a given platform have the same capabilities. Berths are in-line with the platform, meaning that a vehicle must pass through all berths to traverse the platform.

Stations are procedurally created from a saved set of parameters called a “StationModel”<sup>3</sup>. A StationModel specifies a station’s physical dimensions (e.g. the deceleration length), the number of platforms, the number of berths for each platform and their capabilities, and the number of vehicles that can be stored at the station. The user may create new StationModels and update some parameters of existing models from within TrackBuilder. Each station instance will have single StationModel that it is based on and to which it refers to for some of its attributes.

TrackBuilder is limited to linear stations. The stations are linear in the typical PRT usage of the word, meaning that vehicle berths are in-line with the station’s track. Stations are also linear in a much stricter sense—stations don’t curve. Stations may only be placed such that they run parallel to an equal or longer length of straight, main-line track.

When the station tool is selected and the mouse cursor is over a track segment, TrackBuilder attempts to create a station preview using parameters from the currently

---

<sup>3</sup>StationModels are not found in the current release (v0.6.12) but have been implemented in the latest SVN revision. At the time of this writing, the Simulator and controllers have not yet been updated to be compatible with the changes.

selected StationModel. If the point where the station line would rejoin with the main line is beyond the end of main line’s straight segment, then no station can be created and no preview is shown. An exception to this is if the main line’s end is exposed<sup>4</sup>, in which case the the main line is extended to the merge point and a preview station is shown. If the user clicks, the preview is replaced with a permanent station. If the mouse cursor is moved, then the current preview is undone.

### **2.4.6 Creating Vehicles**

Vehicles are akin to Stations in that they also use a “model” to describe a common set of attributes, this time given the rather fanciful name of “VehicleModel.” A VehicleModel specifies such traits as vehicle length, passenger capacity, passenger-comfort based limits on velocity, acceleration, and jerk, as well as (generally looser) hardware-based limits on the same. The model also specifies a number of parameters that affect energy usage: mass, frontal area, coefficient of drag, coefficient of rolling resistance, powertrain efficiency, and regenerative breaking efficiency.

Vehicles may be placed almost anywhere on the track network, the exception being that vehicles may not be placed such that they overlap.

### **2.4.7 Creating Passengers**

Note that TrackBuilder does not create passengers. See the 3.3.6 in the Simulator chapter for more information about passengers and the passenger file.

---

<sup>4</sup>An end is exposed if it is not connected to any other TrackSegments.



## 2.4.8 Google Transit Feed Import

The Google Transit Feed Specification<sup>5</sup> (GTFS) consists of a set of comma separated value (CSV) files. Only a subset of the files and fields are used by TrackBuilder (see Table 2.1).

Filename	Fields used by TrackBuilder
stops.txt	stop_id, stop_name, stop_lat, stop_lon
routes.txt	route_id, route_short_name, route_long_name, route_desc <sup>a</sup> , route_type
trips.txt	route_id, service_id, trip_id, shape_id <sup>b</sup>
stop_times.txt	trip_id, arrival_time, departure_time, stop_id, stop_sequence
calendar.txt	service_id, start_date, end_date, days of the week
calendar_dates.txt <sup>a</sup>	service_id, date, exception_type
shapes.txt <sup>b</sup>	shape_id, shape_pt_lat, shape_pt_lon, shape_pt_sequence

<sup>a</sup> Optional.

<sup>b</sup> Optional in GTFS specification, but required by TrackBuilder.

Table 2.1: GTFS files and fields used.

The task of importing a GTFS feed initially appears to be very straightforward: parse the files, construct a two-way track for each shape in shapes.txt, add a station for each stop in stops.txt, and create vehicles at approximately correct initial positions. Finally, append scheduling data for each vehicle to the save file for use by the GTF\_controller.

In the course of developing the importer, and in testing it with various publicly available feeds, a number of unexpected issues arose and needed to be addressed.

---

<sup>5</sup>[http://code.google.com/transit/spec/transit\\_feed\\_specification.html](http://code.google.com/transit/spec/transit_feed_specification.html)

## Overlapping routes

Service routes often have sections that overlap—the same section of road or rail is used by more than one route. The waypoints describing these overlapping sections are generally not reused however, meaning that aside from the section’s proximity there is no indication that they are actually representing the same piece of road or rail.

If the GTFS importer were to naively create track segments for each route, the overlapping—but not connected—sections would present a confusing result to the user. Additionally, the disconnected nature of the routes would prevent the generated track from being used for anything but the scheduled vehicle trips—specifically, it would exclude reusing the track network for simulation under a PRT or GRT service model.

To efficiently merge the overlapping sections of routes, TrackBuilder uses the assumption that waypoints along the overlapping section will be in close proximity. When two or more waypoints are found to be near each other, the waypoints—nodes within the graph representation—are merged. To find which waypoints are in close proximity, a NearTree data structure is used [9].

The assumption that overlapping routes will have waypoints close to each other will fail in some cases. Particularly when waypoints along the route are sparse, the point at which one route makes a turn and diverges from the other may not have a corresponding waypoint. A more robust test would be to check for colinearity of the graph’s edge segments, rather than just the proximity of the nodes.

## Large Number of Waypoints

The polylines in some transit agencies' shapes.txt files have a very large number of waypoints. In these cases it appears that the polylines were created directly from GPS traces and so have one waypoint for every second of travel, in contrast to the normal case where waypoints are only used when the route makes a significant change in direction.

The design for the importer has TrackBuilder creating track segments to connect adjacent waypoints. With the feeds that have a high density of waypoints, this leads to a large number of track segments being created—more than can be rendered gracefully. In addition to the performance impact on TrackBuilder, the large number of small track segments negatively impacts performance of the downstream Simulator and Controller.

This issue is resolved by using a procedure to simplify the graph, given here in pseudocode:

```
foreach nodeB in graph:
  if nodeB.adjNodes.length == 2: // not an intersection or dead-end
    nodeA = nodeB.adjNodes[0]
    nodeC = nodeB.adjNodes[1]

    // Use triangle inequality to estimate the node's colinearity
    AB = nodeA.distFrom(nodeB)
    BC = nodeB.distFrom(nodeC)
    AC = nodeA.distFrom(nodeC)

    if AC/(AB + BC) > 0.99905: // angle ABC > 5 degrees
      graph.removeNode(nodeB)
```

## Creating Stations From GTF Stops

A TrackBuilder station is created for each stop in the GTF Data. Stations created by the GTF importer are simplified versions of the user-created stations in that they lack offramp and onramp S-curves and have a maximum of one platform.

Since TrackBuilder stations have a platform with some length, whereas GTF stops are a single point, the importer must infer some reasonable orientation for the stations. Additionally, each stop's latitude and longitude is specified within the GTF files but there is no data that directly ties a stop to the route shapes found in the shapes.txt file. The stops do not, in general, lie on the route shapes. In order to orient the stations and to create track segments that connect the station with the main track, the NearTree data structure is used to find the closest point to the stop location. The points that are adjacent to the closest point are examined, and from them the point which is closest to the station is also chosen. The station is created so as to run parallel to the two chosen points, and the station's endpoints are connected to them.

The above method of connecting stations can be visually unappealing. The track segments connecting a station to the main line may be of uneven lengths and will often form unnatural-looking angles. Additionally, a station's platform uses one-way guideway, and the platform may be oriented in the wrong direction for which side of the street it is on. These issues are purely cosmetic however, and have not yet reached sufficient priority to be addressed.

## Creating Vehicles From GTF Data

Vehicles are not explicitly described in the GTFS files. In order for Track-Builder to create vehicles it must infer their existence from the trips information. A vehicle may fulfill trips on several routes over the course of the day and multiple vehicles may be assigned to a single route. Creating a minimum number of vehicles to fulfill all scheduled trips has a simple solution:

```
allTrips.sortBy(StartTime)
vehicles = {}

foreach trip in allTrips:
  foreach vehicle in vehicles:
    if vehicle.trips.length == 0 or \
       (vehicle.trips[-1].endTime <= trip.startTime and \
        vehicle.trips[-1].endPosition.distFrom(trip.startPosition) \
        <= DISTANCE_THRESHOLD):
      vehicle.trips.append(trip)
    else:
      v = new Vehicle
      v.trips.append(trip)
      vehicles.add( v )
```

In the above pseudocode, `DISTANCE_THRESHOLD` is chosen such that a vehicle could travel the distance within a short period of time. In practice, a vehicle is rarely assigned to a new route unless it is at a metro center (i.e. a hub station), thus `DISTANCE_THRESHOLD` need only be large enough to reach the other stops within the metro center. The worst case behavior is that more vehicles are used in the simulation that are actually used to service the imported routes.

If a vehicle's first trip begins sometime after the simulation's start time, then the vehicle is placed at the trip's starting position. For a trip that is already in progress

when the simulation period begins, the vehicle is placed so as to be part-way along its trip. More formally, let  $d_0, a_1, d_1 \dots a_i, d_i, a_{i+1}, d_{i+1} \dots a_n$  be the sequence of arrival and departure times for a trip from station 0 to station  $n$ , and let  $t_{start}$  be the simulation start time. If  $t_{start}$  falls between an arrival and departure, then the vehicle is created in the corresponding station. If  $t_{start}$  falls between a departure and an arrival, say  $d_i$  and  $a_{i+1}$ , then let  $l$  be the length of the path between stations  $i$  and  $i+1$  on the trip. Then the vehicle is placed

$$\frac{t_{start} - d_i}{a_{i+1} - d_i} \times l$$

meters along the path.

Since routes may have been significantly modified prior to creating track segments (overlapping routes merged, and waypoints removed) there is no longer a simple correlation between the original route data and TrackBuilder's track segments. To find paths between stations, Dijkstra's algorithm is used with the assumption that the shortest path between two stations on a trip will be a reasonable approximation of the actual route.

#### 2.4.9 Saving and Loading Files

Save files are formatted in XML, with the XML entities closely resembling the data content of the in-memory Objects used by TrackBuilder. Each Object represented in the save file has member functions that implement the serialization/deserialization to and from XML. The XML schema used for TrackBuilder's save files is described in Appendix A.

## Chapter 3

# Simulator

### 3.1 Introduction

There are many types of simulators, at many different levels of abstraction and coverage. This Sim is a so-called “microsimulator” in which each vehicle’s motion is explicitly modeled. An entire PRT network is modeled, including vehicle movement within stations. The simulator is at a high level of abstraction; vehicle motion is simulated at the level of 1D reference trajectories. The vehicle control logic is separate from the simulation and runs in a separate process, with the simulator providing a message-based API for interacting with one or more controllers. The simulator is intended primarily as a tool to aid the development of high-level PRT control algorithms and to help analyze energy usage, network efficiency, performance statistics, capacity constraints, and collision avoidance.

## 3.2 Level of Detail

The Sim is not a detailed simulation of vehicle or guideway physics. In designing a Personal Rapid Transit system, there are many different choices that a system implementer must make regarding a vehicle’s sensors, propulsion and suspension as well as properties of the guideway. In order to keep the simulator generic, many of those details have been abstracted away.

Vehicles are modeled as though they were captive to a linear guideway with no ability for lateral movement within the guideway. That is, vehicles are treated as though they are on rails, rather than running on a surface.

The limitations of a vehicle’s propulsion and breaking system are modeled as constant-value limits on the maximum and minimum velocity, acceleration, and jerk (i.e. derivative of acceleration) in the direction of travel. Limits on vertical or lateral acceleration and jerk are not modeled, though the scenario layout tool TrackBuilder does provide estimates of lateral acceleration for unbanked curves (at the curve’s maximum speed) during a track network’s design phase.

Low level vehicle control is entirely abstracted away. The Controller controls vehicles by providing reference trajectories, in the form of 1D cubic splines, and the vehicle is assumed to be capable of adhering to the reference perfectly.

Vehicles are assumed to be capable of sensing or otherwise tracking their position, velocity and acceleration, and to be capable of determining the range of a vehicle in front of them (up to 500 meters away, by default) without regard to line-of-sight.



Modeling communications latency, power limits, and battery life are all appropriate for the desired level of detail, but are not implemented at this time.

## 3.3 Implementation

### 3.3.1 Separation of Simulation and Controller

Separating the vehicle control logic from the simulation logic is an obviously good design choice. This project enforces a deeper separation than is found in other publicly available simulators in that the control is separated from the simulation at the process level. The inter-process communication (IPC) is implemented as a message passing system utilizing TCP/IP, allowing third-parties to implement their own controllers without modifying the sim and without imposing any fundamental constraints on their choice of programming languages<sup>1</sup>. The simulator provides an API which all controllers use, including the controllers that are included with the project.

### 3.3.2 Hybrid Simulator

The simulator is a hybrid discrete-continuous event simulator. Vehicle movement is modeled with continuous functions (cubic splines), but other aspects of the simulation are treated as discrete events. Examples of events are:

- A vehicle's nose reaches the end of a track segment.

---

<sup>1</sup>In principle, the use of TCP/IP allows the controller to run from a separate computer, removing machine architecture limitations as well. In practice, all testing has been done with a loopback connection.

- A vehicle's tail clears a track segment.
- A vehicle's nose reaches a specified position.
- A specified time is reached.
- The distance between two vehicles falls below a specified limit.
- A collision occurs.
- A passenger is created at a station.

Most events are accompanied by the Sim sending a message that describes the event to the vehicle controller(s). The Sim can behave as though it were a fixed-timestep simulator by inserting additional events at a regular interval.

### **3.3.3 Simulation Time**

It is useful if the Simulator and controller(s) can together run faster than real time, and also useful if they can tolerate one or both pieces taking more time to compute than would be reasonable in a live system. Though a production system will have hard real-time requirements, the sim is designed with an eye towards algorithm development and with the expectation that controllers may be running unoptimized code, or be running in debug mode. To allow for a distortion of time, either faster and slower than realtime, the Sim only loosely ties simulation time to wall-clock time. When an event occurs the simulation pauses and event message(s) are sent out. The controller(s) may

then send commands or queries to the Sim, which remains paused until all controllers send a “resume” message.

When the simulation is being run via the GUI, the results of the simulation are displayed for the user as the Sim progresses. The user may control what multiple of real-time they wish the Sim to run at. To slow the pace of the simulation the Simulator’s sim thread will, if necessary, sleep for periods between events so that simulation time and wall-clock time will pass at approximately the correct ratio.

### **3.3.4 Vehicles**

#### **Coordinate Frames**

Each track segment (a straight or curved piece of track) has its own 1D coordinate frame. While vehicle positions are stored internally as just an offset from the vehicle’s initial position, vehicle positions are communicated to controllers in the form of a unique track segment id and the distance from the beginning of the segment.

#### **Cubic Splines**

A cubic spline is a piecewise function defined by third order polynomials. Splines provide a compact representation of a vehicle’s position as a function of time. The splines used by the simulation are required to be twice continuously differentiable ( $C^2$ ), i.e. neither velocity nor acceleration can change instantaneously. The simulator allows for constraints to be placed on a vehicle’s maximum and minimum velocity, acceleration and jerk values, and the use of cubic splines makes it simple to verify that

the constraints are being satisfied.

### **Collision detection**

Checking for collisions between vehicles is a matter of checking for intersections between their splines, with an offset to account for vehicle lengths and differences in their coordinate frames. Each spline is composed of multiple polynomials, and each polynomial is only valid within a limited time domain. Detecting a polynomial intersection can be accomplished by subtracting one polynomial from the other, and finding the resulting polynomial's roots. If a root falls within the domain in which both polynomials are valid, a collision has been detected.

The simulator only checks for upcoming collisions that might occur on a vehicle's current track segment. When a vehicle (vehicle A) reaches a new track segment, the simulator checks for upcoming collisions between vehicle A and the next vehicle (vehicle B) on the track segment. If an upcoming collision is detected, the collision is added to the following vehicle's event queue. When a vehicle's spline is changed, the queued collisions are cleared for the changed vehicle and for its following vehicle, and the collision detection routine is rerun for both vehicles.

The collision detection routine described above relies on the nose of the following vehicle intersecting with the tail of the leading vehicle. The case where one vehicle "sideswipes" another at a merge point may not be caught so long as the two vehicles maintain the same velocity. To detect sideswipe collisions an additional check is used. Whenever a vehicle reaches the track segment downstream of a merge and a leading

vehicle is found on the downstream segment, the simulator checks if the leading vehicle is straddling the merge point—if it is, a collision is reported as happening immediately.

### **3.3.5 Stations**

Within the Simulator, Stations are little more than a list of waiting passengers and a container for platforms and their berths.

Early versions of the Simulator treated Stations as though they were a world apart. Once a vehicle entered a station, it no longer moved along TrackSegments in the normal fashion, but would instead move in discrete jumps between berths. Current versions, however, treat Stations as an integrated part of the track network and vehicles within stations move and are controlled as they would be elsewhere.

During a simulation, stations are displayed using a circle icon. The user can hover over a station's icon to see how many passengers are waiting, and the average and maximum waiting times. The color of the station icon also indicates the number of passengers waiting (Figure 3.1).

### **Platforms & Berths**

Platforms do little more than associate Berths with a TrackSegment at this time. In future versions, Platforms will play a role in modeling passengers' movement through a Station.

Berths implement the tasks of moving passengers into and out of vehicles, and of moving vehicles into and out of storage. When a controller requests that a passenger

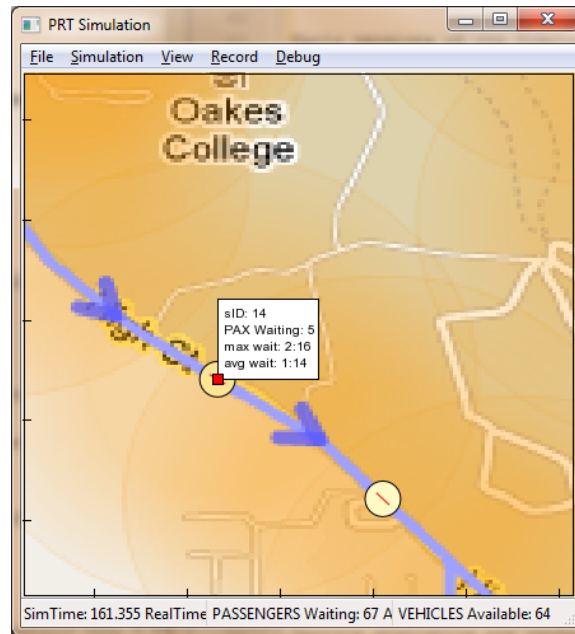


Figure 3.1: A Station icon, and a tooltip showing the passenger information.

embark/disembark from a vehicle, the Berth checks that the vehicle is stopped, is fully inside the Berth, and that the passenger is available at the station or is within the vehicle. If the controller's request is valid, it is informed that the embark/disembark has begun. Once the time specified for passenger loading/unloading has passed, the passenger is moved to the appropriate destination, and the controller is informed that the embark/disembark has completed. The story is much the same with moving vehicles into and out of storage.

### Vehicle Storage

Stations may be designed with vehicle storage capabilities. Storage areas do not have a visual representation and are not constrained by geometry. A station can

have a capacity limit on its storage area, or have unlimited storage. Similarly, it can have a finite initial supply of vehicles in storage, or have an unlimited supply.

A station’s storage area is implemented as a special, infinitely long track segment that is not rendered in the sim’s visualization. When a vehicle is moved into storage, it is placed on this segment. When a vehicle is pulled from storage, a vehicle is removed from the track segment (FIFO ordering), if available. If no vehicle is on the storage segment, but the storage area’s capacity is not yet zero, a new vehicle is created.

### **3.3.6 Passengers**

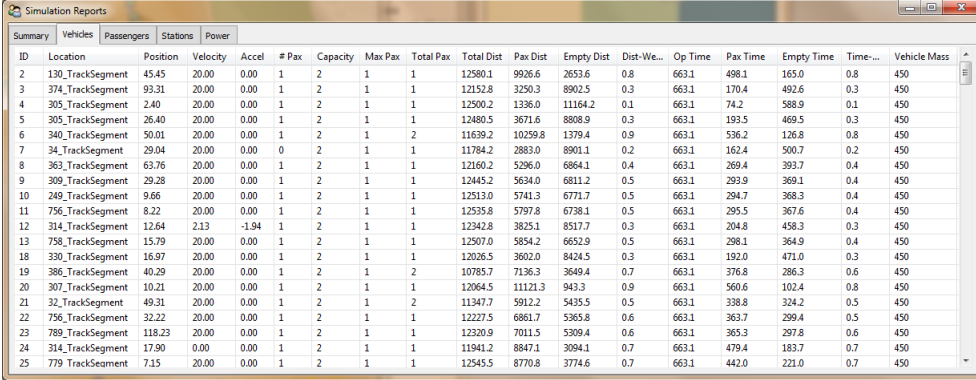
The Sim spawns passengers based on data from a passenger file. The passenger file specifies origin and destination stations, spawn time, loading and unloading times, mass (including luggage), and an optional flag indicating whether the passenger will share a vehicle with another passenger bound for the same destination. In the current version of the simulator, passengers are not modeled prior to their arrival at their origin station and passengers’ positions within a station are not modeled explicitly.

The simulator does not assign active behaviors to passengers—they are treated as passive objects to be manipulated by the controller. The choice of which passenger(s) to embark/disembark from a vehicle is left up to the controller(s), with the limitation that the vehicle and passenger(s) must be at the same station and that the vehicle must be stopped within an appropriate berth. The controller(s) can also direct passengers to walk to other stations. When a passenger arrives at its destination station, it is removed from the simulation and can no longer be controlled.

### 3.3.7 Statistics

#### Vehicles

Per-vehicle statistics include: total distance traveled, total distance with passengers on board, empty distance, average number of passengers weighted by distance, time spent in operation, time spent with passengers on board, time spent empty, average number of passengers weighted by time (Figure 3.2).



ID	Location	Position	Velocity	Accel	# Pax	Capacity	Max Pax	Total Pax	Total Dist	Pax Dist	Empty Dist	Dist-We...	Op Time	Pax Time	Empty Time	Time...	Vehicle Mass
2	130_TrackSegment	45.45	20.00	0.00	1	2	1	1	12580.1	9926.6	2653.6	0.8	663.1	498.1	165.0	0.8	450
3	374_TrackSegment	93.31	20.00	0.00	1	2	1	1	12152.8	3250.3	8902.5	0.3	663.1	170.4	492.6	0.3	450
4	305_TrackSegment	2.40	20.00	0.00	1	2	1	1	12500.2	1336.0	11164.2	0.1	663.1	74.2	588.9	0.1	450
5	305_TrackSegment	26.40	20.00	0.00	1	2	1	1	12480.5	3671.6	8808.9	0.3	663.1	193.5	469.5	0.3	450
6	340_TrackSegment	50.01	20.00	0.00	1	2	1	2	11639.2	10259.8	1379.4	0.9	663.1	536.2	126.8	0.8	450
7	34_TrackSegment	29.04	20.00	0.00	0	2	1	1	11784.2	2883.0	8901.1	0.2	663.1	162.4	500.7	0.2	450
8	363_TrackSegment	63.76	20.00	0.00	1	2	1	1	12160.2	5296.0	6864.1	0.4	663.1	269.4	393.7	0.4	450
9	309_TrackSegment	29.28	20.00	0.00	1	2	1	1	12445.2	5634.0	6811.2	0.5	663.1	293.9	369.1	0.4	450
10	249_TrackSegment	9.66	20.00	0.00	1	2	1	1	12513.0	5741.3	6771.7	0.5	663.1	294.7	368.3	0.4	450
11	756_TrackSegment	8.22	20.00	0.00	1	2	1	1	12535.8	5797.8	6738.1	0.5	663.1	295.5	367.6	0.4	450
12	314_TrackSegment	12.64	2.13	-1.94	1	2	1	1	12342.8	3825.1	8517.7	0.3	663.1	204.8	458.3	0.3	450
13	758_TrackSegment	15.79	20.00	0.00	1	2	1	1	12507.0	5854.2	6652.9	0.5	663.1	298.1	364.9	0.4	450
18	330_TrackSegment	16.97	20.00	0.00	1	2	1	1	12026.5	3602.0	8424.5	0.3	663.1	192.0	471.0	0.3	450
19	386_TrackSegment	40.29	20.00	0.00	1	2	1	2	10785.7	7136.3	3649.4	0.7	663.1	376.8	286.3	0.6	450
20	307_TrackSegment	10.21	20.00	0.00	1	2	1	1	12064.5	11121.3	943.3	0.9	663.1	560.6	102.4	0.8	450
21	32_TrackSegment	49.31	20.00	0.00	1	2	1	2	11347.7	5912.2	5435.5	0.5	663.1	338.8	324.2	0.5	450
22	756_TrackSegment	32.22	20.00	0.00	1	2	1	1	12227.5	6861.7	5365.8	0.6	663.1	363.7	299.4	0.5	450
23	789_TrackSegment	118.23	20.00	0.00	1	2	1	1	12320.9	7011.5	5309.4	0.6	663.1	365.3	297.8	0.6	450
24	314_TrackSegment	17.90	0.00	0.00	1	2	1	1	11941.2	8847.1	3094.1	0.7	663.1	479.4	183.7	0.7	450
25	779_TrackSegment	7.15	20.00	0.00	1	2	1	1	12545.5	8770.8	3774.6	0.7	663.1	442.0	221.0	0.7	450

Figure 3.2: Vehicle Report

#### Passengers

Per-passenger statistics include: times spent waiting, walking, and riding, their total time, whether or not the trip was a success, and current location at time of the report generation (Figure 3.3).



ID	Origin	Destination	Wait Time	Ride Time	Walk Time	Total Time	Trip Success	Current Locat...	Load Delay	Unload Delay	Mass	Will Share
0	UCSC East Fie...	6	0:49	10:20	0:00	11:09	True	6	0:15	0:10	85	True
1	15	21	2:51	4:21	0:00	7:13	True	21	0:15	0:10	85	True
2	26	9	0:48	5:50	0:00	6:39	True	9	0:15	0:10	85	True
3	Delaware Stor...	9	0:38	7:45	0:00	8:23	True	9	0:15	0:10	85	True
4	5	14	1:27	3:06	0:00	4:34	True	14	0:15	0:10	85	True
5	0	14	3:48	6:16	0:00	10:05	True	14	0:15	0:10	85	True
6	0	5	3:48	3:38	0:00	7:27	True	5	0:15	0:10	85	True
7	14	3	4:29	3:55	0:00	8:24	True	3	0:15	0:10	85	True
8	15	20	2:48	2:59	0:00	5:48	True	20	0:15	0:10	85	True
9	15	0	2:50	10:05	0:00	12:56	True	0	0:15	0:10	85	True
10	2	Delaware Stor...	1:18	1:21	0:00	2:40	True	Delaware Stor...	0:15	0:10	85	True
11	25	18	0:59	7:41	0:00	8:40	True	18	0:15	0:10	85	True
12	21	1	1:07	13:35	0:00	14:43	True	1	0:15	0:10	85	True
13	21	24	1:05	2:25	0:00	3:31	True	24	0:15	0:10	85	True
14	10	26	0:17	9:27	0:00	9:45	True	26	0:15	0:10	85	True
15	12	21	0:42	8:00	0:00	8:42	True	21	0:15	0:10	85	True
16	26	15	0:32	7:41	0:00	8:13	True	15	0:15	0:10	85	True
17	13	20	10:47	1:09	0:00	11:57	True	20	0:15	0:10	85	True
18	5	25	2:18	8:40	0:00	10:58	True	25	0:15	0:10	85	True
19	14	20	4:08	1:34	0:00	5:43	True	20	0:15	0:10	85	True

Figure 3.3: Passenger Report

## Station

Per-station statistics include: number of passenger arrivals, number of departures, and the minimum, mean, and maximum passenger wait times for the entirety of the simulation and at the time of the report generation (Figure 3.4).

ID	Current # Pax	# Departures	# Arrivals	Current Pax Min Wait	Current Pax Mean Wait	Current Pax Max Wait	All Pax Min Wait	All Pax Mean Wait	All Pax Max Wait
0	12	10	0	0:00	1:30	6:36	0:00	1:45	7:32
1	9	6	4	0:00	1:12	4:53	0:00	1:14	4:53
2	12	3	3	0:00	1:50	7:32	0:00	1:57	7:37
3	4	6	3	0:00	1:27	4:16	0:00	1:13	4:16
4	7	5	1	0:00	1:14	Mean wait time for passengers currently in station	1:14	6:09	
5	5	11	2	0:00	0:43	2:26	0:00	0:55	4:10
6	5	6	0	0:00	1:51	5:38	0:00	1:30	5:38
7	0	19	4	0:00	0:00	0:00	0:00	0:08	0:49
8	4	6	0	0:00	1:03	3:13	0:00	1:19	5:38
9	4	12	4	0:00	0:45	3:24	0:00	0:50	3:24
10	9	9	1	0:00	1:06	3:50	0:00	1:40	5:52
11	6	8	2	0:00	1:13	3:30	0:00	1:31	4:08
12	12	3	2	0:00	1:47	8:31	0:00	1:56	8:31
13	8	0	1	0:00	2:35	10:27	0:00	2:35	10:27
14	7	9	4	0:00	1:55	5:57	0:00	2:12	7:01
15	6	15	5	0:00	0:56	2:40	0:00	1:16	3:33
16	4	10	4	0:00	1:06	3:45	0:00	1:32	5:07
17	9	10	3	0:00	0:40	3:27	0:00	1:24	5:37
18	10	5	3	0:00	1:10	4:14	0:00	1:28	5:09
19	6	8	4	0:00	1:02	3:03	0:00	1:42	7:04

Figure 3.4: Station Report

### 3.3.8 Energy Usage

The vehicles' energy use is modeled using the following factors: changes in kinetic energy, overcoming air resistance, overcoming rolling resistance, changes in potential energy, power-train efficiency, and regenerative braking efficiency. The last two factors are not modeled directly, but have an effect on the total energy usage. Positive energy usages are divided by the power-train efficiency to account for inefficiencies, and negative energy usages are multiplied by the regenerative braking efficiency to account for energy recovery. Power usage is sampled at 1 Hz (by default) and is integrated to provide the total energy usage for the vehicle (Figure 3.5).

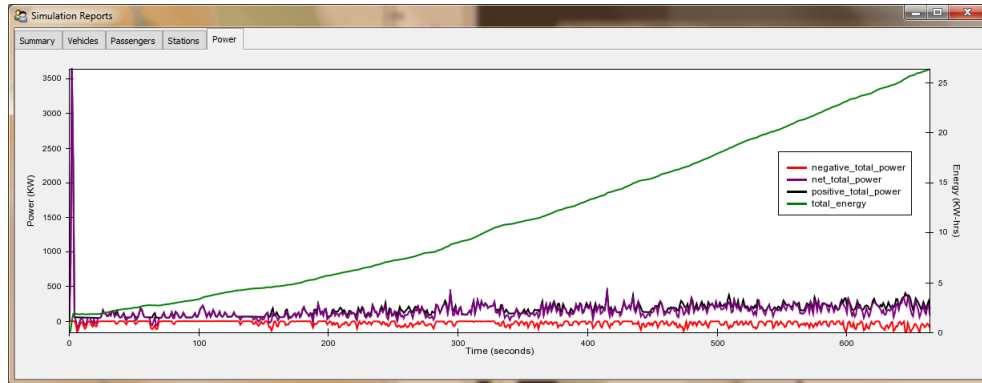


Figure 3.5: Power & Energy Report. In this case, the power requirements during normal operation are dwarfed by the startup, during which approximately 70 vehicles are simultaneously accelerated to line speed (not recommended).

### Kinetic Energy

Instantaneous power is positive when the acceleration is in the same direction as the vehicle's movement and negative when in the opposite direction (i.e. braking).

For a 1D system, power usage stemming from changes in kinetic energy is given by:

$$P_{ke} = Fv = mav$$

where  $m$  is the total mass of the vehicle and passengers,  $a$  is acceleration, and  $v$  is velocity.

### **Potential Energy**

The scenario file provides track elevation values at intervals along each track segment. An estimated track elevation is found at any position along the track segment by linear interpolation. The change in gravitational potential energy is given by:

$$\Delta U = mg\Delta h$$

where  $m$  is the mass of the vehicle in kg,  $g$  is the acceleration due to gravity, and  $\Delta h$  is the change in elevation since the last sample time. The power required for the change in potential energy is estimated by:

$$P_{pe} = \frac{mg\Delta h}{rate}$$

where  $rate$  is the sample rate in Hz.

### **Air Resistance**

The scenario file provides values for air density,  $\rho$ , the vehicle's frontal area,  $A$ , and its coefficient of drag,  $C_d$ . It also provides values for wind speed and wind direction.

Wind has an effect on energy usage by changing the apparent velocity of the vehicle<sup>2</sup>.

The power required to overcome aerodynamic drag is given by:

$$P_d = \mathbf{F}_d \cdot \mathbf{v} = \frac{1}{2} \rho v^3 A C_d$$

### Rolling Resistance

Rolling resistance is modeled with a velocity independent rolling coefficient,  $C_{rr}$ , supplied by the scenario file. The instantaneous power required to overcome rolling resistance is given by:

$$P_{rr} = F_{rr} v = C_{rr} N_f v = C_{rr} m g v$$

where  $F_{rr}$  is the rolling resistance force,  $v$  is the vehicle velocity,  $N_f$  is the normal force,  $m$  is the vehicle mass, and  $g$  is gravitational acceleration. Note that the track is assumed to be perpendicular to the ground for the purpose of calculating normal force.

---

<sup>2</sup> $v$  is set to 0 when the vehicle is stopped, regardless of wind speed.

## Chapter 4

# Controllers

### 4.1 Overview

The simulator has been designed from the start to cleanly divorce the vehicle control from the simulation. A vehicle controller is a standalone program that communicates with the simulator via a network port. This allows the vehicle control logic to be implemented in a variety of languages, and to run on any hardware with adequate network support.

The ultimate goal of a controller is to move passengers from their origin stations to their destination stations. In order to achieve this goal, the controller needs to accomplish the following general tasks:

- Route empty vehicles to useful destinations.
- Route occupied vehicles to their passengers' destinations.

- Adhere to hardware imposed limits on velocity, acceleration, and jerk at all times.
- Adhere to passenger-comfort imposed limits on acceleration and jerk during normal operation.
- Prevent vehicle collisions.

While accomplishing the above tasks, a successful controller should also strive to achieve the following performance goals:

- Minimize passengers' average trip time.
- Minimize the maximum trip time and trip time variance.
- Maximize system's passenger capacity.
- Maximize ride comfort.
- Minimize energy usage.
- Minimize number of transfers required to complete a trip.
- Be robust under failure conditions.
- Guarantee safety under all conditions.

The project includes two controllers at this time. The “PRT Controller” implements a PRT service mode, where vehicles are routed on-demand and there is no fixed schedule. The “GTF Controller” is designed to simulate conventional mass transit, in which relatively high capacity vehicles travel fixed routes on a fixed schedule.

GTF stands for Google Transit Feed, which is a data format for publishing schedule and route information, which is indirectly used by this controller.

## **4.2 PRT Controller**

### **4.2.1 Overview**

This vehicle controller implements a PRT service model, in which vehicles do not adhere to fixed routes or a fixed schedule, and in which vehicles are generally not shared. If vehicles are shared, it is only between passengers having the same origin and destination stations.

The controller uses a "quasi-synchronous" approach, uses shortest-path vehicle routing, has a simple demand-based algorithm for empty vehicle management, and moves vehicles into or out of storage based on the current number of waiting passengers.

The controller is centralized and monolithic, in that there is only one process that controls all of the vehicles in the sim, including all routing and merging tasks.

### **4.2.2 Usage**

Controllers are normally started via the Simulation's GUI, with arguments supplied by the scenario's configuration file.

Starting the controller manually can only be done from an SVN checkout, and not from the installed version. Open a command prompt (Windows) or terminal (OS X, Linux) and navigate to the controller's directory. To see what options are available

enter:

```
python prt_controller --help
```

When started, the controller will try to connect to the simulator on the default network port. The simulator must already be running and be ready to accept connections on the given port. The simulator may be manually set to accept connections by using an option under the “debug” menu. If the simulator is not accepting connections, the controller will exit with an error message.

### 4.2.3 Routing

The controller uses a weighted directed graph representation of the track network in order to route vehicles to their destinations. Each node of the graph is a track segment; each edge represents a connection between two segments. Edge weights represent the length of the track segment at the edge’s tail. Vehicle routes are chosen as the shortest path, as found using a bidirectional variant of Dijkstra’s algorithm.

In the case that the vehicle with passengers on-board is waved off from entering a station, the vehicle is sent on a loop so that it returns to the station at a later time. To accomplish this, we find the shortest path from the vehicle’s current location to the current location’s preceeding segment, then append the route from the current location to the destination station. In the case where the vehicle’s current location has multiple predecessor segments (i.e. the current location is immediately downstream of a merge), the the shortest path is found to each of the predecessor segments and the path with the lowest total length is chosen. Each component of a track network’s graph is required



to be strongly connected, so a path from the current location to the predecessor must exist.

#### 4.2.4 Empty Vehicle Management

To decide where to send empty vehicles, the controller uses a calculation that is based on stations' demand for additional vehicles and on the distance from the vehicle to the stations.

Condition	Demand Formula
<i>All berths are full or reserved</i>	$-\infty$
$pax > vehicles$	$pax - vehicles$
$pax \leq vehicles$	$(load\_berths - vehicles) / (load\_berths + 1.0)$

Table 4.1: Station's vehicle demand under various conditions.  $Pax$  is the number of passengers waiting at the station,  $vehicles$  is the number of vehicles with reserved berths (vehicles make a berth reservation shortly before entering a station), and  $load\_berths$  is the total number of berths in the station capable of embarking passengers.

A station's demand for additional vehicles is calculated based on Table 4.1. The first formula is intended to discourage empty vehicles from routing to a station that is already overloaded with vehicles. The second formula is the number of passengers in excess of the number of available vehicles. Each excess passenger provides one unit of station demand. The third formula is intended to route vehicles to stations which are low on vehicles when there are no excess passengers nearby. The formula is the fraction of unoccupied load berths, with the denominator slightly increased so as to keep the demand in the range  $[0,1)$ .

When an empty vehicle is choosing a station to travel to, the station demands

are multiplied by a distance factor  $k$  where:

$$k = \text{max\_dist} / (9 * \text{dist} + \text{max\_dist})$$

$\text{max\_dist}$  is the path distance from the vehicle to the furthest reachable station, and  $\text{dist}$  is the distance to the station under consideration.  $k$  takes on a value of 1 when  $\text{dist}$  is 0 and asymptotically approaches 0.1 as  $\text{dist}$  approaches  $\text{max\_dist}$ . The magnitude of  $k$  is relatively sensitive to small differences in distance when the distance is small, and relatively insensitive when distance is large.

Since station demand varies over time, empty vehicles reassess their destination periodically as they travel. Rather than recalculating on a fixed time or distance schedule, a vehicle will recalculate whenever it approaches a track split, using the principle that the best time to update information is when you are able to act upon it.

#### 4.2.5 Merging

The point at which two lines of tracks converge is referred to as a merge. There are three main types of merging:

- Main line merges
- Station merges outside a merge controller's Zone of Control
- Station merges within a merge controller's Zone of Control

## Main Line Merges & Merge Controllers

When two “main” (i.e. non-station) lines of traffic are merged, the vehicles from the two tracks are zippered together at speed rather than stopping one line of traffic or the other. A smooth zippering requires that appropriate gaps between vehicles be established in advance. The adjustments to relative vehicle positions required to create the gaps are orchestrated by a merge controller.

Each merge controller has a zone of control (ZoC). The ZoC starts at the merge point and extends upstream until it reaches the nearest non-station switch or merge point. The ZoC extends an equal length up both upstream legs, terminating at the shorter of the two legs (Figure 4.1). Switches and merges caused by stations are ignored when determining the ZoC length.

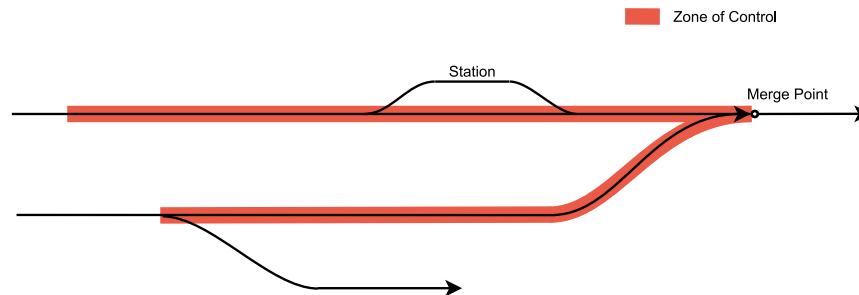


Figure 4.1: Extent of a Merge’s Zone of Control.

When a vehicle enters a merge controller’s ZoC a merge slot is created and added to the merge controller’s reservation queue (Figure 4.2). The reservation queue is a FIFO queue containing all upcoming merge slots. Each merge slot contains a time span in which the vehicle will have exclusive access to the merge point. Vehicles are

allocated merge slots on a first come, first served basis. The time span is calculated as:

$$t_{start} = \max(t_{current} + \frac{length_{ZoC}}{v_{line}}, t_{last})$$

$$t_{end} = t_{start} + headway_{min} + \frac{length_{vehicle}}{v_{line}}$$

where  $t_{last}$  is the end time for the last element<sup>1</sup> of the reservation queue, and  $v_{line}$  is the line speed.

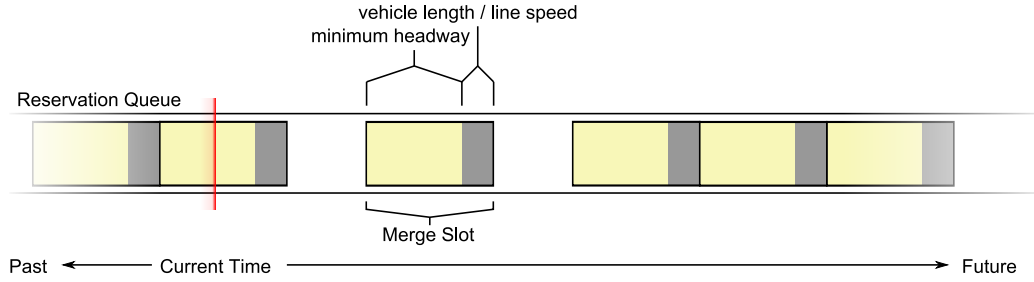


Figure 4.2: Merge slots within a merge's reservation queue.

In addition to the time span, merge slots also contain a reference trajectory (in the form of a cubic spline) that will cause the vehicle to arrive at the merge point at the allotted time. The spline is created using the Trajectory Solver's target time function (see Chapter 5), which adjusts the cruising velocity down-wards, if necessary, so that the vehicle arrives at the merge point at the correct time.

Vehicles which are bound for a station within the ZoC still request a merge slot upon entering the ZoC. This is to ensure that they may be safely waived off from the station if the station is full or otherwise too congested. If the vehicle successfully

<sup>1</sup>i.e. the merge slot that was most recently added and is furthest from the reaching the merge point.

enters its desired station within the ZoC, the merge slot is marked as 'relinquished' but is not removed from the merge's reservation queue.

Vehicles which adjust their velocity do so only once they've fully entered the merge's zone of control and they return to line speed prior to passing through the merge point. Thus, any trajectory adjustments necessary for merging are limited to within the merge's ZoC and do not have an effect on vehicle flow outside of it.

Vehicles entering a merge controller's zone of control begin to slow down (if necessary) when they reach a fixed point on the track. Because of this, the tip-to-tip headway between the slowed vehicles remains unchanged in the time domain, but the tip-to-tip distance is reduced<sup>2</sup>. As the tip-to-tip time separation goes to 0, so does the tip-to-tip distance separation. Because vehicles are not zero length, however, the tip-to-*tail* distance will go negative. Vehicles which slow using the same deceleration profile, begun from the same starting point, cannot reduce their velocity to below  $\frac{\text{length}}{\text{headway}}$  without having a collision, where length is the lead vehicle's length, and headway is the tip-to-tip headway.

Since vehicles utilize the entire length of the ZoC to get into position for the merge, and because the ZoC is typically quite long, the vehicles' change in velocity is usually very small. Imposing a minimum velocity limit within a ZoC is planned, but the situation described above has not been observed to occur in any reasonable simulations.

---

<sup>2</sup>To see this, consider that vehicle A and vehicle B are both traveling at 10 m/s and have a tip-to-tip separation of 1 second. Vehicle A reaches the mark at  $t_0$  and slows to 5 m/s in one second. At  $t_1$  vehicle B reaches the mark and slows using the same deceleration profile. At  $t_2$ , vehicle B will be in the exact place that vehicle A was at at  $t_1$ . At  $t_2$ , vehicle A will have moved 5 meters ahead since it completed its slowing maneuver. The two vehicles tip-to-tip time separation is unchanged at 1 second, but their tip-to-tip distance separation has been reduced from 10 meters to 5 meters.

## Station Merge Outside a ZoC

Outside of a merge controller's zone of control, vehicles travel at a predetermined velocity, referred to as the line speed. To exit a station, a vehicle accelerates to line speed on a separate line, referred to as the station's on-ramp, then merges into a gap in the vehicles on the main line.

Rather than actively creating a gap on the main line for the launching vehicle to merge into, this controller waits for a natural gap to occur. There are several reasons for this decision:

1. Simplicity. Vehicles on the main line are allowed to travel at a constant velocity when on the main line without need to slip forward or backward.
2. Natural gaps are common. Due to the algorithm used, vehicles get tightly packed when traveling through a main line merge. This packing causes many small gaps to coalesce into fewer, larger gaps. Additionally, the station launch algorithms also create tightly packed groups of vehicles. In practice, it's rare for vehicles to end up with small gaps that could be feasibly exploited to create a larger gap.
3. Congestion management. If a line is so congested that naturally occurring gaps are rare or nonexistent, forcing additional vehicles onto the line encourages problems.

Detecting a gap to launch into is achieved by first calculating the length of time that the launching vehicle will require to reach the station's merge point. The distance that a vehicle on the main line would travel in that time is calculated, and

the corresponding position on the upstream track is determined. The upstream point is expanded to account for vehicle length and desired headway to create a “window”. If no vehicles are found in the window, then the launch vehicle may launch immediately and will have an unobstructed merge onto the main line. If one or more vehicles are found in the window, then the time required for them to clear the window is calculated, and the window is rechecked when that time has elapsed. Meanwhile, the launching vehicle stays stopped within the station. This continues until the launching vehicle finds the window to be clear, at which time it launches.

### **Station Merge Within a ZoC**

When a station is located within a merge controller’s zone of control, launching a vehicle from the station can no longer rely on simply checking the upstream track for conflicts. In this case, the vehicle also needs to ensure that there won’t be a conflict at the main line merge further downstream, which it does by acquiring a merge slot prior to launching from the station.

Merge slots are typically created and granted on a first-come, first-served basis as vehicles enter the ZoC at the upstream boundary. As the slots are created, they are appended to the end of the merge’s reservation queue. In the case where a vehicle is launching from a station located within a ZoC, one approach might call for a merge slot to be created as though the vehicle had just entered at the upstream boundary, so that the vehicle’s merge slot is appended to the end of the reservation queue like normal. Like all merge slots, it would be accompanied by a vehicle trajectory, albeit

one that it not being used. The launch of the vehicle from the station could be arranged so as to synchronize with the slot’s trajectory at the station’s merge, then utilizing that trajectory for the remainder of its journey to the merge point. The downside of this approach is that, while simple, every vehicle must wait for a phantom vehicle to progress from the edge of the ZoC to the station merge point before merging onto the main line. This delay reduces the rate at which vehicles can leave the station and thus reduces station throughput.

An improved approach, and the one used by this controller, is to first look for a (time-based) gap in the merge’s reservation queue into which a new merge slot could be inserted. To ensure that a gap may always be found, a dummy slot whose merge time is set to infinity is temporarily appended to the reservation queue. In order for a gap to be considered “usable”, it must have the following properties:

- Reachable: The launching vehicle must not be required to travel faster than line speed in order to reach the merge point within the gap’s time span.
- Sufficient Width: The gap in the reservation queue must be at least  $headway_{min} + \frac{length_{vehicle}}{v_{line}}$  seconds wide.
- Unblocked: If the launch vehicle would need to overtake a vehicle on the same leg of the merge in order to reach the merge point within the gap’s time span, then the gap is considered blocked and is unusable.

Note that a usable gap that doesn’t involve the dummy slot generally only occurs when traffic on the main line is very light and/or vehicles on the main line are



traveling at close to full speed. To see this, recall that the usable gap does not refer to a momentary gap between vehicles, but to a gap that *cannot be closed* by the following vehicle—the following vehicle always travels at full speed when there is a gap in the reservation queue.

If a usable gap in the reservation queue is found, consider the two merge slots that bracket the gap, slot A and slot B and the vehicles that are utilizing the slots, vehicle A and vehicle B. Consider also that there is a point on the mainline track, just upstream of the station’s merge point, that marks the dividing line between, “A vehicle can immediately launch from the station and safely merge onto the main line ahead of Vehicle A” and “A vehicle must delay launching, so as to merge behind Vehicle A.” We refer to this point the decision point.

In order for vehicle A to be past the decision point and traveling at a greatly reduced speed, there must have been a long period in which no other vehicles entered the ZoC.

There are three cases:

1. Vehicle A is past the decision point.
2. Vehicle A has not yet reached the decision point, slot B is *not* the dummy slot.
3. Vehicle A has not yet reached the decision point, slot B *is* the dummy slot.

In the first case, the vehicle wishing to leave the station may do so immediately. If by traveling at full speed the launching vehicle will reach the main merge point within

the usable gap, then it does so. If it can't launch immediately, it delays until it can make the run at full speed, then launches.

In the second case, the vehicle wishing to leave the station must wait some time for vehicle A to reach the decision point, but the situation is thereafter handled the same as for the first case.

In the third case, a phantom vehicle is created that is treated as though it entered the merge's ZoC shortly after vehicle A. A trajectory is generated for the phantom vehicle that takes it all the way from the edge of the ZoC to the merge point. Because vehicle A has not yet passed the decision point, the launch vehicle is guaranteed to be able to synchronize with the phantom's trajectory. The launch vehicle waits until the phantom trajectory reaches the decision point, synchs with the phantom at the station merge point, then follows its trajectory to the merge point.

#### **4.2.6 Station Behavior**

##### **Finite State Machine**

Vehicle behavior within a station is governed by a finite state machine (Figure 4.3). Vehicles are in the RUNNING state when travelling between stations. Upon arriving at a station, if the vehicle has passengers on board then it will alternate between the UNLOAD ADVANCING and UNLOAD WAITING states as it makes its way to an unload berth. Otherwise it will go to a loading berth, alternating between LOAD ADVANCING and LOAD WAITING in the process. Assuming that it has passengers,

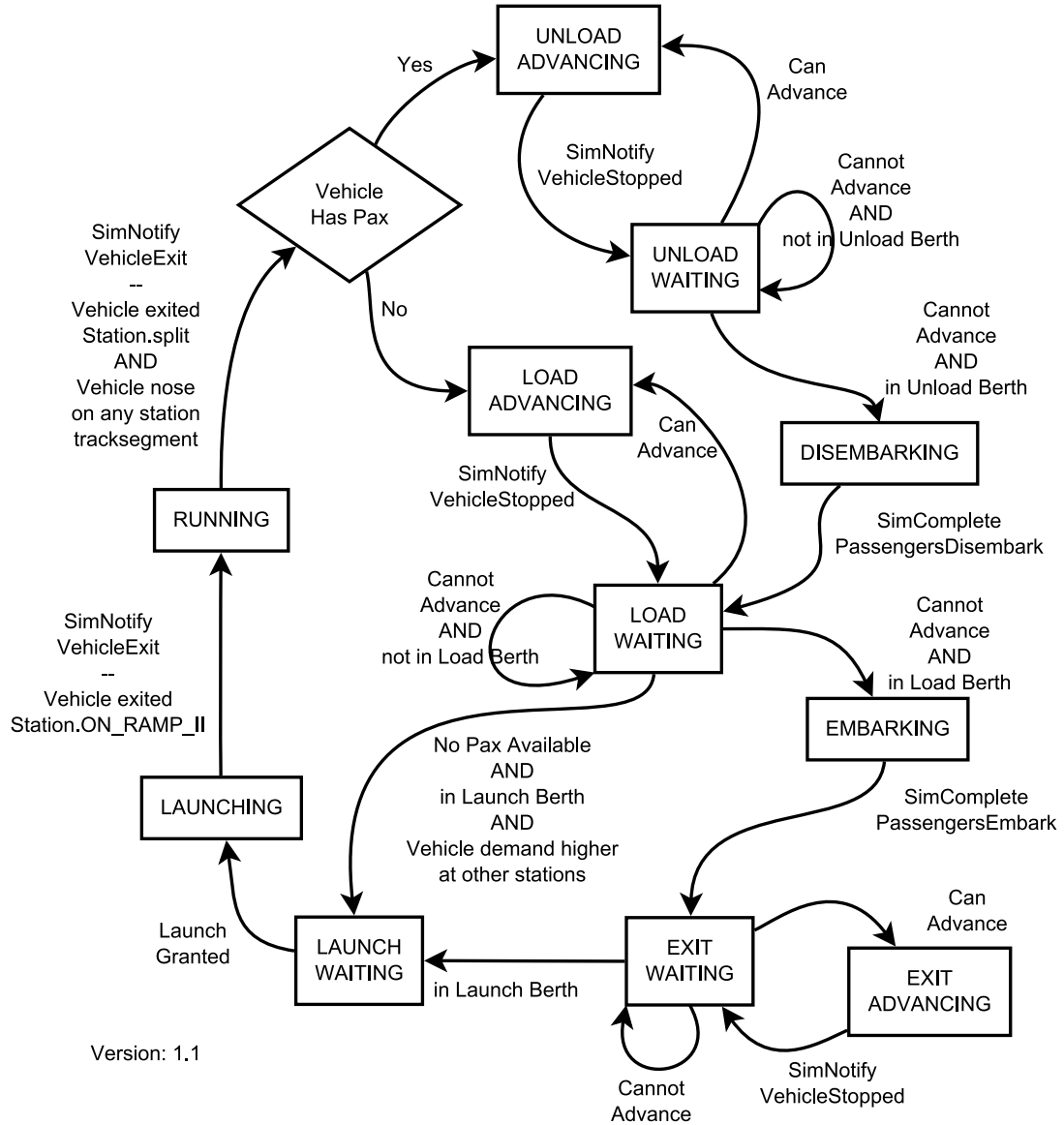


Figure 4.3: State machine used to govern vehicle behavior within a station.

once the vehicle is unable to advance any further, or once it has reached the front-most unloading berth, the vehicle will enter the DISEMBARKING state and begin to unload its passengers. Upon receiving the message stating that the passenger(s) have disembarked, the vehicle advances to the front-most loading berth that it can reach. If there are passenger(s) available, the vehicle will enter the EMBARKING state and begin to load the passenger. Once loading has completed, the vehicle alternates between the EXIT WAITING and EXIT ADVANCING states until it reaches a launch berth. It enters the LAUNCH WAITING state and requests a launch time from the station (the details of which differ depending on the station's proximity to a merge). When the launch time arrives, the vehicle enters the LAUNCHING state and accelerates up to line speed and merges with the main line. Once the vehicle is fully on the main line, it enters the RUNNING state until it reaches its destination station and exits the main line again.

An attempt to transition out of an X WAITING state is triggered whenever another vehicle in the station moves (including exiting the station).

### **4.3 GTF Controller**

The GTF Controller emulates traditional mass transit, moving vehicles over fixed routes on a fixed schedule. The controller moves the vehicles according to their schedules, and attempts to move passengers to their destination stations.

The name comes from “Google Transit Feed,” which is a set of files describing

a transit agency’s routes and schedules. The GTF Controller does not use these files directly, but uses information that has been extracted from them and included in a TrackBuilder scenario file (See Appendix A for format).

### 4.3.1 Vehicle Movement

Vehicles routes are drawn from the scenario file, but they are only specified as a sequence of station ids with an arrival and departure time. The Simulator requires that vehicle routes be communicated as a sequence of TrackSegment ID’s. To plan the finer-detail routes that the Simulator requires, the GTF Controller builds a graph representation of the track network, and uses a bidirectional variant of Dijkstra’s shortest path algorithm to connect the station ids.

GTF Controller uses the same trajectory generation code as the PRT Controller. The “target time” functionality is used to create a trajectory that carries a vehicle from one station to the next in accordance with the schedule. Under heavy passenger loads, a vehicle’s schedule may slip—even at the vehicle’s maximum speed, it is unable to reach the next stop by the desired arrival time. In this case, the “target time” trajectory will fail and a best-effort minimum-time trajectory (“target position”) will be used in its place.

Note that these trajectories are not very realistic, in that they do not include the effects of traffic congestion, do not obey any traffic laws (e.g. speed limits, stop lights, etc.), and may even cause collisions between the transit vehicles. All these effects are ignored, on the assumption that the GTF Controller will usually be working with real

transit schedules from transit agencies, and that the expertise of their schedule planners can be relied upon to create feasible and realistic schedules. Thus, a fine-grained model of the transit vehicles' movement is not necessary for verifying the reasonableness of the schedule.

## **Energy Usage**

The trajectories used by the GTF controller will approximately match the average speed used by the real vehicles in daily operation (assuming that the real vehicles approximately adhere to their schedules), but the simulated vehicles will travel non-stop between stations. This will bias the estimated energy usage of the simulated vehicles downwards in two ways.

First, unless a vehicle is using regenerative braking, whenever a vehicle stops it will lose its kinetic energy as heat. Any unscheduled stops that occur during actual vehicle operation, such as stopping at a traffic light, will incur an energy cost that is not captured by the simulation. Since traditional mass transit vehicles have a high mass, this bias in energy usage is likely quite significant.

Second, a vehicle that has unscheduled stops will need a higher maximum speed in order to match the average speed of a vehicle with no additional stops. In addition to the higher speed exacerbating the bias described above, higher speeds lead to increased aerodynamic drag forces and rolling resistance, further increasing energy usage.

Due to the biases described above, it is recommended that the energy usage

estimates for GTF controlled vehicles be considered as a lower bound, rather than as accurate values<sup>3</sup>.

### 4.3.2 Passenger Routing

Routing passengers through a fixed schedule, fixed route system requires that arrival and departure times be taken into account, and may require that passengers transfer between vehicles. The GTF controller uses a graph to represent the transit network’s schedule and chooses routes based solely on the total trip time (that is, it does not penalize transfers). The controller can handle the case where a passenger must reschedule due to the desired vehicle being full.

#### Time-Expanded Weighted Digraph

The transit network’s schedule is represented by a time-expanded, weighted digraph. The “time-expanded” part means that each station is represented by a set of nodes in the digraph, rather than a single node (Figure 4.4). Each node in a station’s set represents a vehicle arrival or departure time, with one additional node that is time-independent. Each time-dependent node within a station’s set has an edge connecting it to the next arrival or departure time node for the station, with an edge weight equal to the time difference between the two nodes. These edges represent waiting at the station. Nodes within the station’s set also have a 0-weight edge to the time-independent node.

---

<sup>3</sup>The estimates should not be even be considered lower bounds unless accurate vehicle parameters are provided, of course. Though some attempt was made to choose default parameters that are reasonable, they should not be relied upon as such. The parameters can be adjusted via the VehicleModels screen in TrackBuilder.

Vehicle trips are represented by edges from one station's time-dependent node to another station's time-dependent node, with the edge weight being equal to the trip time.

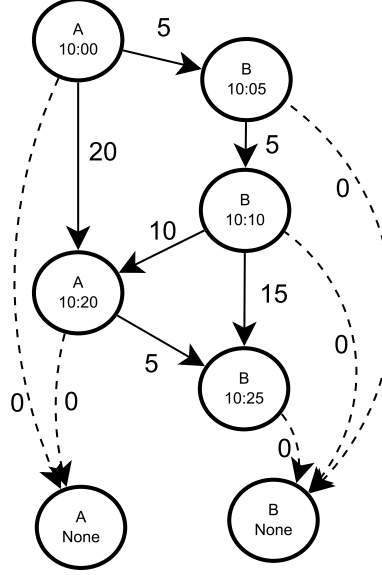


Figure 4.4: A time-expanded digraph representing the schedule for a two-station network. Edge weights represent trip times. Vehicles depart from Station A at 10:00 and 10:20 and arrive at Station B at 10:05 and 10:25 respectively. A vehicle departs from Station B at 10:10 and arrives at Station A at 10:20. The vertical edges represent waiting at a station. The time-independent nodes represent a passenger's ultimate destination.

## Passenger Itineraries

Recall that passengers are created during the course of the simulation, and that each passenger has an origin and destination station. When the GTF Controller receives notice that a passenger has been created, it attempts to craft a passenger itinerary that will route the passenger to its destination. The itinerary begins with the origin station's first time-dependent node to occur after the passenger's creation time. A shortest-path is found from this node to the time-independent node of the destination



station. The nodes of the path are examined in a pairwise fashion in order to create a list of embarking, and disembarking actions. The complete itinerary is stored with the passenger.

To simplify vehicle logic, the passenger actions are also stored in conjunction with the time-dependent nodes of the graph so that there's one centralized place for a vehicle to ask, "Who do I pick up, drop off?" upon arriving at a station.

### **Replanning Itineraries**

If more passengers want to embark on a vehicle than its capacity allows, the excess passengers are rejected and must replan their trips. The itinerary stored with the passenger is used to find and remove the passenger's now-invalid embarking and disembarking actions from the graph. Then the itinerary is cleared, and it is replanned using the station's next time-dependent node as a starting point. The current time-dependent node isn't used in order to prevent the replanning from using the already rejected vehicle.

## Chapter 5

# Trajectory Generation

Trajectories are generated using a set of five procedures: “target acceleration”, “target velocity”, “target position”, and “target time”. The choice of which procedure to use depends on which of the four final state variables are to be specified, where the four state variables are: position, velocity, acceleration, and time. With the “target acceleration” procedure, only the final acceleration is specified. With the “target velocity” procedure, both the final acceleration and the final velocity must be specified. “Target position” requires that the final acceleration, velocity, and position be specified, and “target time” specifies all of the final state variables.

The trajectories generated are, in most cases, the unique trajectories that reach the desired final state in minimum time, while obeying constant-value limits on velocity, acceleration, and jerk<sup>1</sup>. The limits used are specified by the `VehicleModel`,

---

<sup>1</sup>Minimum time does not apply to the target time procedure, since both the initial and final times are specified. Also, see section 5.6 for an exception to the minimum-time criteria.

and are usually chosen so as to achieve some desired level of passenger comfort or to obey some hardware specifications. In order to achieve the minimum time goal, the trajectories are always ride a constraint. Usually this is the jerk limit, but during some segments the trajectory will hit an acceleration or velocity limit and jerk will fall to 0.

## 5.1 Splines

The trajectories are derived from splines, which are piecewise polynomial functions. This project uses  $C^2$  cubic splines, meaning that spline's polynomials are third order and that the spline's first and second derivatives are continuous. In the context of the project, the use of  $C^2$  cubic splines means that jerk can change instantaneously, but acceleration, velocity, and position cannot.

A polynomial is defined as:

$$P_i(t) \triangleq p_0 + p_1t + p_2t^2 + p_3t^3 \quad (5.1)$$

where  $p_0, p_1, p_2, p_3$  are the polynomial coefficients and

$$P'_i(t) = p_1 + 2p_2t + 3p_3t^2 \quad (5.2)$$

$$P''_i(t) = 2p_2 + 6p_3t \quad (5.3)$$

A spline is defined as a collection of polynomials:

$$S(t) \triangleq \begin{cases} P_0(t) & t_0 \leq t < t_1 \\ P_1(t) & t_1 \leq t < t_2 \\ \vdots & \\ P_{k-1}(t) & t_{k-1} \leq t \leq t_k \end{cases} \quad (5.4)$$

and the  $k + 1$  points  $t_i$  are referred to as knots, and the tuple  $\mathbf{t} = (t_0, \dots, t_k)$  as a knot vector. The  $k$  intervals between knots,  $t_i - t_{i-1}$  for  $i = 1, \dots, k$  form the tuple  $\mathbf{h} = (\Delta t_{01}, \Delta t_{12}, \dots, \Delta t_{(k-1),k})$ , known as the knot interval vector. Knot intervals are not allowed to be negative.

At times we use the notation that  $v(t) = \dot{x}(t)$ ;  $a(t) = \ddot{x}(t)$ ;  $j(t) = \dddot{x}(t)$  where  $v$ ,  $a$  and  $j$  stand for velocity, acceleration, and jerk, respectively. In general, numeric subscripts refer to a knot index, thus  $a_1$  refers to the acceleration at  $t_1$ .

### 5.1.1 Constraint Equations

This section will show that if the jerk profile is assumed (i.e. the 3rd order polynomial coefficients are assumed to be known), then there are sufficient constraint equations to drive the degrees of freedom to zero, that is, to determine a unique solution. The trajectory generation in this project attempts to create a unique, minimum-time spline from a set of boundary conditions and constraints on the maximum and minimum jerk, acceleration, and velocity values. Given that a cubic polynomial is determined by 4 coefficients, for a cubic spline with  $n$  segments and with knot vector  $\mathbf{t} = (t_0, \dots, t_n)$ ,

there are  $4n + (n + 1) = 5n + 1$  unknowns.

For a  $C^2$  spline, we require that  $P_{i-1}(t_i) = P_i(t_i)$ ,  $P'_{i-1}(t_i) = P'_i(t_i)$ , and  $P''_{i-1}(t_i) = P''_i(t_i)$  for  $i = 1, \dots, n - 1$ . This provides  $3(n - 1)$  constraints. We assume that the vehicle's initial position, velocity, acceleration and time are known:  $P_0(t_0) = x_0$ ,  $P'_0(t_0) = v_0$ ,  $P''_0(t_0) = a_0$ ,  $t_0 = t_0$  which provides 4 additional constraints.

We also assume that the jerk coefficient (third order) is known for every polynomial which provides  $n$  additional constraints. Finally, in some cases we assume that the acceleration coefficient (second order) and velocity coefficient (first order) are known for some polynomials in the spline. These assumptions will be discussed in more detail where they are applied.

## 5.2 System of Equations

The initial and final conditions for a polynomial's valid domain are related by:

$$\begin{bmatrix} x(t_f) \\ v(t_f) \\ a(t_f) \\ j(t_f) \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} & \frac{\Delta t^3}{6} \\ 0 & 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(t_i) \\ v(t_i) \\ a(t_i) \\ j(t_i) \end{bmatrix} \quad (5.5)$$

where  $\Delta t$  is the knot interval or duration of the polynomial.

Target	Profile	n	Ukn	$C^2$	Init	Final	Jerk	Accel	Vel
Accel	H	1	6	0	4	1	1	0	0
Vel	HZL	3	16	6	4	2	3	1	0
Vel	HL	2	11	3	4	2	2	0	0
Pos	HZLZLZH	7	36	18	4	3	7	3	1
Pos	HZLZLH	6	31	15	4	3	6	2	1
Pos	HLZLZH	6	31	15	4	3	6	2	1
Pos	HZLZH	5	26	12	4	3	5	2	0
Pos	HLZH	4	21	9	4	3	4	1	0
Pos	HLZ	3	16	6	4	3	3	0	0
Pos	HLH	3	16	6	4	3	3	0	0
Time	HZLZLZH	7	36	18	4	4	7	3	0
Time	HZLZLH	6	31	15	4	4	6	2	0
Time	HLZLZH	6	31	15	4	4	6	2	0
Time	HLZLH	5	26	12	4	4	5	1	0
		n	$5n + 1$	$3n - 3$	4		n	varies	varies

Table 5.1: Number of unknown variables and the number of constraint equations. *Target*: The procedure used. *Profile*: The assumed jerk profile.  $H = j_{max}$ ,  $Z = 0$ ,  $L = j_{min}$ .  $n$ : # of polynomials.  $Ukn$ : # of unknowns in the system of equations.  $C^2$ : # of constraint eqns arising from continuity requirement. *Init*: # of constraint eqns from initial conditions. *Final*: # of constraint eqns from final conditions. *Jerk*: # of constraint eqns from known jerk coefficients. *Accel*: # of constraint eqns from known acceleration coefficients. *Vel*: # of constraint eqns from known velocities.

## 5.3 Trajectory Procedures

The following subsections describe the procedures used to generate the trajectory splines. Each procedure follows the general pattern:

1. Assume that the spline is some number of segments, with known jerk profile. Start with the maximal number of segments allowed for the procedure (see Table 5.1), which assumes that the trajectory reaches acceleration and/or velocity limits.
2. Starting from the boundary conditions and using the known limit values, work towards the center of the spline while solving for the knot intervals  $\mathbf{h}$ .
3. If any knot intervals are found to be negative it is because the trajectory was assumed to reach an acceleration or velocity limit, but in reality it does not. Choose a new profile that does not assume the appropriate limit was reached and re-solve.

The jerk profiles given in Table 5.1 assume that the target value (be it acceleration, velocity, position, or time) is increasing. If the target value instead decreases from the initial value, the jerk profile is vertically mirrored.

The procedures make very few assumptions. Though it is common for initial and final accelerations to be zero, for example, the procedures do not assume that they will be. The maximum and minimum limits on velocity, acceleration, and jerk are not necessarily symmetrical about 0.

## 5.4 Target Acceleration

This section discusses creating a spline with a specific final acceleration, but whose final position, velocity, and time are unconstrained.

This is the simplest of all cases, and requires only a single cubic polynomial<sup>2</sup>. As is the case with all of the procedures, the initial state is known, and with this procedure, only the final acceleration is specified. Referring to eqn (5.2), we can see that we can solve for the unknown knot interval  $\Delta t$  with:

$$a(t_f) = a(t_i) + j(t_i)\Delta t \quad (5.6)$$

or, solving for the variable of interest,

$$\Delta t = \frac{a(t_f) - a(t_i)}{j} \quad (5.7)$$

where  $j(t_i)$  has been shortened to  $j$  since it is constant for a given polynomial. The choice of  $j$  is given by:

$$j = \begin{cases} 0 & \text{if } a(t_f) - a(t_i) = 0 \\ j_{max} & \text{if } \text{sign}(a(t_f) - a(t_i)) \text{ is positive} \\ j_{min} & \text{otherwise} \end{cases}$$

---

<sup>2</sup>In some cases it may be necessary to use multiple polynomials in order to satisfy velocity constraints. For example, if the target acceleration is positive, but the initial velocity is already at the maximum velocity limit, then a valid spline requires two segments: a negative jerk segment that moves the velocity away from the limit, followed by a positive jerk segment to reach the target acceleration. Because it is rare to command an acceleration without regard for position or velocity, this procedure is rarely used and it was decided not to handle those cases—if a one-segment spline leads to a constraint violation, the procedure simply raises an error.



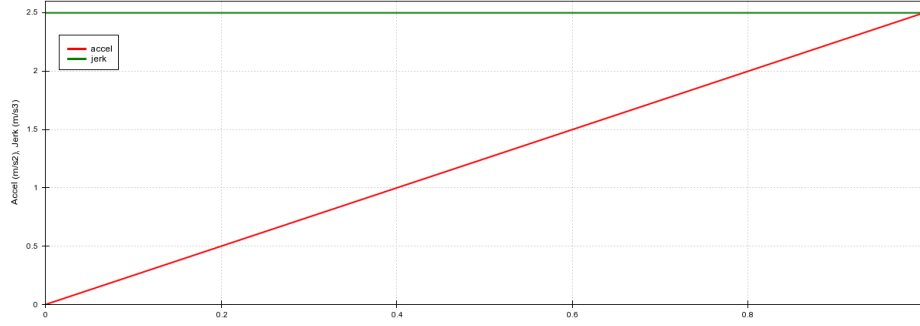


Figure 5.1: A single polynomial spline targeting an acceleration.

## 5.5 Target Velocity

This section discusses creating a spline with a specific final acceleration and velocity, but whose final position and time are unconstrained. In the general case, this task requires a spline with three polynomial segments. Conceptually, the vehicle will (i) change it's acceleration until it hits an acceleration limit (ii) spend some time changing velocity at the acceleration limit (iii) change it's acceleration again so as to hit the desired final acceleration, which corresponds to the three polynomial segments. Achieving the target acceleration may be accomplished by considering only the final polynomial, but the  $\Delta v$  is affected by every segment of the spline. In the case where an acceleration limit is not reached, the middle segment will have a zero duration.

### Acceleration Limit Reached

In this case we assume that the known acceleration limit is reached, and that the spline will consist of three polynomial segments. Examining Figure 5.2 it is clear that

the knot intervals of the first and last polynomial segment may be found by application of Eqn 5.7. From the known jerk, and from having solved for the knot interval of the first segment, the initial position, velocity, and acceleration of the middle segment may be found by application of Eqn 5.2. Working backwards from the other end, the knot interval of the third segment can be found, as can the final conditions for the middle segment.

At this point, it is entirely possible upon application of Eqn 5.2 to the middle segment with the solved-for initial and final conditions, that the  $\Delta h$  (i.e., the knot interval of the middle segment) will be found to be negative. If this occurs, it is because the acceleration limit is not actually reached. This case is handled below.

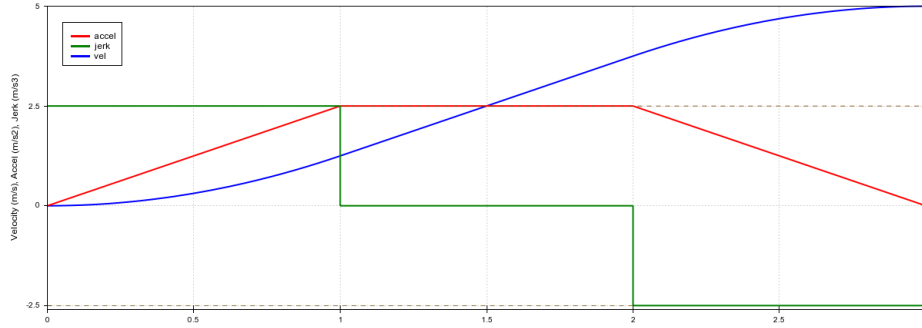


Figure 5.2: A three segment spline targeting a velocity.

### Acceleration Limit Not Reached

If the assumption that the acceleration limit was reached proves to be incorrect, then the problem is resolved assuming a profile like that shown in Figure 5.3. Note that

the figure shows a case where symmetry can be used to simplify the problem, but the target acceleration procedure is capable of handling the general case as well.

$$h_0 = (a_1 - a_0)/j_{max} \quad (5.8)$$

$$v_{01} = \frac{j_{max}h_0^2}{2} + a_0h_0 \quad (5.9)$$

$$h_1 = (a_2 - a_1)/j_{min} \quad (5.10)$$

$$v_{12} = \frac{j_{min}h_1^2}{2} + a_1h_1 \quad (5.11)$$

Solving for the peak acceleration  $a_1$  yields:

$$0 = a_1^2 \left( \frac{1}{2j_{min}} - \frac{1}{2j_{max}} \right) + v_2 - v_0 + \frac{a_0^2}{2j_{min}} - \frac{a_2^2}{2j_{min}} \quad (5.12)$$

Using the quadratic formula results in two solutions for  $a_1$ , the larger one is used. With  $a_1$  known, and with a known jerk, the knot durations can be found. From there, the final state variables can also be found.

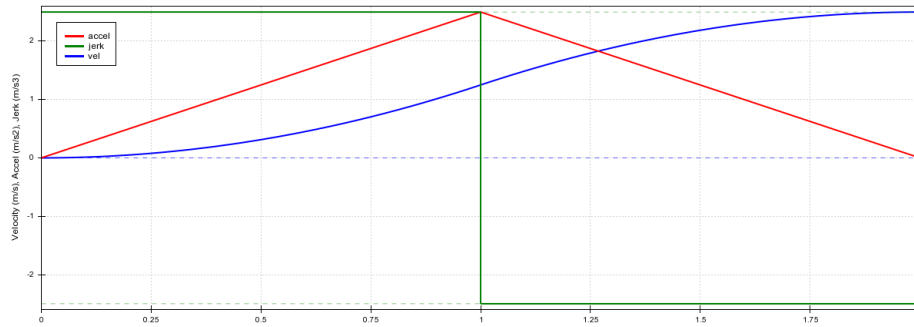


Figure 5.3: A two segment spline targeting a velocity value.

## 5.6 Target Position

### Velocity Limits Reached

In this case it is assumed that the spline has the maximum seven segments, and both the acceleration and the velocity limits are reached (Figure 5.4). Using the assumption that the middle segment is a constant velocity segment at  $v_{max}$ , the “target velocity” procedure can be reused to solve the first three segments completely. The last three segments can be found in a similar manner, with minor alteration. The “target velocity” procedure expects fully specified initial conditions, but the position and time at the beginning of the fifth segment are not yet known. Dummy values are used in place of the missing initial values with the understanding that the absolute values will be incorrect but the relative values (i.e. delta position) will be correct. The remaining unknown segment distance can be found  $\Delta x_{34} = x_f - x_i - \Delta x_{03} - \Delta x_{47}$ , and from it and the known velocity, the remaining unknown knot interval can be found  $h_{34} = \frac{\Delta x_{34}}{v_{max}}$ .

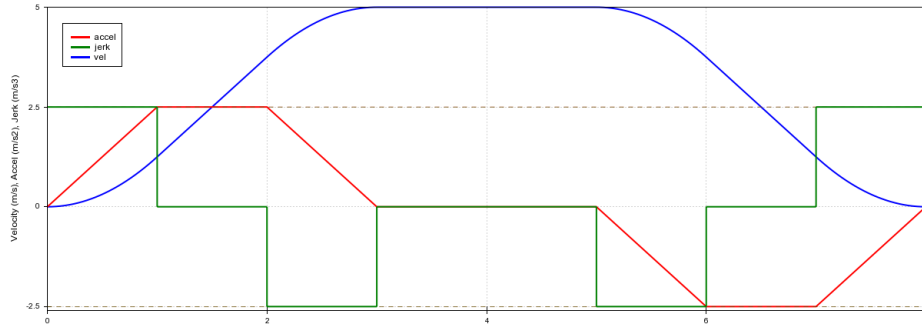


Figure 5.4: A seven segment spline targeting a position. Both velocity and acceleration limits are reached.

Although the above description assumes a full seven segments with acceleration limits being reached, the use of the “target velocity” procedure allows the same approach to work when acceleration limits are not reached. All that is required is for there to be a middle segment for which the velocity can be assumed to be a known constant, and acceleration to be zero.

### **Velocity Limit Not Reached, Acceleration Limits Reached**

In this case it is assumed that the spline has a maximum of 5 segments, and that only acceleration limits are reached.

In the subsection above, where the velocity limit *is* reached, the presence of a 1st order polynomial (with a known velocity coefficient) in the middle of the spline allowed the problem to be neatly divided into two halves that can be solved (mostly) independantly. This case, however, is not so amenable.

In this case, then entire five segment spline is coupled. The following quadratic equation for  $h_3$ , the knot interval for the second constant-acceleration segment, is by no means optimized. Once  $h_3$  has been found, the remainder of the spline may be easily deduced.

$$\begin{aligned}
\alpha &= \frac{v_4 - v_1}{a_{max}} + \frac{a_{max}}{2j_{min}} - \frac{a_{min}^2}{2a_{min}j_{min}} \\
0 &= h_3^2 \left( \frac{a_n}{2} - \frac{a_n^2}{2a_x} \right) + \\
&\quad h_3 \left( -v_1 + \alpha(a_{min} - a_{max}) + \frac{a_{min}^2}{2j_{min}} + \frac{a_{max}^2}{2j_{min}} + \frac{a_{min}v_1}{a_{max}} - \frac{a_{min}a_{max}}{j_{min}} \right) + \\
&\quad q_4 - q_1 - \alpha v_1 - \frac{a_{max}\alpha^2}{2} + \left( \frac{\alpha a_{max} + v_1}{j_{min}} \right) (a_{max} - a_{min}) + \\
&\quad \frac{-a_{max}(a_{min} - a_{max})^2}{2j_{min}^2} - \frac{(a_{min} - a_{max})^3}{6j_{min}^3}
\end{aligned}$$

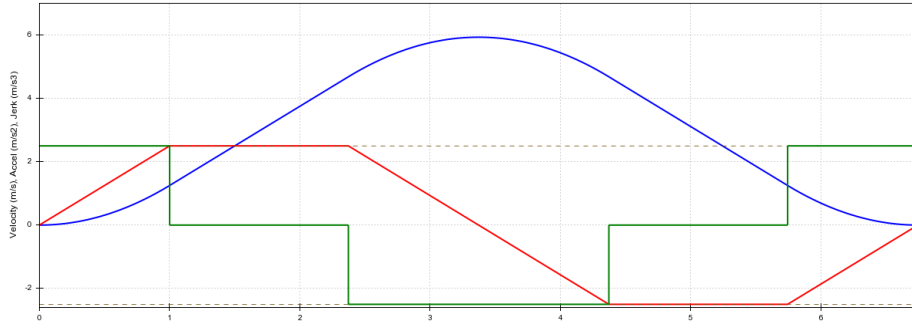


Figure 5.5: A five segment spline targeting a position. Acceleration limits are reached, but velocity limits are not.

### One Acceleration Limit, or No Limits Reached

The remaining cases for the “target position” procedure are handled by a fallback approach that uses a non-linear function minimizer<sup>3</sup> to generate a valid spline.

The splines generated by this approach still obey the velocity, acceleration, and jerk

<sup>3</sup>Minimizer uses Powell’s method; implementation is part of the code library SciPy.

constraints, and they still match the initial and final states, but they no longer make claims of being time-optimal.

The minimizer operates on the knot interval vector for a five segment spline and tries to drive an error function to zero. The error function is calculated by creating a spline from the initial state, the assumed jerk profile<sup>4</sup>  $(j_{max}, 0, j_{min}, 0, j_{max})$ , and the current guess for knot intervals. Then an error value is calculated using the penalties:

- $p_h$  – Negative knot interval penalty. Negated sum of any negative knot intervals.
- $p_{al}$  – Acceleration limit penalty. Sum of amounts by which spline exceeds accel limits at each knot.
- $p_x$  – Final position penalty. Difference between the target final position and the spline’s final position.
- $p_v$  – Final velocity penalty. Same as above, for final velocity.
- $p_a$  – Final acceleration penalty. Same as above, for final acceleration.

The error value returned is the result of the formula:

$$error = p_h + p_a + p_x^2 + p_v^2 + p_a^2$$

Note that there is no penalty for the sum of the knot intervals, that is, there is no direct penalty for creating longer than necessary splines. It was decided to omit such a penalty so that the minimizer would have a clear stopping point—when the

---

<sup>4</sup>As always, the profile is vertically mirrored if  $x(t_f) < x(t_i)$

error value has fallen to zero, a valid spline has been produced. The tendency for the minimizer to create longer than necessary splines is mitigated by the built-in assumption that the splines will always use maximal/minimal jerk values.

As an initial guess for the knot interval vector uses the assumption that  $a(t_i) = a(t_f) = 0$  and  $v(t_i) = v(t_f) = 0$  are zero, even if this is known not to be true. These assumptions give an initial guess of:

$$s = \sqrt[3]{\frac{x(t_f) - x(t_i)}{2j_{max}}} \quad (5.13)$$

$$[h] = (s, 0, 2s, 0, s) \quad (5.14)$$

Because the minimizer is not a global minimizer, it is possible that it will become trapped in a local minima and be unable to find a valid spline when one does exist. To combat this, the program wraps the minimization routine in a loop that will make several attempts. If the first guess fails, subsequent calls use random values choosen from the range (0,1) for the initial knot intervals.

Since this sub-procedure is relatively computationally expensive, and since the resulting splines are not guaranteed to be minimal-time, the controller tries to avoid situations where its use would be required.

## 5.7 Target Time

The final spline generation procedure targets a final time in addition to position, velocity, and acceleration. The assumed jerk profile is similar to seven segment profile as used for the “target position, velocity limits reached” sub-procedure except



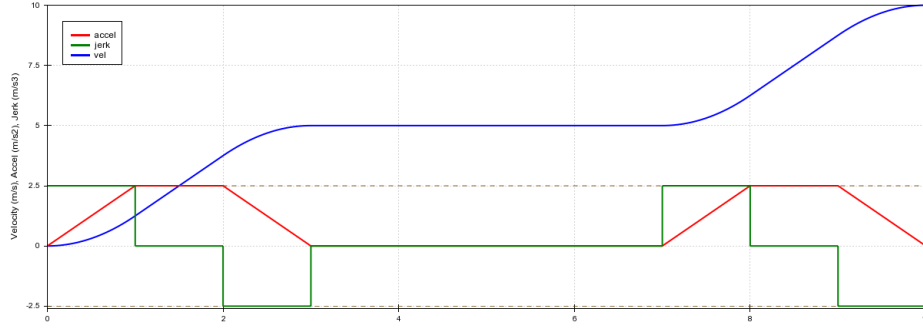


Figure 5.6: A seven segment spline targeting a time. Acceleration limits are reached.

the middle, constant velocity segment is not assumed to reach the velocity limit (see Figure 5.6. The middle segment’s velocity is referred to as the “cruise” velocity and is determined during the course of the procedure. Another important difference is that the “target position” spline would always begin with an increase in speed (the first three segments) and end with a decrease in speed (the last three segments). With “target time”, the cruise velocity may be either above or below the initial or final velocities, and so any of the four combinations of increases and decreases in speed may be used. The four profiles will be referred to as  $+/+$ ,  $+/-$ ,  $-/+$  or  $-/-$ <sup>5</sup>

To ensure that a viable trajectory exists, the “target position” procedure is invoked using the same initial and final conditions (sans time). If the spline created has a final time that is greater than the target final time, then the target state cannot be reached without violating some constraint and the procedure fails.

In order to make an initial guess at which of the four profiles to use, a special

---

<sup>5</sup>An example: If the initial velocity is 10, the final velocity is 0, and the cruise velocity is found to be 5, then the  $-/-$  profile would be used.

spline that targets the final position, but which does not alter the initial velocity is created. This spline is not a minimum-time spline, it simply travels along at the initial velocity until the final position has nearly been reached, then accelerates or decelerates as necessary to hit the final velocity and acceleration. If this spline reaches the target position before the target time, then we know that we should begin our maneuver by slowing down, i.e. use one of the two  $-/*$  profiles. Otherwise, we know to use one of the two  $+/*$  profiles.

To decide which of the remaining two profiles to use, the average speed is calculated,  $v_{ave} = \frac{x(t_f) - x(t_i)}{t_f - t_i}$ . If the target velocity is less than  $v_{ave}$  then we choose the  $*/-$  profile, otherwise the  $*/+$  profile is used. The average speed is being used as an estimate for the desired cruise velocity, but discrepancies between the two may lead to the wrong profile being used. If the procedure fails with the chosen profile, the other profile is tried before giving up entirely.

With the profile known, a valid spline can be generated (See Appendix C) under the assumption that the acceleration limits will be reached. In the cases where acceleration limits are not reached, it is apparent from Table 5.1 that there will still be a unique solution. Finding the solution in these cases is made more difficult, however, by the necessity of finding cubic roots (other cases have only required finding quadratic roots). Rather than solve these cases directly, a work-around is used.

The work-around is to artificially reduce the acceleration limits, until the assumption that the acceleration limits are reached is once again true. Lowering the limits requires some care; if the limits are made too low, then their may be insufficient control

to reach the target state (including target time). That is, if the limits are lowered too far, the vehicle may not be able to speed up or slow down quickly enough to reach the desired position, velocity, and acceleration by the desired time. Furthermore, the reduced acceleration limit for the first half of the profile may need to be different than for the last half (Figure 5.7).

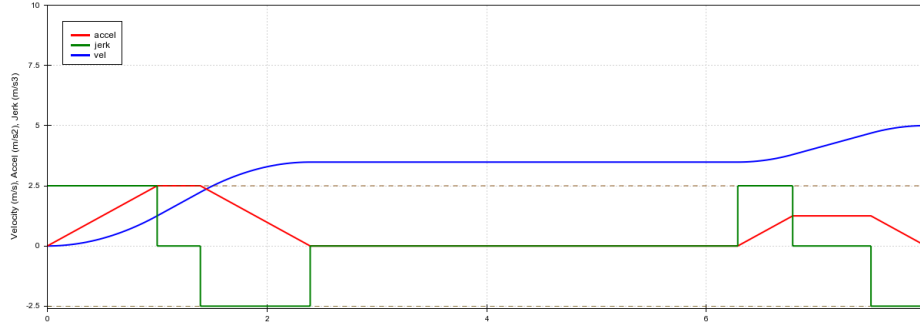


Figure 5.7: A seven segment spline targeting a time. The acceleration limit for the last half of the profile was reduced.

To find appropriate limits, an iterative approach is used. A trial spline is created, and if it is found to have a negative duration (a sign that the *assumption* that the acceleration limit would be reached was wrong) then the corresponding acceleration limit is reduced and a new trial spline is created. If after reducing a limit, a trial spline is unable to be created (due to lack of sufficient control), the limits are increased. The search for correct limits for the first half and the last half of the profile occur simultaneously.

In practice, this approach is very effective at finding a valid spline if one exists. The amount by which the limits change is halved upon each reversal of search direction,

allowing it to rapidly home in on the appropriate limits. In the current implementation, the number of steps is capped at 30, which allows the limit to be resolved to within  $\frac{1}{2^{30}} \approx 1 \times 10^{-8}$ th of the range.

# Chapter 6

## Discussion

### 6.1 Future Work

#### 6.1.1 Spiral Curves

TrackBuilder’s use of constant-radius curves may not be appropriate for high speed and/or short wheelbase vehicles. Recall that a constant radius curve (a circular arc) will impose a constant lateral acceleration on vehicles that traverse it at constant speed. Unless vehicles have a suspension system capable of damping lateral motion, transitions between straight and curved track segments will have infinite jerk as the lateral acceleration instantly transitions between zero and the curve’s acceleration<sup>1</sup>. Quite some work went into ensuring that longitudinal jerk could be constrained to reasonable values, and the use of constant-radius curves, while simple, partially defeats

---

<sup>1</sup>Assuming that the vehicles have no length, that is. Longer vehicles will experience a lower, but potentially still unacceptably high, lateral jerk

the work that went into that constraint.

In place of constant-radius curves, TrackBuilder should use so-called spiral curves, in which the radius gradually decreases coming into a curve, and gradually increases coming out of a curve. Intuitively, spiral curves match the experience felt when riding in an automobile. A driver does not instantly change the position of the steering wheel, but rather gradually turns it as he navigates a corner.

### 6.1.2 LQR Trajectory Generation

The procedures used to generate trajectories are, in most cases<sup>2</sup>, time optimal given the constraints used. If a trajectory is not viable under the current constraints, however, there is no capability to violate constraints “softly”, that is, violate them just enough to produce a viable trajectory. The procedures will instead fail completely, and need to be retried with looser constraints—which they will then aggressively exploit. Furthermore, the current procedures do not allow trajectories to be optimized for anything but minimum time. Though time is an important consideration, the ability to minimize some combination of time, power, and ride comfort would be preferred.

The case where system dynamics are described by a set of linear differential equations and cost can be described by a quadratic function is called the LQ problem. Finding an optimal solution to an LQ problem can be accomplished with the use of a Linear-Quadratic Regulator (LQR) or by solving a two-point boundary value problem. Since the existing trajectory generation code is well-isolated from the rest of the project,

---

<sup>2</sup>As discussed in Chapter 5, some cases fall back to a non-time-optimal approach.

it would be straight-forward to replace it with an LQR-based approach or some other optimal control method if preferable.

### 6.1.3 Safety Checks

The Simulator currently has no built-in checks for the safety of vehicle trajectories, aside from detecting collisions. The most conservative check would be to ensure that vehicles always maintain a so-called “Brick Wall Stopping Distance” between them. The name for this distance comes from the thought experiment, “If the leading vehicle were to suddenly transform into a brick wall, at what distance would the following vehicle need to be in order to avoid a collision?” If acceleration can change instantaneously and there is no delay, then this distance is simply  $x_{min} = v/a_{min}$  where  $v$  is the following vehicle’s velocity and  $a_{min}$  is its minimum acceleration (maximum deceleration). The formula becomes more complicated if a time delay is added and if there is a finite jerk, but the principle remains the same. A more flexible safety check assumes that the leading vehicle cannot stop instantly, but has some finite failure deceleration  $a_f$ . The brick wall assumption becomes a special case wherein  $a_f$  is negative infinity.

### 6.1.4 Nonlinear Stations

There are many potential station designs that cannot be simulated with the current implementation. In order to accomodate most station layouts, the Simulator (and ideally, TrackBuilder as well) would need to support the following addiitons:

1. Offline berths. Vehicles do not obstruct the station’s track while in a berth.

- (a) Parallel berth. Vehicles may pull forward to exit the berth.
  - (b) Perpendicular berth. Vehicles must reverse to exit the berth. If the vehicle is symmetrical, it may not need to reverse direction on the station's track.
  - (c) Angled berth. May be normal or reverse angle. In a reverse angle berth, the vehicle backs in.
2. Splits and Merges within the station. Allows for parallel platforms.
  3. Arbitrary number of additional connecting track pieces within a station.

#### **6.1.5 Simulated Latency**

Adding communications latency to the simulator will not require significant technical changes. Messages emanating from the Simulator will be delayed prior to being sent; messages received by the Simulator will be delayed prior to being processed. Most of the machinery required to simulate latency is already in place.

The more difficult task is altering the controller(s) to tolerate latency. How much alteration is required will depend in part upon how much latency is introduced, but will also depend upon how much noise and error is introduced (see next section). To compensate for latency, the controller will need to make predictions about vehicles' future states. If noise, error, and latency are all low then a relatively simplistic predictor can be implemented, but if they are high then the controller will need a more development in order to use more robust approaches.



### 6.1.6 Noise and Error

The Simulator currently operates as though vehicles are able to follow their reference trajectories perfectly, and that sensors provide precise and accurate measurements of vehicle position, velocity, acceleration, and of inter-vehicle distances. Under these highly unrealistic conditions, a controller is able to operate open-loop. That is, a controller can feasibly use its own internal model of the world to operate and does not need to incorporate any feedback from the simulator<sup>3</sup>.

To make the Simulator suitable as a testbed for feedback-based controllers, the simulator needs (optional) models for noise and error. Rather than reporting a vehicle's true state variables to the controller, it would first add some amount of probabilistic noise. At a minimum, it should be capable of adding white Gaussian noise, but the noise model should be implemented in such a way that it is easily modifiable. Also, rather than vehicles perfectly following the trajectories supplied by a controller, the Simulator should introduce some amount of error to the target speeds and accelerations. Again, the error model that is used should be easily modifiable.

### 6.1.7 Multi-lane tracks

At least three major implementations of PRT use vehicles that run on a surface, rather than on some form of rails<sup>4</sup>. One of the advantages that running on a surface offers is the ability to have multiple lanes of traffic, and for vehicles to move between the

---

<sup>3</sup>The controller can't be wholly open-loop since it must rely on the Simulator for information about passenger actions, but it can be close!

<sup>4</sup>2GetThere, ULTra, and the Morgantown PRT

lanes at any point along the guideway. This offers opportunities for increasing system capacity, and is an area that is apparently not well covered by the PRT literature.

### **6.1.8 Group Rapid Transit Controller**

Mentioned briefly in the Introduction, Group Rapid Transit (GRT) is a variant on the PRT service model that retains the on-demand scheduling and (usually) the on-demand routing, but uses larger vehicles that are shared by multiple parties. Trips are no longer non-stop to the destination, but allow for multiple stops to pick-up and drop off passengers.

Writing a controller that implements a GRT service mode would allow modeling of scenarios in which GRT is used to boost capacity. Especially interesting are scenarios in which GTF is used alongside PRT, so as to increase system capacity while retaining high quality of service for less-frequently used origin-destination pairs.

## **6.2 Postmortem**

### **6.2.1 What Went Right**

Have at least one client. Get them involved early. I am deeply indebted to Dr. Baertsch from Unimodal for his frequent feedback and testing of the project during its development. His participation as a client was extremely helpful throughout the project and often served as a motivating factor.

Unit tests. There's not enough of them, but where they were used, unit tests

were extremely helpful. Even just the philosophy that you should keep working to isolate a bug until you can craft a unit test that reproduces it is useful, in that it forces you to understand the bug well enough to recreate it, rather than just well enough to make it go away (hoping it doesn't come back). Once unit tests were in place, they were invaluable for ensuring that a difficult or tricky piece of code didn't break during further development.

### **6.2.2 What Went Wrong**

A Master's project is, by nature, not a team project—but that doesn't mean that working alone is necessarily a good idea. The project would have greatly benefited had at least one other person been heavily involved with the design and coding aspects. Were it not a solo project, the design work could have benefited from another mind checking for flaws and looking for improvements from the start. Likewise for code implementation—what I would have given to have been able to do regular code-reviews with someone intimately familiar with the code-base and the languages used! The separation between simulation and control meant that this project would have been particularly well suited for parallel development.

Feature Creep. A common problem in any software project, feature creep struck this project particularly hard, and has been particularly damaging to both the project's timeline and its overall level of fit and finish. This was mostly caused by the usual reason—it's more fun to work on exciting new features than it is to work out the nagging issues with existing ones—but this natural tendency was amplified by

the nature of our weekly lab meetings. Progress reports were made weekly, but were never backed by detailed inspection in the form of demos or code-reviews. This meant that spending extra time to move a feature from merely “working” to “working really well” wasn’t particularly required or rewarded, and could even give the impression that productivity for a given week had been unusually low. However, having a new feature on hand to talk about would always make a good impression, and would make for a more interesting discussion. This extra incentive for the already dangerous tendency to flit from one shiny new feature to another was like giving caffeine to a squirrel.

Inability to schedule. This must be a deeply innate flaw, because after two years of practice I still can’t accurately estimate whether a feature will take a week or a month, and I’m *never* wrong in the “too conservative” direction. Even when I generously pad an estimate, I never really believe the padded value to be correct, and behave accordingly. This flaw was yet another enabler for feature creep.

Premature optimization? More like postpartum optimization. I’ve often read that premature optimization is practically a sin, that engaging in optimization early-on is liable to be a wasted effort. The logic is that without a working system to profile or benchmark, it is very difficult to predict where performance bottlenecks will actually occur. While perhaps true, this blinded me to the fact that assessing performance early on can be vital. It’s one thing to say, “I can make this faster” without first considering whether it needs to be faster, but it’s quite another to say, “I can make this faster later” when the initial performance is demonstrably not good. Because, sometimes, *you can’t*.

## 6.3 Final Conclusions

The prt-sim project has been a moderate success. It has an easy-to-use, if still a bit unpolished, scenario creation tool. It has a flexible simulator with relatively few arbitrary limitations. It has a PRT controller that achieves reasonable system performance and is capable of correctly and reliably operating on a wide-variety of track layouts. Finally, it has a mass-transit controller that demonstrates the flexibility of the simulator, and provides a first step towards modeling multi-modal transit networks that incorporate PRT.

Yet, despite all the work that has gone into the project, and despite all that has been achieved, it still barely scratches the surface of what could yet be done. I am determined that this paper shall not mark the end-of-life for this project, but will instead be just one of its many milestones. The “Future Work” section is a far from an exhaustive wishlist, but it does not include my number one goal—to contribute something *genuinely* useful to the PRT community. To help accomplish this, I intend to impose a geis against implementing new features and resolve to spend the time necessary to fix the numerous “papercuts” that hamper the project’s usability. Once it is robust and truly user-friendly, I will finally exit “stealth-mode” and actively encourage people to put it to use. The project has always been open source in principle; at this point I hope it will become open-source in practice as well—open to people’s contributions, and to them extending it for their own purposes.

## Appendix A

# Scenario File XML Schema

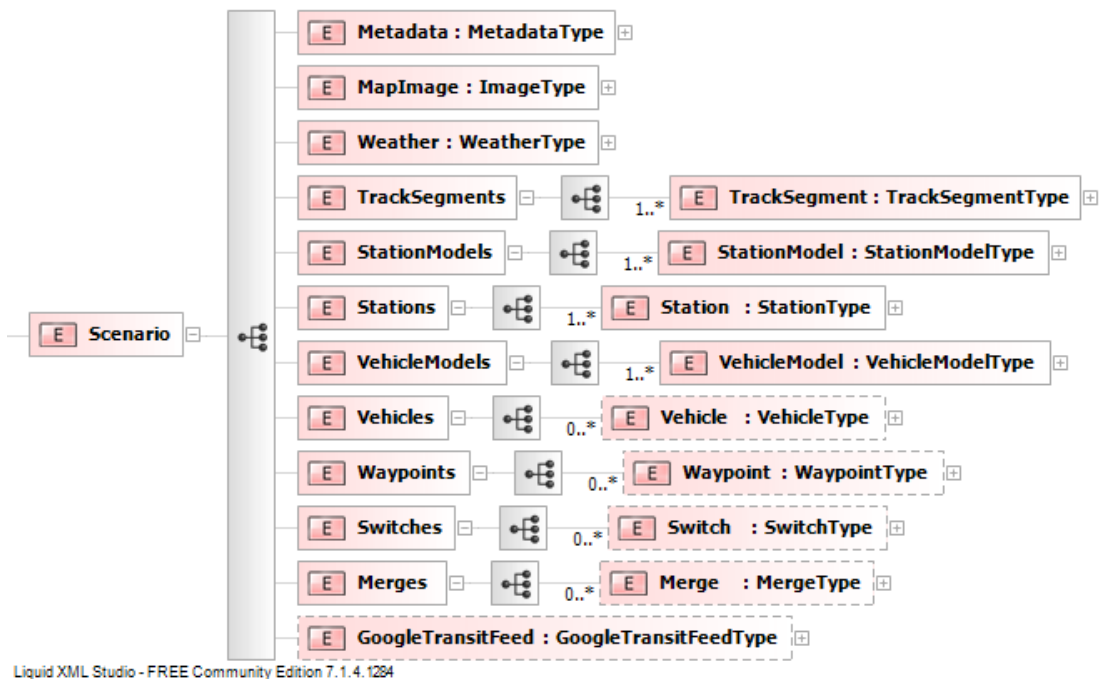


Figure A.1: Scenario, the root element.

This appendix contains an overview of the XML format used for scenario files

(aka TrackBuilder save files). An XML Schema describing the format may be found in the project's source tree at:

<http://code.google.com/p/prt-sim/source/browse/#svn/trunk/schema>

Values are reported in meters, meters per second, kgs, radians, or decimal degrees.

The root element of the XML document is the “Scenario” entity (Figure A.1). The Scenario element contains sequences of StationModels, VehicleModels, TrackSegments, Vehicles, Stations, Waypoints, Switches, and Merges. Scenario also contains elements for MetaData, a MapImage, Weather data, and (optionally) an element containing trip information for the GTF controller. Each of the sequences in Scenario contain elements of the corresponding type, i.e. the StationModels sequence contains elements that are of the StationModelType.

An element of the TrackSegmentType (Figure A.2) contains CoordinateType elements (described below) for the start and end endpoints; for curved TrackSegments it also includes one for the center point of the arc. It contains the ID's of other track segments that are connected to it and, for bidirectional track, the ID of the antiparallel TrackSegment. This type includes an “Elevations” element which is a sequence of CoordinateTypes from points along the TrackSegments length. This type requires “id” and “length” attributes, and includes “max\_speed” and “label” as optional attributes. Curved TrackSegments will include “radius” and “arc\_angle” attributes. The arc\_angle is positive for a segments that curve counter-clockwise, and negative for clockwise segments.

An element of the CoordinateType (Figure A.3) requires latitude and longitude

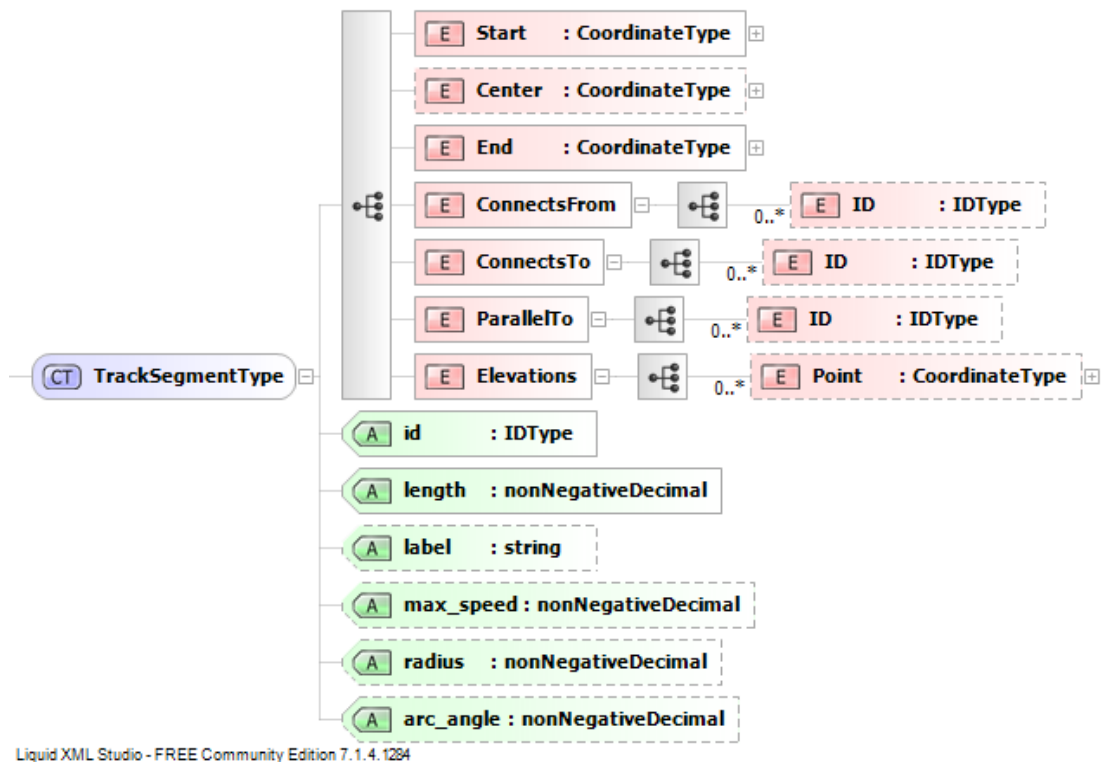


Figure A.2: TrackSegmentType

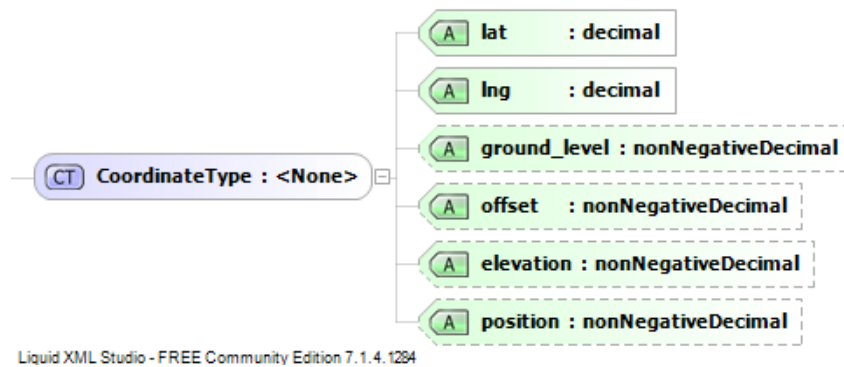


Figure A.3: CoordinateType



data, and may optionally include ground level (as meters above sea level), an “offset” value that indicates the height of the track, and an “elevation” value which is the elevation of the track (again, as meters above sea level). If all three of the optional values above are present, then elevation should be the sum of the other two. Another optional attribute is “position” which is the distance from the start of the TrackSegment.

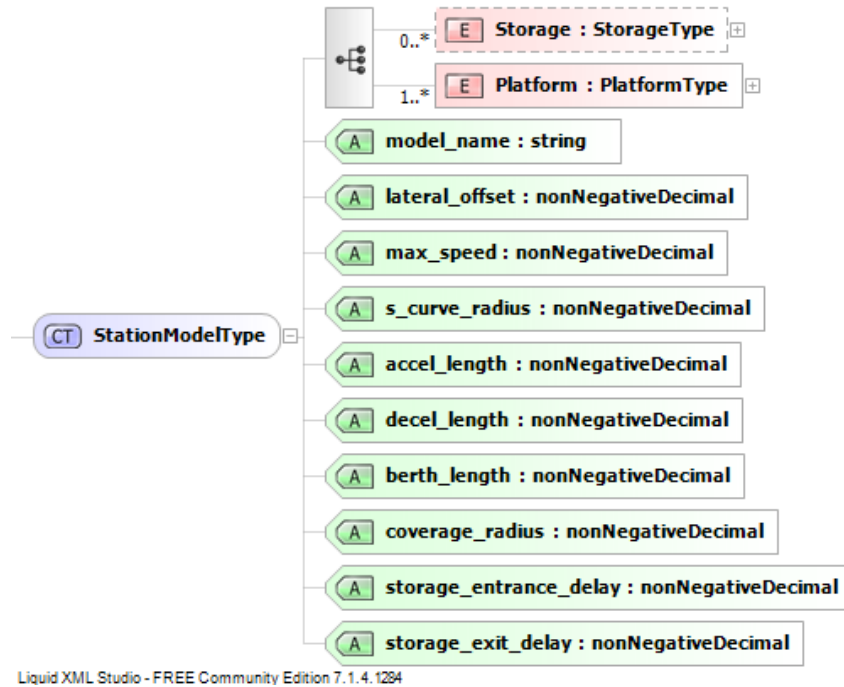


Figure A.4: StationModelType

An element of the StationModelType (Figure A.4) describes a station layout, as opposed to describing a particular station instance. Elements of this type must have at least one Platform element, and may include one or more Storage elements. All attributes shown in the figure are mandatory. The “model\_name” attribute refers to the StationModel name, and must be unique. The “lateral.offset” attribute gives the

horizontal distance between the main line and the station line.

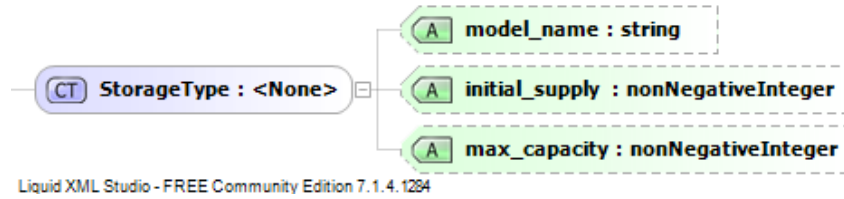


Figure A.5: StorageType

An element of the StorageType (Figure A.5) gives an initial quantity, and maximum limit for how many vehicles of the type specified by “model\_name” may be stored at the parent station type.

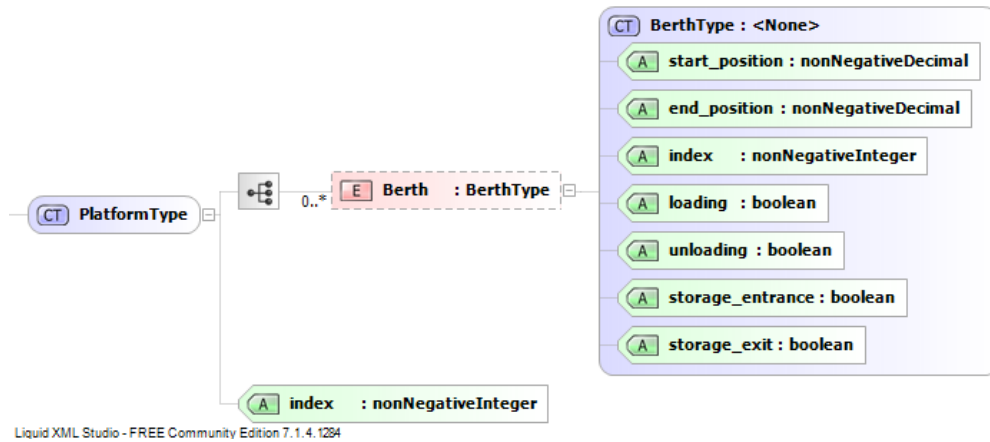


Figure A.6: PlatformType and BerthType

An element of the PlatformType (Figure A.6) may contain one or more elements of BerthType, and has a 0-based index as an attribute. The BerthType element also contains a 0-based index, which corresponds with the Berth’s order on the platform. The BerthType has four Boolean attributes where a “true” value indicates that

the Berth has the associated capability. “Loading” and “unloading” refer to passengers, and “storage\_entrance” and “storage\_exit” refer to moving vehicles into and out of the station’s vehicle storage area, respectively.

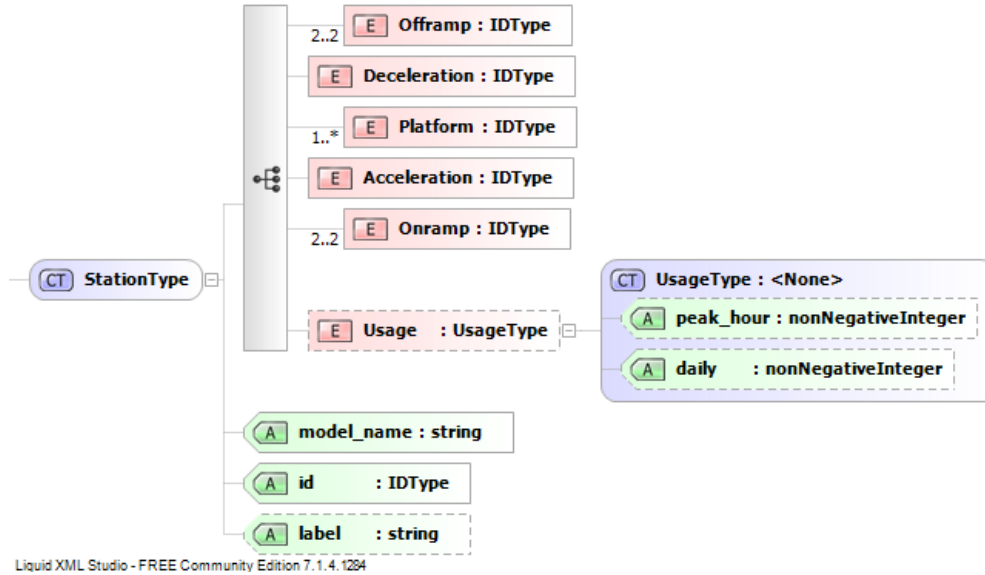


Figure A.7: StationType

A StationType (Figure A.7) element describes a station instance. Elements of StationType contain a sequence containing TrackSegment ID’s: two Offramp elements (an S-Curve), a Deceleration element, one or more Platforms, an Acceleration, and two Onramp elements (another S-Curve). The StationType may also contain a Usage element, which specifies “peak\_hour” and “daily” passenger volumes. This type’s “model\_name” attribute is required to match one of the StationModelType elements “model\_name” attribute.

A VehicleModelType (Figure A.8) element describes a particular vehicle model,

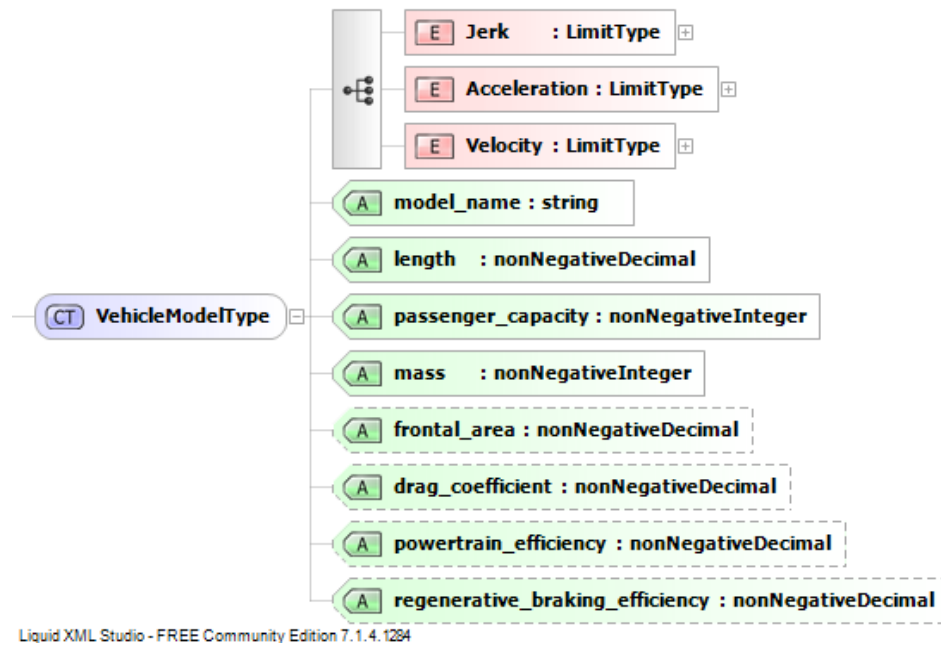


Figure A.8: VehicleModelType

rather than a vehicle instance. Elements of this type specify limits on the vehicle model’s velocity, acceleration, and jerk. The “powertrain\_efficiency” attribute is limited to the range  $(0,1]$  and the “regenerative\_braking\_efficiency” attribute is limited to the range  $[0,1)$ .

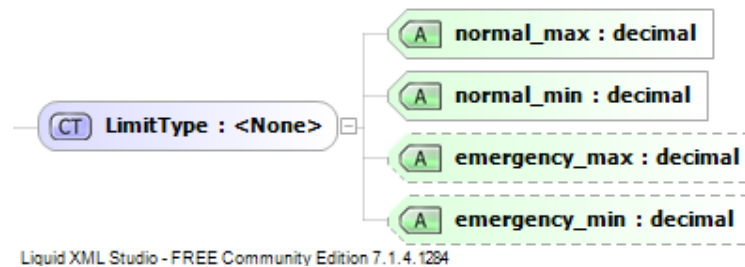


Figure A.9: LimitType

A LimitType (Figure A.9) element specifies up to four constant-value limits, one pair specifying limits for normal operation, and another pair specifying limits during non-normal operation or emergency maneuvers. The min values are often negatively signed, but are not required to be.

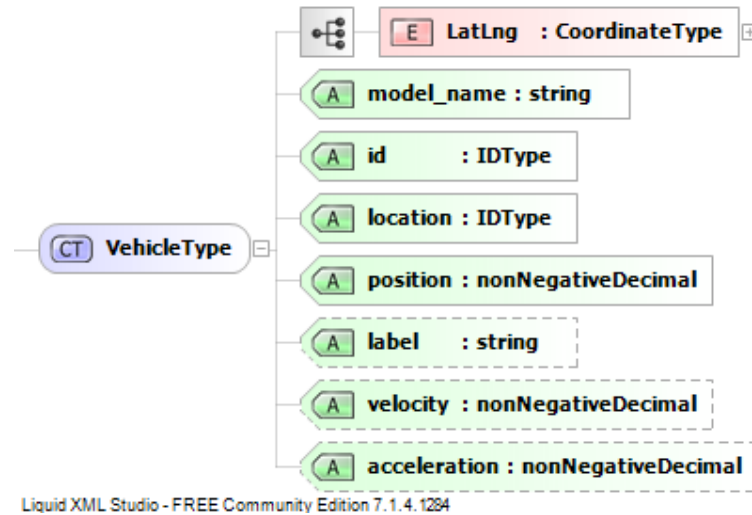


Figure A.10: VehicleType

An element of the VehicleType (Figure A.10) describes a particular vehicle instance. The “location” attribute always contains a TrackSegment’s ID, and the “position” specifies the position of the vehicle’s nose as a distance from the TrackSegment’s start. This type’s “model\_name” attribute is required to match one of the VehicleModelType elements “model\_name” attribute.

A WaypointType (Figure A.11) element describes the boundary between the TrackSegments identified by the Incoming and Outgoing elements. The “max\_speed” attribute is the maximum speed out of the Outgoing TrackSegment. MergeType (Figure

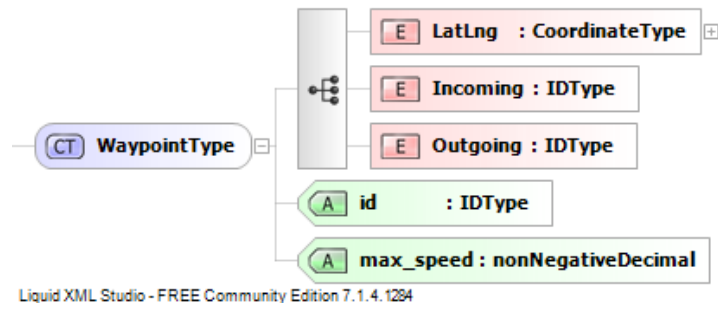


Figure A.11: WaypointType

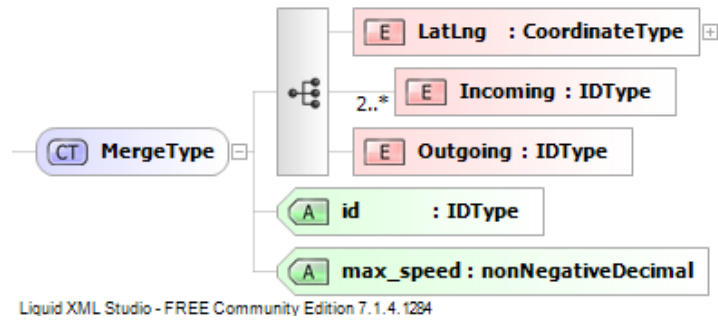


Figure A.12: MergeType

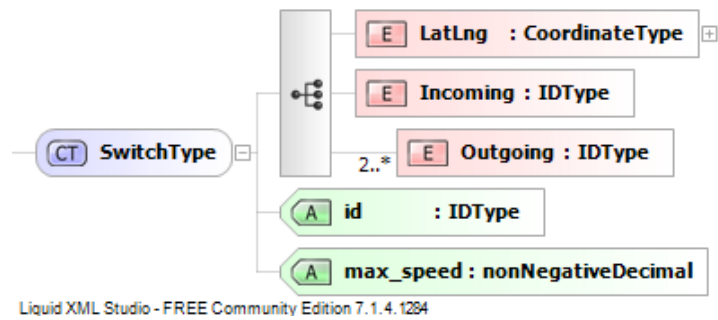


Figure A.13: SwitchType

A.12) and SwitchType (Figure A.13) elements are similar to the WaypointType.



Figure A.14: MetadataType

A MetadataType element only provides a version string at this time.

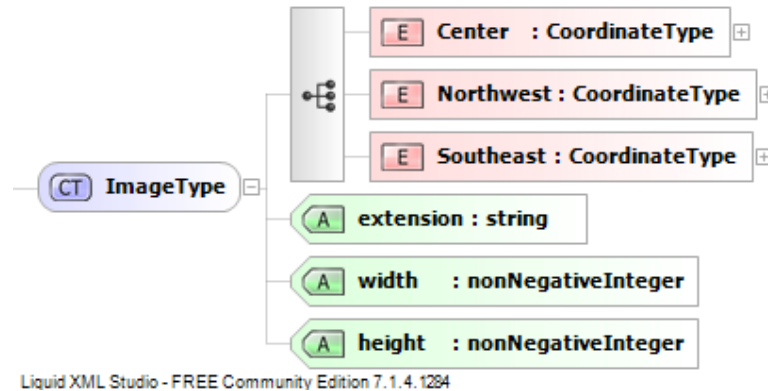


Figure A.15: ImageType

An ImageType (Figure A.15) element provides georeferencing metadata for an image file. The NorthWest, SouthEast, and Center elements only contain the required “lat” and “lng” attributes. The “extension” attribute specifies the file type, e.g. “png”. The “width” and “height” attributes give the image dimensions in pixels.

An WeatherType (Figure A.16) element’s “temperature” attribute is in degrees Celcius. The “elevation” attribute is the average elevation for the track network, measured in meters above sea level; “pressure” is absolute pressure in Pascals; “air\_density” is in  $kg/m^3$ , “wind\_speed” in  $m/s$ ; “wind\_direction” is a value between 0 and 359, where

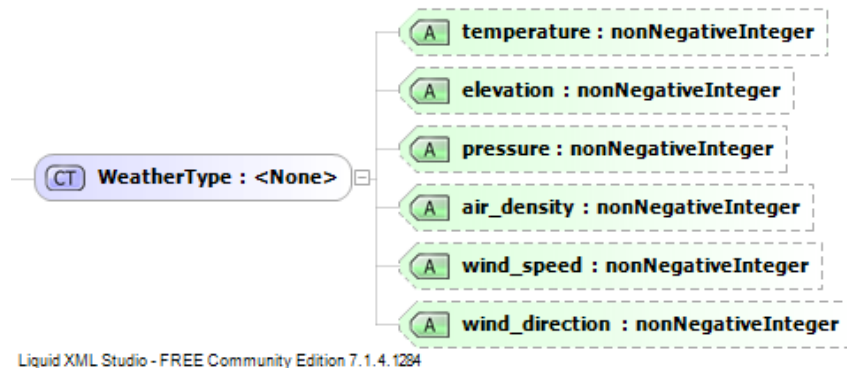


Figure A.16: WeatherType

0 is a North wind, 90 is a East wind, etc.



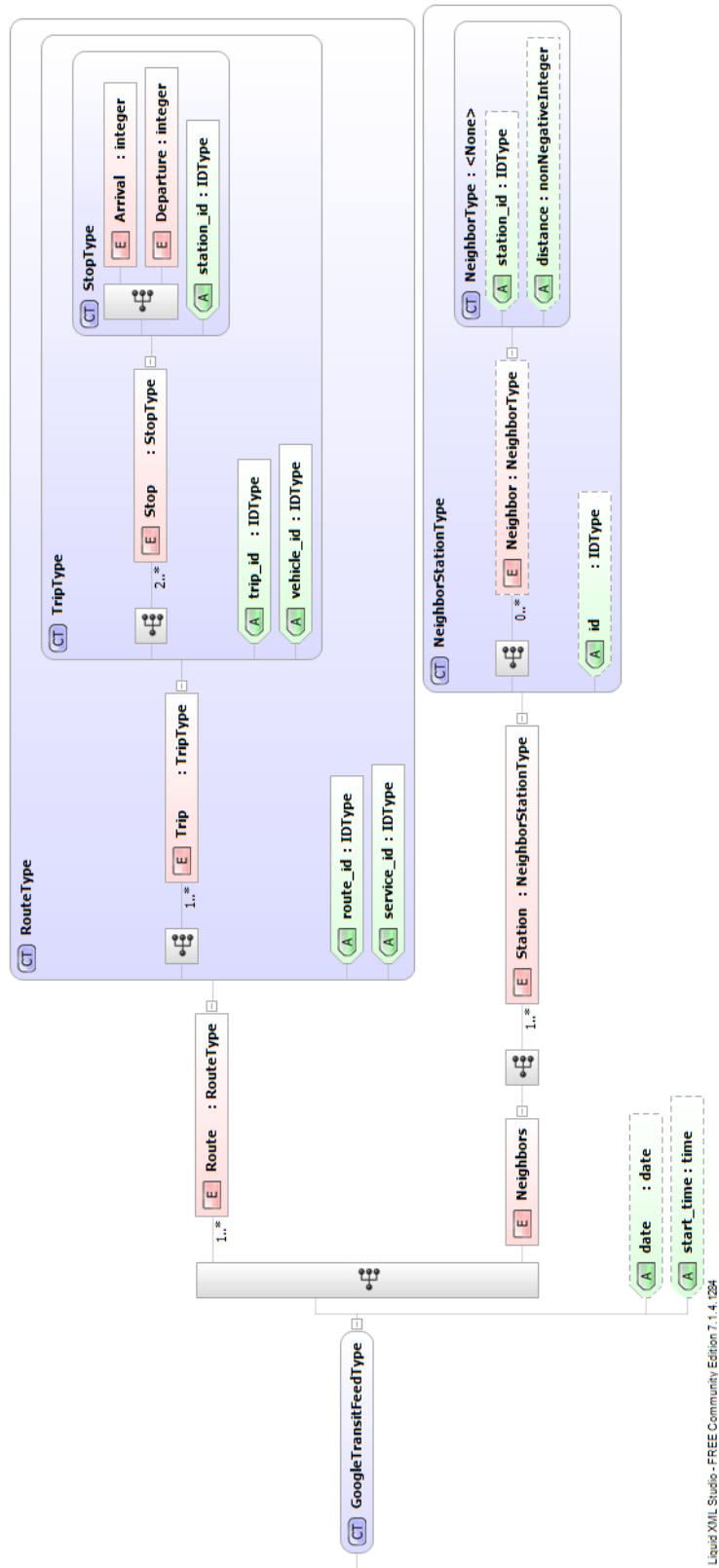


Figure A.17: GoogleTransitFeedType

A `GoogleTransitFeedType` (Figure A.17) contains a sequence of `Route` elements. Each `Route` contains `Trips`, and `Trips` contain `Stops`. Each `Stop` specifies an Arrival time and Departure time, measured as seconds after to the simulation's start time, and a "station\_id" attribute. A `GoogleTransitFeedType` also contains a `Neighbors` element, which supplies inter-station distances for a selection of stations (those within a distance threshold of each other). The distances supplied are "as the crow flies" distances.

## Appendix B

# Communication

Messages are passed between the Sim and controller(s) using TCP/IP. To establish the connection(s), the Simulator binds to a port<sup>1</sup> and waits for the pre-determined number of controllers to connect. Once connected, the Sim sends each controller a SIM\_GREETING message containing the contents of the scenario file. Once the user starts the simulation, the Sim sends each controller a SIM\_START message. After the simulation has completed, the simulator sends each controller a SIM\_END message and closes the connections.

During the simulation, the Simulator pauses after every event and sends one or more messages describing the event to the controllers. The simulation remains paused until all controllers have replied with a CTRL\_RESUME message containing the sim's latest *Msg ID* in the body (see below). Once the simulation resumes, controllers have no way of requesting that it pause again—they must wait for the next event to occur.

---

<sup>1</sup>Port 64444 by default.

Prior to sending a CTRL\_RESUME, controllers may request that an event occur when a particular time is reached, or when a vehicle reaches a position.

The reason for requiring that controllers respond with the *latest Msg ID* prior to resuming is that controllers have no way of determining whether additional messages will be arriving. Not resuming the simulation until controllers have responded to the latest message sent by the Sim allows a controller to be designed to send a CTRL\_RESUME as a final acknowledgement to *every* message, yet still have an opportunity to “change its mind” if/when a new message arrives.

## B.1 Messages

Communications between the Simulator and controller(s) are encoded using Protocol Buffers<sup>2</sup> (PB), a message-based mechanism for serializing structured data. Protocol Buffers uses a compact, binary wire-format and places an emphasis on encoding and decoding performance. Libraries for using PB messages are available for most commonly used languages<sup>3</sup>.

PB messages are not self-describing, and require an additional mechanism to specify the message type and length. To accomplish this, all messages passed between the Simulator and Controller(s) are prefixed with a fixed-length header (see Table B.1).

The header is transmitted in big endian byte order, and all fields are signed values. The *Separator* field has a fixed value of -32123<sup>4</sup>. The *Msg Type* field must

---

<sup>2</sup><http://code.google.com/apis/protocolbuffers/>

<sup>3</sup>See <http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns> for a list of available libraries

<sup>4</sup>0x82 0x85 in the wire-format

Purpose	Bytes
Separator	2
Msg Type	2
Msg ID	4
Sim Time	4
Msg Length	4

Table B.1: 16-byte Message Header

contain one of the enumeration values found in Table B.2 if the message is sent from a controller, or in Table B.3 if sent from the Sim. The *Msg ID* field is a counter, incremented with each message sent; the sim and the controller(s) may each use their own counter. *Sim Time* is the simulation time at which the message will arrive at its destination, including the effects of any simulated latency<sup>5</sup>. *Sim Time* is measured as milliseconds since the simulation start. The *Msg Length* field contains the length of the message body, in bytes.

For descriptions of the message bodies, see the `api.proto` file that defines them.

The latest version may be found here:

<http://code.google.com/p/prt-sim/source/browse/trunk/pyprt/shared/api.proto>

---

<sup>5</sup>Simulated latency has not been implemented at this time; Sim Time values are equal to the simulation time at which the message is sent.

Enum	Msg Type
1	CTRL_CMD_VEHICLE_TRAJECTORY
2	CTRL_CMD_VEHICLE_ITINERARY
3	CTRL_CMD_SWITCH
4	CTRL_CMD_PASSENGERS_EMBARK
5	CTRL_CMD_PASSENGERS_DISEMBARK
6	CTRL_CMD_PASSENGER_WALK
7	CTRL_CMD_STORAGE_ENTER
8	CTRL_CMD_STORAGE_EXIT
10	CTRL_REQUEST_VEHICLE_STATUS
11	CTRL_REQUEST_STATION_STATUS
12	CTRL_REQUEST_PASSENGER_STATUS
13	CTRL_REQUEST_SWITCH_STATUS
14	CTRL_REQUEST_TRACKSEGMENT_STATUS
15	CTRL_REQUEST_TOTAL_STATUS
20	CTRL_SETNOTIFY_VEHICLE_POSITION
21	CTRL_SETNOTIFY_TIME
30	CTRL_RESUME
50	CTRL_SCENARIO_ERROR

Table B.2: Controller Message Types. Messages originating from a controller include an enum value from this table in the *Msg Type* field of the message header.

Enum	Msg Type
1000	SIM_GREETING
1001	SIM_START
1002	SIM_END
1003	SIM_UNIMPLEMENTED
1010	SIM_COMPLETE_PASSENGERS_EMBARK
1011	SIM_COMPLETE_PASSENGERS_DISEMBARK
1012	SIM_COMPLETE_PASSENGER_WALK
1013	SIM_COMPLETE_SWITCH
1014	SIM_COMPLETE_STORAGE_ENTER
1015	SIM_COMPLETE_STORAGE_EXIT
1030	SIM_RESPONSE_VEHICLE_STATUS
1031	SIM_RESPONSE_STATION_STATUS
1032	SIM_RESPONSE_PASSENGER_STATUS
1033	SIM_RESPONSE_SWITCH_STATUS
1034	SIM_RESPONSE_TRACK_STATUS
1035	SIM_RESPONSE_TOTAL_STATUS
1040	SIM_NOTIFY_VEHICLE_POSITION
1041	SIM_NOTIFY_VEHICLE_ARRIVE
1042	SIM_NOTIFY_VEHICLE_EXIT
1043	SIM_NOTIFY_VEHICLE_STOPPED
1044	SIM_NOTIFY_VEHICLE_SPEEDING
1045	SIM_NOTIFY_VEHICLE_COLLISION
1046	SIM_NOTIFY_VEHICLE_CRASH
1047	SIM_NOTIFY_PASSENGER_EMBARK_START
1048	SIM_NOTIFY_PASSENGER_EMBARK_END
1049	SIM_NOTIFY_PASSENGER_DISEMBARK_START
1050	SIM_NOTIFY_PASSENGER_DISEMBARK_END
1051	SIM_NOTIFY_TIME

Table B.3: Simulator Message Types – Part 1 of 2. Messages originating from the Simulator include an enum value from this table in the *Msg Type* field of the message header.

Enum	Msg Type
1060	SIM_EVENT_TRACK_DISABLED
1061	SIM_EVENT_TRACK_REENABLED
1062	SIM_EVENT_SWITCH_DISABLED
1063	SIM_EVENT_SWITCH_REENABLED
1064	SIM_EVENT_STATION_DISABLED
1065	SIM_EVENT_STATION_REENABLED
1066	SIM_EVENT_VEHICLE_DISABLED
1067	SIM_EVENT_VEHICLE_REENABLED
1068	SIM_EVENT_PASSENGER_CREATED
1069	SIM_EVENT_PASSENGER_CHANGEDEST
1080	SIM_MSG_HDR_INVALID_SEPARATOR
1081	SIM_MSG_HDR_INVALID_TYPE
1082	SIM_MSG_HDR_INVALID_ID
1083	SIM_MSG_HDR_INVALID_TIME
1084	SIM_MSG_HDR_INVALID_SIZE
1085	SIM_MSG_BODY_INVALID
1086	SIM_MSG_BODY_INVALID_TIME
1087	SIM_MSG_BODY_INVALID_ID

Table B.4: Simulator Message Types – Part 2 of 2. Messages originating from the Simulator include an enum value from this table in the *Msg Type* field of the message header.



## Appendix C

# Trajectory Calculations

Mathematica:

General System of Equations for targeting time when acceleration limits are reached (7 segment spline):

```
t01 = (a1 - a0)/j01
v01 = j01*t01^2/2 + a0*t01
v1 = v0 + v01
q01 = j01*t01^3/6 + a0*t01^2/2 + v0*t01
t23 = (a3 - a2)/j23
v23 = j23*t23^2/2 + a2*t23
v2 = v3 - v23
q23 = j23*t23^3/6 + a2*t23^2/2 + v2*t23
v12 = v2 - v1
t12 = v12/a1
q12 = a1*t12^2/2 + v1*t12
q03 = q01 + q12 + q23
t03 = t01 + t12 + t23
t67 = (a7 - a6)/j67
v67 = j67*t67^2/2 + a6*t67
v6 = v7 - v67
q67 = j67*t67^3/6 + a6*t67^2/2 + v6*t67
t45 = (a5 - a4)/j45
v45 = j45*t45^2/2 + a4*t45
v4 = v3
v5 = v4 + v45
```

```

q45 = j45*t45^3/6 + a4*t45^2/2 + v4*t45
v56 = v6 - v5
t56 = v56/a5
q56 = a5*t56^2/2 + v5*t56
q47 = q45 + q56 + q67
t47 = t45 + t56 + t67
q34 = q7 - q0 - q03 - q47
t34 = t7 - t0 - t03 - t47
FullSimplify[Solve[v3 == q34/t34, v3]]

```

```

v3 -> (3 (a0 - a1)^2 a5 j01 j23^2 j45^2 j67^2 +
  3 j01^2 j23^2 j45^2 j67 (-2 a5 j67 v0 +
    a1 (a6^2 - a7^2 + 2 a5 (-a6 + a7 + j67 t0 - j67 t7) +
      2 j67 v7)) +
  Sqrt[3] \[Sqrt](j01^2 j23^2 j45^2 j67^2 (3 j23^2 j45^2 (a1^2 a5 \
j67 + a5 j67 (a0^2 - 2 j01 v0) +
  a1 (-2 a5 (a0 j67 + j01 (a6 - a7 - j67 t0 + j67 t7)) +
    j01 (a6^2 - a7^2 + 2 j67 v7)))^2 - (a1 -
  a5) (a1^4 a5 j23^2 j45^2 j67^2 -
  6 a1^2 a5 j23^2 j45^2 j67^2 (a0^2 - 2 j01 v0) +
  3 a5 j45^2 j67^2 (a2^4 j01^2 -
    j23^2 (a0^2 - 2 j01 v0)^2) +
  a1 (-6 a4^2 a5^2 j01^2 j23^2 j67^2 +
    a5^4 j01^2 j23^2 j67^2 -
    4 a5 (a6^3 j01^2 j23^2 j45^2 +
      2 a7^3 j01^2 j23^2 j45^2 +
      j67^2 (-2 a4^3 j01^2 j23^2 +
        j45^2 (a2^3 j01^2 -
          2 j23^2 (a0^3 + 3 j01^2 (q0 - q7) -
            3 a0 j01 v0))) - 6 a7 j01^2 j23^2 j45^2 j67 v7 -
          3 a6 j01^2 j23^2 j45^2 (a7^2 - 2 j67 v7)) +
          3 j01^2 j23^2 (-a4^2 j67 +
            j45 (a6^2 - a7^2 + 2 j67 v7)) (a4^2 j67 +
            j45 (a6^2 - a7^2 + 2 j67 v7)))))))/(6 (a1 -
a5) j01^2 j23^2 j45^2 j67^2)

```

```

v3 -> (3 (a0 - a1)^2 a5 j01 j23^2 j45^2 j67^2 +
  3 j01^2 j23^2 j45^2 j67 (-2 a5 j67 v0 +
    a1 (a6^2 - a7^2 + 2 a5 (-a6 + a7 + j67 t0 - j67 t7) +
      2 j67 v7)) -
  Sqrt[3] \[Sqrt](j01^2 j23^2 j45^2 j67^2 (3 j23^2 j45^2 (a1^2 a5 \
j67 + a5 j67 (a0^2 - 2 j01 v0) +
  a1 (-2 a5 (a0 j67 + j01 (a6 - a7 - j67 t0 + j67 t7)) +

```

$$\begin{aligned}
& j_{01} (a_6^2 - a_7^2 + 2 j_{67} v_7))^2 - (a_1 - \\
& a_5) (a_1^4 a_5 j_{23}^2 j_{45}^2 j_{67}^2 - \\
& 6 a_1^2 a_5 j_{23}^2 j_{45}^2 j_{67}^2 (a_0^2 - 2 j_{01} v_0) + \\
& 3 a_5 j_{45}^2 j_{67}^2 (a_2^4 j_{01}^2 - \\
& j_{23}^2 (a_0^2 - 2 j_{01} v_0)^2) + \\
& a_1 (-6 a_4^2 a_5^2 j_{01}^2 j_{23}^2 j_{67}^2 + \\
& a_5^4 j_{01}^2 j_{23}^2 j_{67}^2 - \\
& 4 a_5 (a_6^3 j_{01}^2 j_{23}^2 j_{45}^2 + \\
& 2 a_7^3 j_{01}^2 j_{23}^2 j_{45}^2 + \\
& j_{67}^2 (-2 a_4^3 j_{01}^2 j_{23}^2 + \\
& j_{45}^2 (a_2^3 j_{01}^2 - \\
& 2 j_{23}^2 (a_0^3 + 3 j_{01}^2 (q_0 - q_7) - \\
& 3 a_0 j_{01} v_0))) - 6 a_7 j_{01}^2 j_{23}^2 j_{45}^2 j_{67} v_7 - \\
& 3 a_6 j_{01}^2 j_{23}^2 j_{45}^2 (a_7^2 - 2 j_{67} v_7)) + \\
& 3 j_{01}^2 j_{23}^2 (-a_4^2 j_{67} + \\
& j_{45} (a_6^2 - a_7^2 + 2 j_{67} v_7)) (a_4^2 j_{67} + \\
& j_{45} (a_6^2 - a_7^2 + 2 j_{67} v_7)))))))/(6 (a_1 - \\
& a_5) j_{01}^2 j_{23}^2 j_{45}^2 j_{67}^2)
\end{aligned}$$

Python: (Unoptimized)

```

numerator_a = 3*(a[0]-a[1])**2*a[5]*j[0]*j_sq[2]*j_sq[4]*j_sq[6]+
3*j_sq[0]*j_sq[2]*j_sq[4]*j[6]*(-2*a[5]*j[6]*v[0]+a[1]*(a_sq[6]-
a_sq[7]+2*a[5]*(-a[6]+a[7]+j[6]*t[0]-j[6]*t[7])+2*j[6]*v[7]))

```

```

numerator_b = math.sqrt(3)*math.sqrt(j_sq[0]*j_sq[2]*j_sq[4]*
j_sq[6]*(3*j_sq[2]*j_sq[4]*(a_sq[1]*a[5]*j[6]+a[5]*j[6]*
(a_sq[0]-2*j[0]*v[0])+a[1]*(-2*a[5]*(a[0]*j[6]+j[0]*
(a[6]-a[7]-j[6]*t[0]+j[6]*t[7]))+j[0]*(a_sq[6]-a_sq[7]+2*j[6]*v[7]))))**2-
(a[1]-a[5])*(a_sq[1]*a_sq[1]*a[5]*j_sq[2]*j_sq[4]*j_sq[6]-6*a_sq[1]*
a[5]*j_sq[2]*j_sq[4]*j_sq[6]*(a_sq[0]-2*j[0]*v[0])+3*a[5]*j_sq[4]*
j_sq[6]*(a_sq[2]*a_sq[2]*j_sq[0]-j_sq[2]*(a_sq[0]-2*j[0]*v[0])**2)+
a[1]*(-6*a_sq[4]*a_sq[5]*j_sq[0]*j_sq[2]*j_sq[6]+a_sq[5]*a_sq[5]*
j_sq[0]*j_sq[2]*j_sq[6]-4*a[5]*(a_sq[6]*a[6]*j_sq[0]*j_sq[2]*j_sq[4]+
2*a_sq[7]*a[7]*j_sq[0]*j_sq[2]*j_sq[4]+j_sq[6]*(-2*a_sq[4]*a[4]*
j_sq[0]*j_sq[2]+j_sq[4]*(a_sq[2]*a[2]*j_sq[0]-2*j_sq[2]*
(a_sq[0]*a[0]+3*j_sq[0]*(q[0]-q[7])-3*a[0]*j[0]*v[0])))-
6*a[7]*j_sq[0]*j_sq[2]*j_sq[4]*j[6]*v[7]-3*a[6]*j_sq[0]*
j_sq[2]*j_sq[4]*(a_sq[7]-2*j[6]*v[7]))+3*j_sq[0]*j_sq[2]*
(-a_sq[4]*j[6]+j[4]*(a_sq[6]-a_sq[7]+2*j[6]*v[7]))*
(a_sq[4]*j[6]+j[4]*(a_sq[6]-a_sq[7]+2*j[6]*v[7]))))

```

```

denominator = (6*(a[1]-a[5])*j_sq[0]*j_sq[2]*j_sq[4]*j_sq[6])

```

```
v3_soln_1 = (numerator_a + numerator_b) / denominator  
v3_soln_2 = (numerator_a - numerator_b) / denominator
```

# Bibliography

- [1] J. Edward Anderson. *Transit Systems Theory*. D.C. Heath and Company, 1978.
- [2] JE Anderson. Cabintaxi- Urban transport of the future. In *Society of Allied Weight Engineers, Annual Conference, 35 th, Philadelphia, Pa; United States*, page 29, 1976.
- [3] J.E. Anderson. Some History of PRT Simulation Programs. [http://www.prtzn.com/component/option,com\\_docman/task,doc\\_download/gid,24/Itemid,37/](http://www.prtzn.com/component/option,com_docman/task,doc_download/gid,24/Itemid,37/), 2007.
- [4] Beamways. BeamEd. <http://www.beamways.com>.
- [5] R. Gilbert and A. Perl. Grid-connected vehicles as the core of future land-based transport systems. *Energy Policy*, 35(5):3053–3060, 2007.
- [6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

- [7] Jack H. Irving, Harry Bernstein, C. L. Olson, and Jon Buyan. *Fundamentals of Personal Rapid Transit*. Lexington Books, 1978.
- [8] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.
- [9] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *Software Engineering, IEEE Transactions on*, SE-9(5):631 – 634, Sep 1983.
- [10] Klaus Müller and Tony Vignaux. SimPy: Simulating Systems in Python. *ON-Lamp.com Python DevCenter*, February 2003.
- [11] Joerg Schweizer. innovative Transport Simulator (iTTS). <http://www.trasporti.ing.unibo.it/personale/schweizer/mait/projects/index.html#its>, 2007.
- [12] J.R. Stout. JPRT: A Distributed PRT Simulation Program. Master’s thesis, West Virginia University, 2004.
- [13] Markus Szillat. *A Low-level PRT Microsimulation*. PhD thesis, University of Bristol, Apr 2001.
- [14] Christos Xithalis. Hermes PRT Network Simulator. [http://students.ceid.upatras.gr/~xithalis/simulation\\_en.html](http://students.ceid.upatras.gr/~xithalis/simulation_en.html), 2006.