Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Concepts for Advanced Integration of SuperTuxKart into Connected Cars

Robin Dominic Pesl

Course of Study:	Informatik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. Ilche Georgievski Dr. rer. nat. Uwe Breitenbücher
Commenced:	June 22, 2021
Completed:	December 22, 2021

Abstract

Connected cars enhance cars with high-performance computational devices and the ability to connect with the cloud as well as with other connected cars. Usage of these computational devices enables video games within connected cars. But connected cars also offer additional abilities, which can be used for advanced integration of video games into connected cars to exploit the additional functionality to enhance the gaming experience. As basis for this advanced integration a data exchange model is introduced, which covers the injection of data from the real-life into the video game and the extraction of data from the in-game world to be used in the real-life. To demonstrate the potential of this vision for advanced integration of video games into connected cars, the kart racing game SuperTuxKart is used as foundation for two prototypes. The first implemented prototype is the World Generation. The World Generation uses OpenStreetMap data and satellite images to generate a SuperTuxKart track based on the real-life. It takes the car's position and the size of area that should be mapped as input parameter and produces the track with all necessary track constituents as result. The World Generation prototype demonstrates how data injection can be used to connect the game with the real-life and improve realism and therefore immersion in-game. The second implemented prototype is the REST API for SuperTuxKart, which serves as middleware for bidirectional data exchange. The REST API eases interaction with SuperTuxKart as knowledge about the internal structure of SuperTuxKart is not needed to interact with the REST API. The REST API itself does not improve the integration of SuperTuxKart into connected cars but serves as foundation for further projects that extend and amplify the integration of video games and especially SuperTuxKart into connected cars. Both prototypes reveal the potential of this vision and future projects.

Kurzfassung

Vernetzte Autos erweitern Autos um leistungsfähige Computersysteme und die Fähigkeit, sich mit der Cloud und anderen vernetzten Autos zu verbinden. Die Nutzung dieser Computersysteme ermöglichen es Videospiele innerhalb des vernetzten Autos zu spielen. Zusätzlich bieten vernetzte Autos aber noch weitere Möglichkeiten und damit Potential, das genutzt werden kann, um das Spielerlebnisses zu verbessern. Dazu ist allerdings eine tiefgreifende Integration der zusätzlichen Hardware und Möglichkeiten nötig. Aus diesem Grund wird ein Datenaustauschmodell eingeführt, das die Injektion von Daten aus der echten Umgebung des Autos ins Spiel und die Extraktion von Daten aus der Spielwelt zur Verwendung im echten Leben umfasst. Das Kart-Racing-Spiel SuperTuxKart dient als Grundlage für zwei Prototypen, die das Potenzial der Vision für die tiefe Integration von Videospielen in vernetzte Autos zeigen. Der erste implementierte Prototyp ist die Welt-Generierung. Die Welt-Generierung verwendet OpenStreetMap-Daten und Satellitenbilder, um eine SuperTuxKart-Rennstrecke auf Basis der realen Umgebung zu generieren. Eingabe ist die Position des Fahrzeugs und die abzubildende Fläche und Ausgabe ist die Rennstrecke mit allen notwendigen Streckenbestandteilen und Dateien. Der Welt-Generierungs-Prototyp zeigt, wie die Dateninjektion genutzt werden kann, um den Realismus und damit die Immersion im Spiel zu verbessern und das Spiel mit der Realität zu verbinden. Der zweite implementierte Prototyp ist die REST-API für SuperTuxKart, die als Middleware für den bidirektionalen Datenaustausch dient. Die REST-API erleichtert die Interaktion mit SuperTuxKart, da die Interna von SuperTuxKart vor dem Anwender verborgen bleiben. Damit sind keine Vorkenntnisse außerhalb der REST-API nötig, um mit der REST-API und damit SuperTuxKart zu interagieren. Das Ziel der REST-API ist dabei nicht selbst die Integration von SuperTuxKart in vernetzte Autos zu verbessern, sondern als Grundlage weitere Integrationsprojekte von Videospielen und insbesondere von SuperTuxKart in vernetzte Autos zu ermöglichen. Beide Prototypen zeigen das Potenzial der Vision von Videospielen in vernetzten Autos und damit einhergehenden zukünftiger Projekte.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

2D two-dimensional. 22

3D three-dimensional. 19

AI artificial intelligence. 19

API application programming interface. 30

GIS geographic information system. 39

HTTP Hypertext Transfer Protocol. 30

LOD level of detail. 40

OSM OpenStreetMap. 39

REST Representational State Transfer. 30

SPM Space Partitioned Mesh. 19

1 Introduction

Video games are popular within large parts of the population and are played in various situations and on various devices. Usual cars are not equipped to allow playing video games within the car. With the real-world availability of connected cars, usage scenarios for their additional functionality are becoming more and more relevant and possible to realize. Therefore, it stands reason to also play video games in connected cars and strongly integrate those games into these connected cars.

1.1 Context

The purpose of a usual car is to transport passengers from a source to a destination. In order to increase comfort while driving, many technical support systems have been added like driver assistance systems or entertainment systems for passengers's amusement and satisfaction. Connected cars with their additional sensors and ability to connect to the cloud and other connected cars go beyond the limitations of regular cars. These additional possibilities allow advanced driving assistant systems as well as advanced entertainment systems.

Video games are a well-known entertainment possibility, usually played on stationary devices, e.g., computers, video game consoles, or mobile devices. With the availability of all hardware requirements needed to run video games built into connected cars, it is possible to play video games inside a connected car. Either while taking a break from driving or with full autonomous driving even while being en route. But connected cars are not limited to local computing hardware. Therefore, it is possible to create new or extend existing video games that make use of the additional sensor data and accessibility of the connected system of cars.

Car racing in sports has a long tradition. In racing, cars drive on a racetrack built into a specific landscape trying to be as fast as possible and score the lowest lap time or reach the finish line before their opponents. But car racing is not limited to the real-life. The concept of car racing is mapped to video games as racing games. Based on the closely related concepts of cars, racing, and racing games, it is possible to generate high immersion by integrating racing games into connected cars.

SuperTuxKart as kart racing game that allows players to drive through virtual worlds using fictional karts against computer-controlled opponents or against other players on the network. Because of the close relation of racing and racing games it is possible to use SuperTuxKart as a representative example and basis for advantageous integration of video games into connected cars.

1.2 Objectives

To demonstrate the advantages of strong integration of video games into connected cars the goal of this work is to extend the functionality of SuperTuxKart to integrate into the context of connected cars. Therefore, SuperTuxKart is extended by a mechanism to increase realism within the game by generating an in-game world based on the real-world surrounding and an abstraction layer easing maintainability by hiding the internal structure of the game and allowing the easy usage of SuperTuxKart for future projects in the context of connected cars.

1.2.1 World-Generation

To increase realism within SuperTuxKart the idea is to generate an in-game world based on real-world data. This replication of the real-life can then be used to stage races within the real-world surrounding of the player. The result is a stand-alone service taking the user's position and producing an archive containing the racetrack and anything required to load the track with SuperTuxKart. The service integrates existing data sources and transforms their information into a format that is understandable by SuperTuxKart. It is thereby necessary to generate an in-game world that is as realistic as possible but still integrates into the SuperTuxKart universe and that follows the SuperTuxKart gameplay.

1.2.2 Maintainability

As SuperTuxKart is a rather complex game and knowledge of the internal structure and interfaces requires time and effort to be understood, an abstraction layer eases future projects using SuperTuxKart as foundation. To further ease usage and reduce time needed to learn how to use this abstraction layer it is necessary to use well-known interaction standards. The goal is therefore to create a REST API that offers access to the game's internal state. Using this REST API, it is then possible to access and extract as well as manipulate the game's current state. As interaction format the well-known JSON format is used, which further reduces learning effort.

1.3 Structure

The resolution of these objectives is structured as follows:

Chapter 2: Background Information contains the knowledge needed to understand the suggested solutions.

Chapter 3: Vision describes the detailed vision about advanced integration of video games in connected cars and what steps need to be done to achieve it.

Chapter 4: World Generation presents the realization of the World Generation prototype.

Chapter 5: REST API introduces the prototypical solution of the maintainability objective.

2 Background Information

The following sections contain a brief elaboration of the relevant software projects and technologies needed to understand the introduced concepts and their realization. It contains information about SuperTuxKart in section 2.1, the definition of the Space Partitioned Mesh (SPM) file format in section 2.2, building blocks of a SuperTuxKart track in section 2.3, the algorithm for finding cycles in graphs in section 2.4, and an overview about application interfaces and REST in section 2.5.

2.1 SuperTuxKart

SuperTuxKart is an open-source, multi-platform kart racing game using a three-dimensional (3D) comic-style artwork. It takes places in a fictional world using fictional karts and characters. Each character has its own unique kart. The karts differ in design and driving capabilities like acceleration and mass. The goal of the game is to drive races by controlling one of the characters against time limits or opponent characters. While a race is running, the 3D world is rendered showing the kart of the player centered and in third person perspective. If there is no active race, the menu is shown, which allows the player to select options or start a new race.

There are multiple game modes possible. The player can choose between a regular race with or without opponents where the first characters who crosses the finish line wins, a race against time, a race against prerecorded rival karts, or special modes like “Battle” to destroy opponents, “Soccer” to score goals using a soccer ball, or “Egg hunt” to find hidden Easter eggs. For each game mode the player can choose between different tracks, or a consecutive set of tracks called a “Grand Prix”. Each race takes place in its own location and therefore its own 3D model of the world. For the regular game modes, the racetrack follows a cyclic path through the in-game world, and the number of laps can be specified, which needs to be driven until the race ends. The opponents can be controlled by an artificial intelligence (AI) or by other players using the same device by splitting the screen into multiple views, showing the kart of each player in its own view, or over the network [Sup21b].

2.2 Space Partitioned Mesh File Format

SuperTuxKart expects 3D models to be stored in the SPM format. SPM is a binary file format containing the 3D model represented as triangles in the 3D vector space, the model’s textures, and the color of each triangle endpoint. The SPM specification shown here is extracted from the SuperTuxKart source code as there is no formal specification and no publicly available documentation. The used source code is the SPM writer used to export 3D models out of Blender [SPM21] and the parser used to load the models into SuperTuxKart [Sup21a].

Offset	Bytes	Type	Value	Description
0	1	character	'S'	
1	1	character	'P'	
2	1	binary	0x0a	Static mesh
3	1	binary	0b00000 normal color tangent	
4	4	float32	ignored	x_{min}
8	4	float32	ignored	y_{min}
12	4	float32	ignored	z_{min}
16	4	float32	ignored	x_{max}
20	4	float32	ignored	y_{max}
24	4	float32	ignored	z_{max}
28	2	uint16_t	no restriction	Number of textures n
30	s_n	texture	texture	Textures
$30 + s_n$	2	uint16_t	no restriction	Number of sectors
$32 + s_n$	2	uint16_t	no restriction	Number of used textures in sector
$34 + s_n$	4	uint32_t	no restriction	Number of vertices m
$38 + s_n$	4	uint32_t	no restriction	Number of indices i
$40 + s_n$	2	uint16_t	consecutively numbered	Material identifier
$42 + s_n$	s_m	vertex	vertex	Vertices
$42 + s_n + s_m$	$i \cdot s_i$	uints_i_t	index triple	Indices

Table 2.1: Space Partitioned Mesh memory layout

Table 2.1 shows the general memory layout of the SPM format. As it is a binary format the constituents are shown bytewise. The byte order is Little Endian. It starts with the two characters "SP" followed by one byte indicating that a static model follows. It is also possible to load animated models by using the flag 0x0a instead of 0x09 and add the data needed to animate the model to each vertex and after the vertex data section. As the focus is on static models, the structure of the animation data is omitted here. The flags "normal", "color" and "tangent" indicate whether each vertex contains normal, color or tangent information. The following six floating point numbers represent the three-dimensional bounding-box of the model but remain unused in SuperTuxKart.

Given the number of textures n, SuperTuxKart tries to read n textures following the structure described in table 2.2. A texture consists of the two filenames of the two images that should be used for this texture. The first image is displayed on the model whereas the second image can only be used in shaders to create specific effects. The second or both of these filenames may be empty. If a filename is empty, SuperTuxKart creates an empty texture. In all other cases if the filename does not correspond to a valid file, i.e., the file does not exist or cannot be loaded by SuperTuxKart, the loading of the SPM file fails. The files are looked up in the assets directory of

Offset	Bytes	Type	Value	Description
0	1	uint8_t	no restriction	Length f_{first} of filename of the first texture in bytes
1	f_{first}	string	characters	The f_{first} characters of the second filename
1 + f_{first}	1	uint8_t	no restriction	Length f_{second} of filename of the second texture in bytes
2 + f_{first}	f_{second}	string	characters	The f_{second} characters of the second filename

Table 2.2: Space Partitioned Mesh texture memory layout

the SuperTuxKart installation or in the directory where the SPM file is located. If the same first filename occurs multiple times, the SPM file cannot be loaded correctly. All textures in the SPM result in an accumulated size of $s_n = 2 \cdot n + \sum_{i=1}^{n}(f_{first_i} + f_{second_i})$ bytes for all n textures.

Bytes	Type	Flag	Condition	Description
12	3 · float32	“normal”		Normal vector of vertex
1	uint8_t	“color”		If set to 1, color is set to white
3	3 · uint8_t	“color”	Previous not equal to 1	Color given as red, green and blue
4	2 · float16		First filename of material is not empty	u and v coordinates of first material
4	2 · float16		First and second filename of material are not empty	u and v coordinates of second material
4	uint32_t	“tangent”	First filename of material is not empty	Tangent of vertex

Table 2.3: Space Partitioned Mesh vertex memory layout

For each sector and material SuperTuxKart tries to load the m triangle vertices whereas each vertex is defined by the structure described in table 2.3. SuperTuxKart supports at most 65535 vertices per material else the SPM file cannot be loaded. The rows of table 2.3 are evaluated in consecutive order. The data for each row is only loaded if the flag specified in column “Flag” is set and the condition in column “Condition” is satisfied. Else the row is skipped and the condition for the next row is evaluated. The u and v coordinates are in the range $[0, 1]$ describing the normalized horizontal and vertical position on the image used as texture. Each texturized triangle has for each vertex its own u and v coordinates resulting in a distorted projection of the image to the triangle. Figure 2.1 illustrates this mapping using u and v coordinates to transform a rectangular image to a texturized triangle.

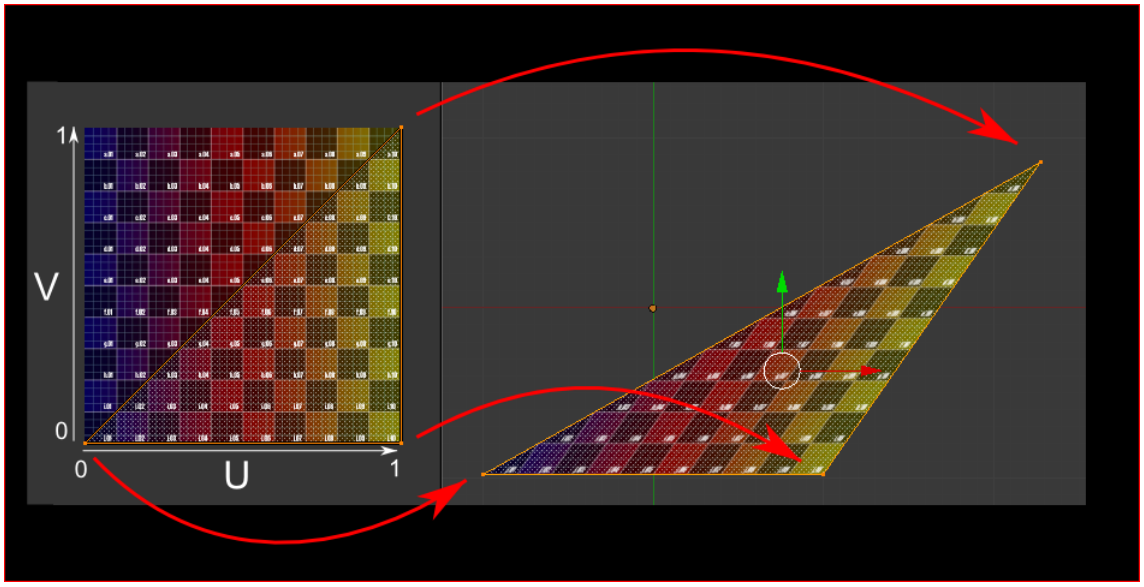

Figure 2.1: Mapping of textures to triangles using *uv* coordinates [ope13]

To define the triangles based on the set of vertices the list of indices follows the vertex data. Every three indices span a triangle of the three vertices corresponding to the three given indices in the set of vertices. Therefore, the number of indices must be divisible by three and each index must have a corresponding vertex in the list of vertices. If the number of vertices is at most 255, each index is stored using one byte else two bytes are used.

2.3 Track Constituents

A SuperTuxKart track consists of distinct components stored in multiple files in a single track directory. Given such a track directory, SuperTuxKart can load it at startup or during runtime. The loaded track is then shown in the list of available tracks, and the user can select it to start a race on this track. The racetrack path is a circuit through the 3D landscape on which the karts can drive without interruptions.

The racetrack circuit is thereby implemented in two parts. The first part is the representation as graph used for the AI to navigate through the track and show a minimap of the current track for the user. This graph consists of quads. Each quad represents a two-dimensional (2D) edge of the graph. Adjacent quads must not intersect. These quads represent the width of the road, and the AI tries to drive within its borders. The second part of the racetrack circuit is the lap counting functionality. Therefore, there are so called “checklines” placed along the circuit. A “checkline” is a line segment on the horizontal plane. These line segments must then be driven through in consecutive order. It can be configured if the kart must be in a specific height to cross the “checkline” or if it is enough that the 2D projection to the horizontal plane crosses the line segment. If a segment is omitted by the driver, the next segment cannot be crossed and therefore the lap not finished [Sup16].

Table 2.4: List of SuperTuxKart track constituents

File	Description
*.spm	The 3D models of the world and objects are stored in the SPM format with extension “spm”. If the model is invalid, SuperTuxKart terminates. If a texture cannot be found in the SuperTuxKart installation or the current track directory, the track cannot be loaded. If an empty texture is used, the model appears in gray.
.jpg, *.png	The track directory can contain additional image resources. All textures, which are not part of the default SuperTuxKart installation, and an optional preview icon of the track are placed into the track directory.
scene.xml	The “scene.xml” file is the main configuration file for the track. It contains references to all used 3D models of the world, the placed objects and their placement information, the checklines, and the bonus items.
quads.xml	The “quads.xml” file contains a list of quads. Each quad is a set of four coordinates. These quads are then used to build the routing graph.
graph.xml	The “graph.xml” file contains the mapping of quads to a routing graph. This routing graph is then used as minimap and for the steerage of the AI karts.
track.xml	The “track.xml” file contains static meta information about the track like name, version, membership in the standard or add-on group, author, music track name, preview icon filename, default number of laps, and driving direction.
materials.xml	The “materials.xml” file contains the information about which shader and shader configuration shall be used for a texture.
scripting.as	Tracks can be extended by scripted functionality. This scripting is located in the “scripting.as” file in the AngelScript programming language.
easter-eggs.xml	The “easter-eggs.xml” file is an optional file with information about the existence and positioning of Easter eggs.

The possible files and therefore track constituents in a track directory and their description are shown in table 2.4. Additional files are ignored while loading.

2.4 Circuit search

Finding a long circuit or even a circuit with a specific length for a given graph is not simple as it is closely related to the NP-complete Hamiltonian cycle problem [Die17, p. 307]. There are some heuristic approaches, which try to approximate the exact solution and find a long cycle in polynomial time in order to overcome the exponential worst-case runtime of computing the exact solution, e.g., [CSF20] or [CBHG17]. Therefore, the definition of a graph is recapitulated, and then the fundamental algorithms for finding a fixed-length circuit in a graph are elaborated.

2.4.1 Graph

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each edge $e \in E$ connects two vertices $v_1, v_2 \in V$. If each edge has a direction, i.e., it connects an initial vertex with a terminal vertex but does not connect the terminal vertex with the initial vertex, the graph is called directed. A path is a graph with distinct vertices $V = \{x_1, x_2, ..., x_k\}$ and edges $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$. A cycle is a path where the first vertex x_1 equals the last vertex x_k. Two paths are independent if they do not share any vertex of each other except their first or final vertex [Die17, pp. 2–8].

2.4.2 Hamiltonian Cycle

A Hamiltonian cycle H of a graph G is a cycle that contains each vertex of G exactly once. Therefore, the upper limit for the length of path without duplicated vertices is the length of a Hamiltonian cycle, which is equal to the number of nodes in the graph. Under the assumption $NP \neq P$ there is no polytime algorithm to determine H for all kinds of graphs [Die17, p. 307]. As a Hamiltonian cycle is the longest vertex distinct cycle in a graph and it is not possible to determine it in polynomial time finding the longest cycle in a graph is also not possible in polytime as it could be used to find Hamiltonian cycles.

2.4.3 Dijkstra

Given a directed graph $G = (V, E)$ with a weight for each edge defined by a cost function $cost : E \to \mathbb{R}_{\geq 0}$ Dijkstra's algorithm finds the shortest path by edge weight if existent from a starting vertex $s \in V$ to a target vertex $t \in V$ in polytime [Dij59].

Algorithm 2.1 describes Dijkstra's algorithm in pseudo code. The input is a weighted graph $G = (V, E)$, a starting vertex $s \in V$, a target vertex $t \in V$, and a cost function $cost : E \to \mathbb{R}_{\geq 0}$. The result is the list D of shortest distance from s to each vertex $v \in V$ and the list P of the previous vertex for each vertex $v \in V$ for the shortest paths starting in s. For all vertices part of the shortest $s - t$ path the values of D are provably their global minimum and it is guaranteed that P contains the previous vertices of all vertices of the $s - t$ path. The shortest $s - t$ path can then be determined using backtracking by starting at a vertex t and repeatably using the previous vertex as current vertex as long as the current vertex is not s. The list of visited nodes is the shortest $s - t$ path in reverse order. If no path $s - t$ exists, the previous vertex of t is undefined.

At the beginning all elements of D except of $D[s]$ are set to ∞ and all elements of P are set to undefined. The distance $D[s]$ of s is set to zero. The set of yet unvisited vertices Q is initialized with s. As long as there are unvisited vertices, the element v of Q with the minimal distance in D is popped out of Q. If $v = t$, the loop can be terminated as the distance of v is minimal in Q and can therefore not decrease anymore as $D[v]$ is the lower limit for any further distance calculation, which can be seen on line 14. This also halts for all neighbors of v as their distance is based recursively on the distance of v. If $v \neq t$ for each of the neighbors of v, an alternative distance u_{alt} is determined by adding the weight of the edge (u, v) to the current distance of v. If u_{alt} is smaller than the current distance of u, the distance of u is set to u_{alt} and the previous vertex of u is set to v. All neighbors of v with an u_{alt} smaller than their current distance are then added to Q as their current distance to s decreased and also their neighbors's distance may decreased in one of the following iterations.

Algorithm 2.1 Dijkstra's algorithm [Dij59]

```
1:  function DIJKSTRA(G = (V, E), s ∈ V, t ∈ V, cost)
2:      D, P ← ∅                                          // D and P are arrays of size |V|
3:      Q ← {s}
4:      ∀v ∈ V \ s : D[v] ← ∞
5:      ∀v ∈ V : P[v] ← undefined
6:      D[s] = 0
7:      while |Q| > 0 do
8:          v ← v ∈ Q : D[v] = min({D[x]|x ∈ Q})    // v is vertex with minimal distance D[v]
9:          Q ← Q \ v                                          // Remove v from Q
10:         if v = t then
11:             break                          // Early exit as D[t] cannot decreased anymore
12:         end if
13:         for all e = (v, u) ∈ E do                          // All neighbors u of v
14:             u_alt ← D[v] + cost(e)
15:             if D[u] > u_alt then
16:                 D[u] ← u_alt
17:                 P[u] ← v
18:                 Q ← Q ∪ u                                  // Add u to Q
19:             end if
20:         end for
21:     end while
22:     return D, P
23: end function
```

After Q is empty or the early exit of line 11 terminated the loop, D contains the shortest distance of all vertices $v \in V$ part of the shortest $s-t$ path or ∞ if not reachable from s and P contains for each vertex $v \in V$ of the shortest $s-t$ path the vertex that is previous on this $s-t$ path or undefined if the vertex is not reachable from s. All other entries of D and P may contain temporary values and cannot be used for shortest distance computations. If there are multiple shortest paths with the same distance, the path that is found first is used as result [Dij59].

Q can be implemented as an array or using an advanced data structure like a min-heap in order to improve performance. The runtime complexity using a min-heap data structure for Q Dijkstra's algorithm has a runtime complexity of $O((|E|+|V|)log|V|)$ [Lia14]. With an almost linear runtime complexity, Dijksta's algorithm is suitable for nearly all kinds of real-world routing problems.

As Dijkstra's algorithm is limited to non-negative edge weights Bhandari suggests a modified version of Dijkstra's algorithm, which can handle negative edge weights. This modification can be done by altering Algorithm 2.1 to allow the same vertex u to be added multiple times to Q. This can happen if u was already processed and removed from Q but because of negative edge weights u is added again to Q with a smaller current distance $D[u]$ than in the previous iteration. The early exit of Algorithm 2.1 in lines 10-12 is no longer possible. The algorithm terminates when Q is empty. Because Q will never be empty if G contains a cycle with negative accumulated edge weight G must not contain such negative cycles to allow the algorithm to terminate [Bha99a]. As

the same vertex may be visited multiple times the runtime of the modified version of Dijkstra's algorithm may increase. An investigation on runtimes and a comparison to alternative algorithms like Bellman-Ford is part of [Lew20].

2.4.4 Yen's Algorithm

Yen's algorithm finds the k shortest $s-t$ paths by their accumulated edge weight beginning at a start vertex $s \in V$ and ending at a target vertex $t \in V$ within a weighted, directed graph $G = (V, E)$. Each of the k shortest paths can be determined in polytime.

Algorithm 2.2 Yen's algorithm [Yen71]

```
1:  function YEN(G = (V, E), s ∈ V, t ∈ V, cost, k)
2:      result ← [dijkstra(G, s, t, cost)]
3:      for i ← 1 to k do
4:          for j ← 0 to |result[i − 1]| − 2 do
5:              V_current ← V
6:              E_current ← E
7:              spurNode ← result[i − 1][j]
8:              rootPath ← result[i − 1][0 : j]
9:              for all P ∈ result do
10:                 if rootPath = P[0 : j] then
11:                     E_current ← E_current \ (P[j], P[j + 1])
12:                 end if
13:             end for
14:             V_current ← (V_current \ rootPath) ∪ spurNode ∪ s
15:             spurPath = dijkstra((V_current, E_current), spurNode, t, cost)
16:             totalPath = rootPath ∪ spurPath
17:         end for
18:         if |candidates| = 0 then
19:             break                                   // No further s − t path exist
20:         end if
21:         result[i] ← c ∈ candidates : length(c) = min({length(x)|x ∈ candidates})   //
    Candidate with minimal cumulated edge weight
22:         candidates ← candidates \ result[i]
23:     end for
24:     return result
25: end function
```

Algorithm 2.2 shows the pseudo code of Yen's algorithm. The input is a weighted directed graph $G = (V, E)$, a start vertex $s \in V$, a target vertex $t \in V$, a cost function $cost : E \to \mathbb{R}_{\geq 0}$, and the number of to be searched shortest paths k. The first step is the computation of the shortest $s-t$ path using Dijkstra's algorithm. The other $k-1$ shortest paths are determined in ascending order. The i-th shortest path is determined by looping over the vertices of the $i-1$-th path and maintaining a set of possible shortest $s-t$ paths called "candidates". As the set of vertices and edges is manipulated the set of vertices V is copied to $V_{current}$ and the set of edges E is copied to $E_{current}$. The "spurNode" is the j-th vertex of the $i-1$-th path. The "rootPath" is the $i-1$-th path but using only the first j

vertices. To avoid duplicated nodes, the edges of all previous shortest path from the "spurNode" to the next vertex of the previous shortest path are removed if the previous shortest path contains the "rootPath". All vertices of the "rootPath" except the "spurNode" are then removed from the set of vertices $V_{current}$. The set of candidates is then extended by the path beginning with the "rootPath" and ending with the shortest path from the "spurNode" to t. After all possible root paths of the $i-1$-th shortest path are handles the i-th shortest path is taken from the set of "candidates" by selecting the shortest path out of these "candidates". If the set of "candidates" is empty, no further $s-t$ path exists and the algorithm terminates and returns all yet found paths in ascending order by their length. The result of Yen's algorithm is the list of the k shortest $s-t$ paths [Yen71]. To find the k shortest cycles through a vertex s Yen's algorithm can be used by splitting s into two vertices s_1 and s_2 and introducing an edge (s_1, s_2).

2.4.5 Bhandari

Bhandari suggests an algorithm to find the k edge-disjoint $s-t$ paths whose accumulated edge weight is minimal for a given weighted graph $G = (V, E)$ in polytime.

Algorithm 2.3 Bhandari's algorithm [Bha99b]

1: **function** BHANDARI($G = (V_1, E_1), s \in V, t \in V, cost, k$)
2: $E_{P_1} \leftarrow$ dijkstra($G, s, t, cost$)
3: **for** $i \leftarrow 2$ to k **do**
4: $(I_{P_i}, cost_{modified}) \leftarrow inverse(E_{P_i}, cost)$
5: $G_{modified} = (V_{i+1}, E_{i+1}) \leftarrow (V_i, (E_i \setminus E_{P_i}) \cup I_{P_i})$
6: $E_{P_{i+1}} \leftarrow$ modified_dijkstra($G_{modified}, s, t, cost_{modified}$)
7: **end for**
8: $R \leftarrow \bigcup_{i=1}^{k} E_{P_i}$
9: $R \leftarrow$ remove_opposite_edges(R)
10: **return** R
11: **end function**

Algorithm 2.3 shows the pseudo code of Bhandari's algorithm. The input is a weighted, directed graph $G = (V, E)$, a start vertex $s \in V$, a target vertex $t \in V$, a cost function $cost : E \rightarrow \mathbb{R}_{\geq 0}$, and the number of searched paths k. The first step is the search for the shortest $s-t$ path in G using Dijkstra's algorithm. The second step is to find the other $k-1$ shortest paths iteratively. To find each of the other $k-1$ shortest paths the graph of the previous iteration is modified to ensure edge-disjointedness by removing the edges of the previous path and adding their inverse counterpart. An edge is inversed by switching its direction so that (x_i, x_j) becomes (x_j, x_i) and assignment of the negative $cost_{modified}((x_j, x_i)) = -cost((x_i, x_j))$. Using the modified version of Dijkstra's algorithm, which allows negative edge weights, for $G_{modified}$ reveals the i-th path P_i and its edges E_{P_i}. To assemble the final k $s-t$ paths the edges of all k paths E_{P_i} are merged. Pairs of edges between the same two nodes with opposite direction and the accumulated edge weight of 0 cancel each other out and are removed using the function "remove_opposite_edges". The result of Bhandari's algorithm is the set of edges of the k edge-disjoint paths with the smallest cumulated edge weight, which can be assembled to form the desired paths [Bha99b].

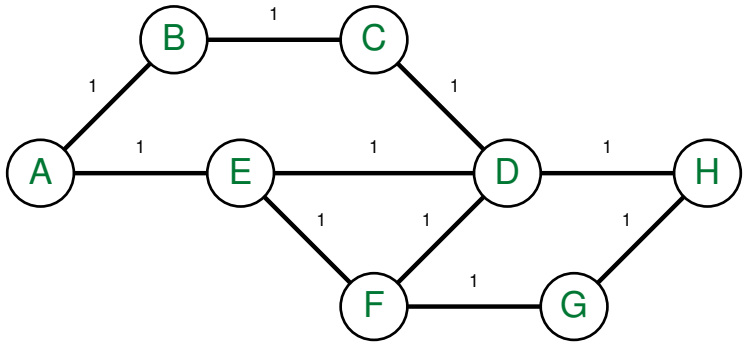

Figure 2.2: Example graph for Bhandari's algorithm [Mac15]

Figures 2.2 to 2.5 depict an example of Bhandari's algorithm to elaborate the algorithm in more detail. The input graph $G_{example} = (V, E)$ is shown in figure 2.2. It has the set of vertices $V = \{A, B, C, D, E, F, G, H\}$ and the set of undirected edges $E = \{\{A, B\}, \{A, E\}, \{B, C\}, \{C, D\}, \{D, E\}, \{D, F\}, \{D, H\}, \{E, F\}, \{F, G\}, \{G, H\}\}$. The edge weight of each edge is 1 and therefore $\forall e \in E : cost(e) = 1$. Bhandari's algorithm is used to find the two shortest edge-disjoint $A - H$ paths. Therefore, the start vertex is $s = A$ and the target vertex is $t = H$.

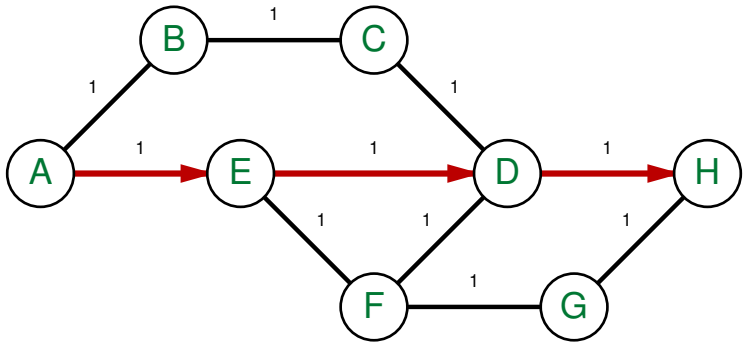

Figure 2.3: Step 1: Finding the shortest $A - H$ path of figure 2.2 [Mac15]

The first step is the execution of Dijkstra's algorithm to find the shortest $A - H$ path. As Dijkstra needs a directed graph each edge $\{x_i, x_j\} \in E$ is replaced with the two edges (x_i, x_j) and (x_j, x_i) pointing in opposite direction both with the same weight as the original edge. Figure 2.3 shows the result of the run of Dijkstra's algorithm marked in red. The shortest $A - H$ path P_1 goes through the vertices $\{A, E, D, H\}$ using the edges $E_{P_1} = \{(A, E), (E, D), (D, H)\}$ and has the accumulated weight of 3.

To look for the second shortest path and ensure edge-disjointedness the input graph $G_{example}$ needs to be modified. Each edge $(x_i, x_j) \in E_{P_1}$ is removed from $G_{example}$ and replaced with an edge in the opposite direction with the negative weight $cost_{modified}((x_j, x_i)) = -cost((x_i, x_j))$ of the original edge. For all other edges $e \in E \setminus E_{P_1}$ the weight does not change and therefore the cost function is equal to the unmodified cost function $cost_{modified}(e) = cost(e)$. Note that the

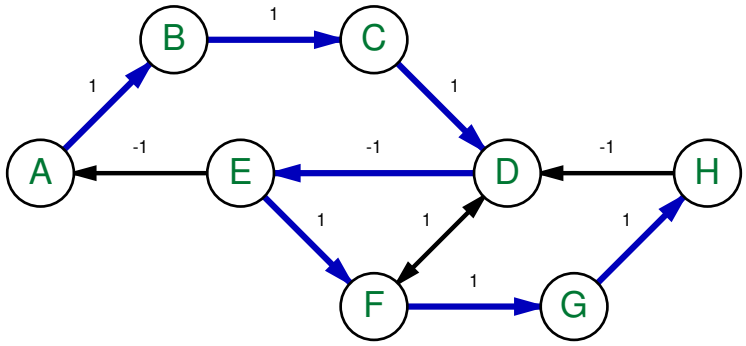

Figure 2.4: Step 2: Finding the shortest $A - H$ path in the modified graph [Mac15]

edges in opposite direction of the edges in E_{P_1} are omitted as they link to the same edge of the original undirected graph $G_{example}$. The resulting graph $G_{modified}$ is then used as input for finding the second shortest $A - H$ path using the modified version of Dijkstra's algorithm. Figure 2.4 shows the results of the run of the modified version of Dijkstra's algorithm marked in blue. The shortest $A - H$ path P_2 of $G_{modified}$ goes through the vertices $\{A, B, C, D, E, F, G\}$ using the edges $E_{P_2} = \{(A, B), (B, C), (C, D), (D, E), (E, F), (F, G), (G, H)\}$ and has the accumulated weight of 5.

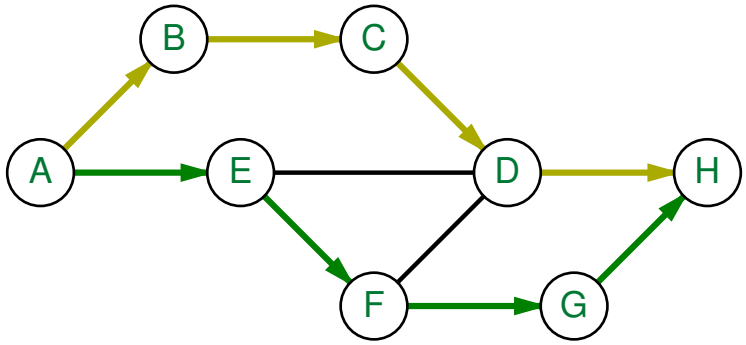

Figure 2.5: Step 3: Cancel out opposite edges [Mac15]

Figure 2.5 shows how to construct the two shortest edge-disjoint $A - H$ paths out of P_1 and P_2. If there are two edges $e_1 \in E_{P_1}$ and $e_2 \in E_{P_2}$ connecting the same two vertices in opposite direction, these two edges cancel each other out. The correctness of the shortest path search is not affected as the sum of both edge weight is 0 due to the construction of $G_{modified}$ using the negative cost of the original edge. Therefore, the resulting paths are the two shortest $A - H$ paths. To assemble the two resulting paths the remaining edges of P_1 and P_2 needs to be joined. Starting from A there are only the two starting edges of the resulting paths left. By following these two edges and using the only available edge at each vertex results in the two searched paths. Figure 2.5 shows these two paths in green and yellow. To generate the k shortest paths the graph modification and path search using the modified version of Dijkstra's algorithm need to be repeated until k paths are found. The final path assembly is then done like the version for $k = 2$.

2.5 Service

While web services and service computing are ubiquitous the most relevant parts are recapitulated here. Also, a short summary of problems with concurrency are stated, which are relevant when services process requests in parallel.

"A service is a self-contained unit of software functionality, or set of functionalities, designed to complete a specific task such as retrieving specified information or executing an operation. It contains the code and data integrations necessary to carry out a complete, discrete business function and can be accessed remotely and interacted with or updated independently." [Red20b]. A service is called stateless if it does not store information about the past or other interactions. The state of interaction lasts only as long as the request takes. After each request the service returns to a state without knowledge of the former interaction and works independently of other requests [Red20a]. Therefore, stateless services can easily be scaled horizontally.

An application programming interface (API) is the set of interfaces of an application, which can be used to integrate with the application without knowledge about the application's internal structure. The API can therefore be seen as the specified contract between two application, which allows them to interact and integrate. The API of a service defines the structure of request and the form of response and is therefore fundamental for any interaction and communication with this service [Red17].

2.5.1 REST

Representational State Transfer (REST) is an architectural style for distributed systems and mainly relevant for services using hypermedia. It is based on the stateless interaction between clients and servers. Servers provide interactive resources. Resources are represented using hypermedia and request are formulated using the Hypertext Transfer Protocol (HTTP) [Fie00]. Using REST and HTTP provides therefore a canonical architectural style for service APIs.

HTTP defines the interaction protocol for REST resources. Each request and response message consists of a status code, a resource representation as payload, and header fields describing the payload.

Verb	**Summary**	**Expected response**
GET	Transfer of a resource to the client.	200 OK
POST	Manipulate a resource.	200 OK
PUT	Create a new resource.	201 Created
DELETE	Remove a resource.	204 No Content

Table 2.5: Selected HTTP verb definitions

Each HTTP method is linked to a HTTP verb, which specifies which HTTP method is requested. Table 2.5 contains a selection of HTTP methods for accessing resources by their HTTP verb. Also, the summary of the expected handling and the expected status code of the response are stated. A GET request requires a resource representation identified by a resource identifier to be transmitted

to the client. A POST request enforces that the request is processed on the resource based on the resource's specification. A PUT request demand that a resource is created or replaced based on the content of the request. A DELETE request requires the target resource or correlation to be removed from the server [FR14, pp. 24–30].

Code	Name	Class
200	OK	Success
201	Created	Success
204	No Content	Success
400	Bad Request	Client error
404	Not Found	Client error
500	Internal Server Error	Server error

Table 2.6: Selected HTTP status code definitions

Table 2.6 contains a selection of relevant HTTP status codes. Codes in the range $[200, 299]$ indicate that the request was processed successfully. The code range $[400, 499]$ is used for client errors, which indicate a syntactical or semantical invalid request. A code in $[500, 599]$ represents a server error that could not be resolved by the server.

The result of a successful GET or POST request is the status code 200 OK. The response of a GET request is the requested resource. The response of a POST request is the manipulated resource. The status code 201 Created is the result of a successful PUT request. It is used to indicate that a new resource was created. The response describes the created resource. A Location header field contains the address of the newly created resource. The code 204 No Content indicates an empty body. It is return after a DELETE request indicating that the resource was deleted.

The client error code 400 Bad Request notifies the client about an invalid request. The client error code 404 Not Found is used to indicate that the requested resource could not be found. The server error code 500 Internal Server Error is used to describe that the request failed because the server erred. The response payload body of all failed request should contain a description about the occurred error [FR14, pp. 57–71].

2.5.2 Concurrency

In order to process request in a performant way without interruptions of other parts of the system a service can use concurrency to process requests in parallel. Working in parallel can cause special kinds of failures, which must be addressed. The following definitions are based on the ISO standard for the C++ programming language. "Two actions are potentially concurrent if [...] they are performed by different threads, or [...] at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation" [ISO17, p. 18]

The list of possible faults and how to handle these to not became failures is as follows:

- **Data race**: “Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [...] The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other[...]. Any such data race results in undefined behavior” [ISO17, pp. 15–18]. Therefore, it is necessary to prohibit data races by using atomic operations and mutexes as synchronization operations as undefined behavior cannot be handled during runtime.
- **Race condition**: A race condition arises when a common resource is accessed concurrently, and the result is dependent on the timing of the requests. A program containing a race condition may produce different results during different executions. Unintended non-deterministic results based on timing are in general non-desirable and should therefore be tackled with correct synchronization [Pra11]. A race condition may lead to a data race, which must be prohibited due to its undefined behavior.
- **Deadlock**: “A deadlock is a condition that may happen in a system composed of multiple processes that can access shared resources. A deadlock is said to occur when two or more processes are waiting for each other to release a resource. None of the processes can make any progress” [Cam11]. As a deadlock is a fault, which may interrupt the overall system activity, the error that led to the deadlock must be fixed by correcting the interfering synchronization mechanisms.

3 Vision

Video games take place in virtual worlds and provide immersive adventures for their players. These virtual in-game worlds are visualized and shown on a display. The player can influence the game by using some kind of input device. Music and ambient sound effects are made audible using speakers.

Connected cars extend cars by additional hardware and software in order to gather detailed data about the car, its hardware, and its surrounding and connect the cars with the cloud and each other. As part of the additional hardware, connected cars are equipped with powerful computers and high-resolution displays with quick response times. This allows video games to be played in a connected car using the built-in display, speakers and input devices.

But connected cars also offer access to external data sources and additional hardware that goes beyond display, speakers and input devices. This offers the possibility to augment the gaming experience and introduce more realism and therefore immersion to the game. In order to take advantage of these additional abilities in video games, the video games need to be adapted and extended to integrate the new possibilities. When integrating, the question arises how the real-life relates to the in-game world and how data can be exchanged.

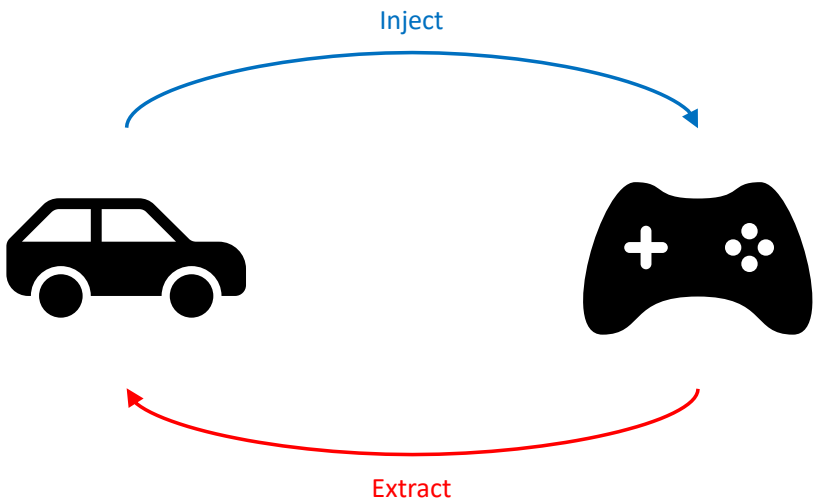

Figure 3.1: The data exchange model

To elaborate the relations between real-world data and in-game data and how data can be exchanged to connect the real-life with the video game a data exchange model is introduced. Figure 3.1 contains this data exchange model for data exchange between connected cars and video games adapted for connected cars. The real-life is represented as car on the left side of the figure, and the game is represented as video game controller on the right side. The blue arrow labeled with "Inject" describes the data transfer from the real-life into the game. The data injection into the game takes effect by altering the in-game state and keeps the game in sync with the real-life. The red arrow labeled with "Extract" describes the data transfer from the in-game world to the real-life. To transfer the game data, the in-game state is extracted and can then be used to align the real-world surrounding to the in-game reality.

Figure 3.2: Car halting on a street

In order to demonstrate the amount of data of an everyday situation, an example is examined. Figure 3.2 shows a car on a regular real-world street and serves as a representative example for the real-world environment of cars. The car is in front of red traffic light. The stoplight shines red indicating that the car is braking in order to stop. Behind the traffic light there is a construction site on the left half of the street. The street has been repaired at least once, which can be seen on the heterogeneous thickness of the road pavement. A cyclist is waiting in front of the car. On the left there are three pedestrians sitting on a bench. The street is surrounded by trees. The picture is taken in fall and therefore the leaves are yellow, have fallen and are now on the ground surrounding the trees. There are no leaves on the road surface. In the front of the picture a traffic sign and a streetlight are visible. In the background are buildings and parked cars. There is a corn field on the left and grassland on the right. The sun is shining and the sky shines in light blue.

Figure 3.3 shows a screenshot of Bing Maps [Mic21]. The position of the car of figure 3.2 is marked with a pushpin. It is used to illustrate the additional data that is neither visible nor detectable by sensors but accessible through the cloud infrastructure of the connected car. The additional spatial data visualized is the road network including street names, administrative division, vegetation,

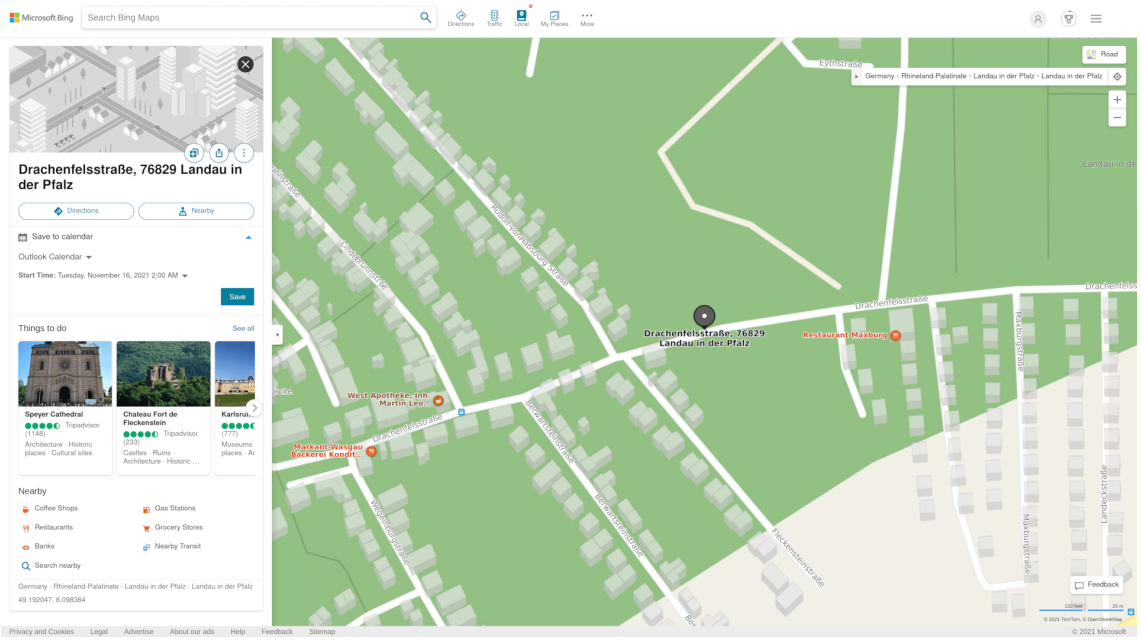

Figure 3.3: Bing Maps screenshot of the surrounding of figure 3.2

abstract 3D models of the buildings, and points of interest like restaurants or bus stops. Further map related data like routing, traffic information, or satellite images are directly reachable by selecting the appropriate option in the menu. Other location related data that goes beyond pure spatial data like opening hours, event schedules, or creation of calendar appointments at a specific location is visualized and accessible in the same way.

This everyday situation shows how much information is accessible using the visual information and external data sources located in the cloud. Further external data sources can be used to access additional data like weather forecasts or leverage the locally available data by adding additional information or usage of external compute power. In addition, cars can access sensor data like temperature, wind, distance to other road users, current speed, steering, motor and oil temperature, air conditioning settings, remaining fuel, car and ambient light settings, information about the audio system and any other value, which is measurable by an installed sensor. But also, the video game itself produces data, which can be gathered and analyzed.

Applications, which benefit from the advanced integration of video games in connected cars, can be put into two categories. The first category is data “injection” and contains the applications that use the additional information inside the game. The second category is data “extraction” and covers the applications that use the extracted data from the game. Possible applications that go beyond current limitations and benefit from the additional data are listed below.

3.1 Injection

Using data injection of real-world data into the running game enables many use cases. The list of possibilities covers:

- **Manipulation of in-game state**: This makes it feasible to adapt the current state of the game while the game is active and allows the reaction to environmental changes in or around the car while keeping the immersion. E.g., if it starts to rain it can also start to rain in-game.
- **Real-world surrounding**: To create an immersive in-game environment the real-world environment of the player can be mapped to the in-game world. This includes terrain, objects and textures. The data sources can be local sensors to determine the surrounding as well as a data lake in the cloud providing data for the regional city or landscape architecture, the road network including street types and road signs, rivers, train or aircraft schedules, 3D models of buildings or further advanced scenery options. As an example, the road data at the position of the car can be used to generate in-game streets. The player is then able to drive within the game along the streets that surround them in real-life.
- **Actual user / car**: To increase realism further 3D models of the actual user respectively the actual car and the surrounding road users can be used as in-game characters.
- **Realistic input devices**: By using the car's steering devices like steering wheel, blinker, or accelerator pedal as input device for the game it is possible to convey a realistic driving experience when the player drives through the in-game world.

3.2 Extraction

By extracting and gathering the in-game state further possibilities arise. These include:

- **Mapping of in-game state**: It is possible to map the in-game conditions to the car's hardware. This includes to activate the car's fans to map in-game wind conditions, the usage of the car's air conditioning to transmit the in-game temperature, or to splash washer fluid over the windscreen to simulate rain.
- **Augmented reality**: The extracted data can also be used to create augmented reality environments. In combination with full autonomous driving this can be used by the passengers to play augmented reality games inside their car while driving.
- **Telemetry**: Collecting the extracted data for big data analysis can be used to find out what players actually do in-game and how to improve the games based on this analysis. Also the analysis results can be used to introduce comprehensive player statistics, leaderboards, achievements, or special user challenges in order to increase the players's motivation.

3.3 Prototypes

As representative example for sensible integration of video games in connected cars, the kart racing game SuperTuxKart is examined. It also serves as foundation for further research projects, prototypes, and feasibility studies to show the potential of advanced integration of video games into connected cars.

Figure 3.4: SuperTuxKart surrounding of karts

Figure 3.4 shows a screenshot of SuperTuxKart running a race on the track “Cornfield Crossing” using the character “Beastie”. The kart is shown in the center. The race takes place between multiple cornfields on a dirt road. On the left side a combine harvester produces hay bales. A power line runs through the landscape. There are mountains in the background. The sky is realized as a blue and purple color gradient. In the lower left corner, a minimap of the current track is visible. Three bonus boxes are places in front of the kart.

In comparison to the real-world environment of figure 3.2 the in-game world of figure 3.4 reveals many similarities. Both worlds offer lanes on which the vehicles can drive, have a surrounding landscape with fields and trees, and have additional road user like the combine harvester in-game and the cyclist and pedestrians in the real-world image. Also, both worlds can have objects placed around the streets, e.g., buildings, construction sites, street signs, trees, benches, traffic lights, street lights, leaves or parked vehicles. Weather effects like sky color, sun, and shadows are also available in both worlds. But there are differences. One main difference is the lack of details in the in-game world. Reasons for that lack of detail are for example the low resolution of in-game textures and their repetition, or the lower number of placed objects in-game. As a concrete example for missing details from the figures the road surface in-game is just a repetition of one dirt texture whereas in real-life the road surface can differ, e.g., like the repaired asphalt in the real-world image. The reduced number of objects can be seen at the lack of stoplights in-game. Another main difference is the usage of a comic-style art, which further reduces realism. Also, there are minor differences

to improve gameplay in-game. Instead of realistic cars fictional karts and characters are used. To show additional information the game uses additional user interface components like a minimap to show the road network. In the real-world car this information is visible on an additional screen as in figure 3.3 and may not be visible for the driver all the time. Additional objects like bonus boxes can be added to the world. Another fundamental difference is that the real-life changes during a day, during the change of season, or because of environmental changes whereas the in-game world remains rather static. All these gameplay changes make the game more enjoyable but less realistic.

This comparison shows that the in-game world is a simplified version of the real-life. Details are omitted, additional user interface elements and fictional items are placed. But regarding the similarities the in-game world can be used as foundation for exchanging information between the real-life and the game. To put real-world data into the in-game world this data needs to be mapped to the simplified model of the in-game world and to use the data extracted out of the in-game world this data may need to be extended with further details. Additional gameplay elements need to be mapped to adequate real-world counterparts.

To approach this vision for games in connected cars the realized prototypes are based on two stages of the game life cycle. The first stage covers any action that is done before the start of the game. The second stage consists of everything that takes place while the game is active. Anything after the end of the game can be seen as before the next game and thus be part of the first stage. As starting point for projects for the first stage, the first realized prototype is the usage of existing data to form the in-game world. This world is then loaded at the beginning of the game and transfers the player into a virtual reproduction of the real-life surrounding them. To fit into the world model of the game this reproduction is of abstract manner and changes the reality in ways that integrate into the gameplay but reduce realism. This prototype can then be used as an example or starting point for more sophisticated concepts or projects. The second prototype is the creation of a REST API for SuperTuxKart. This REST API serves as abstraction layer for the SuperTuxKart internals and is the easy-to-use and extendable foundation for further projects that implement the vision for video games in connected cars. It eases the exchange of information and state from and into the game and allows the user to manipulate the game or the surrounding while the game is running. It is in the second stage as it is launched with the game and terminated with the end of the game.

4 World Generation

The first prototypically realized idea of the data injection part of the vision for advanced integration of video games into connected cars is the "World Generation" for SuperTuxKart. It covers the generation of a racetrack based on real-world data, which can be loaded by SuperTuxKart. After the injection of the generated racetrack, it can be selected for new races. Races can then take place in the generated track and provide an immersive replication of the real-life. The World Generation is done before the race start and therefore realized as performant, independent service, which can be consulted on demand.

4.1 State of the Art

There are already some suggestions to solve subproblems of the automated World Generation. SuperTuxKart tracks can be manually designed using Blender using a SuperTuxKart add-on. Further details are elaborated in section 4.1.1. Section 4.1.2 portrays the stand-alone application OSM2World, which can generate areal 3D models out of OpenStreetMap (OSM) data. As alternative for OSM2World section 4.1.3 presents a Blender add-on, which extends Blender with geographic information system (GIS) functionality and is able to generate 3D models out of OSM, terrain, and satellite data inside Blender. Relevant ideas for finding an appropriate racetrack within a road network with relaxed time and path conditions are listed in section 4.1.4.

4.1.1 SuperTuxKart

Tracks for SuperTuxKart are designed by hand using the open-source 3D computer graphics software Blender [Ble21] with a SuperTuxKart add-on. The SuperTuxKart add-on allows the export of the Blender 3D model into the SuperTuxKart track format. The first step is the manual creation of an in-game world using the regular Blender designer and utilities for 3D model construction. Then the racetrack path, checklines, objects, sound emitters, and bonus items are placed. Existing SuperTuxKart 3D models can also be embedded. To release the track, the meta data like track name, author, or music information is added. The completed track is then either submitted to be part of the next SuperTuxKart release or made available for download as add-on track. The SuperTuxKart documentation states some guides on how to create tracks [Sup16].

4.1.2 OSM2World

OSM2World is a stand-alone, open-source application that produces 3D models based on OSM data. The input is a file containing the OSM data and the output is the 3D model of the world in one of many common standard formats for 3D data. Figure 4.1 shows an example of a generated 3D

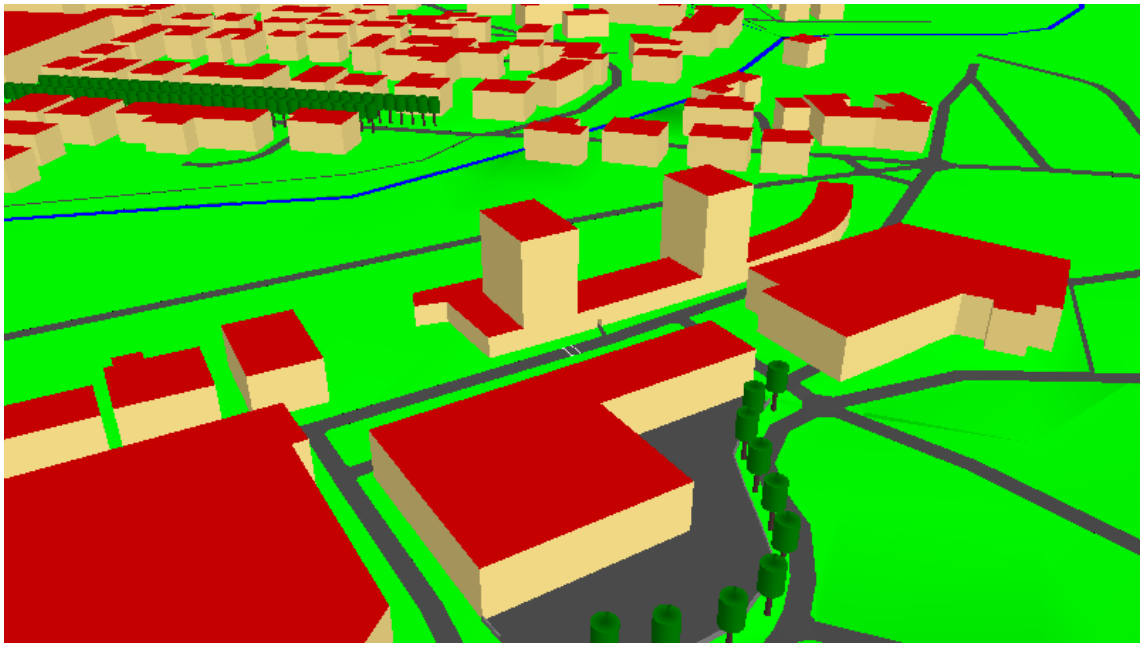

Figure 4.1: OSM2World exemplary mapping of OpenStreetMap data to a 3D model [Ope21c]

model using OSM2World. Buildings are mapped using their shape and height. Streets, railroads, and rivers are transformed to 2D shapes and made visible using colors. Objects like trees, benches, or bridges are mapped using an embedded set of predefined 3D models. Additional pre-built 3D miniatures of some historic sight and points of interest are also available. Only small map fragments are supported as according to the documentation inefficient algorithm are used and are not yet replaced with faster alternatives. There is an experimental support for elevation to support terrain data [Ope21c].

3D models generated by OSM2World can be imported into Blender and the regular SuperTuxKart track creation workflow of section 4.1.1 can be used to construct a SuperTuxKart track. Using the imported 3D model as foundation the racetrack path, objects, and bonus items need to be added by hand. Predefined 3D models, e.g., of trees, which enable advanced SuperTuxKart gameplay possibilities like sound effects, animation, or an adaptive level of detail (LOD), cannot be used to replace the OSM2World models of these objects without manual editing. In general, each replacement or modification of parts of the OSM2World 3D model requires manual editing using the appropriate Blender functionality. The workflow is in conclusion to manually generate the 3D model using OSM2World and the OSM data for a specific location as input, import the generated 3D model into blender, and then design the remaining parts of the SuperTuxKart racetrack by hand [Ope19].

4.1.3 Blender GIS

Blender GIS is an add-on for blender that enables GIS functionality within Blender. It allows the usage of satellite images, OSM data, and terrain data for elevation to automatically generate a 3D model of the real-life. The created models can then be used within Blender for further modeling or exported using the default Blender export functionality [Ble20]. Figure 4.2 contains a screenshot of

Figure 4.2: Blender GIS screenshot

Blender GIS using satellite images as ground textures and generated 3D models of buildings and streets highlighted in orange. Terrain information is used to generate elevation, which can be seen at the lowered buildings in the background.

4.1.4 Racetrack Generation

A road network can be mapped to a directed, weighted graph where each road segment corresponds to an edge and each crossroads corresponds to a vertex. The cost function assigns to each edge the linear distance between the initial and terminal vertex of the edge as weight. Out of the graph point of view a racetrack is a path or cycle in a road network. The objective of the racetrack generation is to find a racetrack in a road network with a specific length l, which starts at a specific vertex s. Willems et al. and Lewis each suggest an algorithm to find such a racetrack within a given road network.

Willems et al. suggest to use Yen's algorithm as described in section 2.4.4 to search for the shortest path with minimum length l. Yen's algorithm is adapted to not terminate after the first k shortest paths but after the first shortest path that is longer than l. To find cycles Willems et al. suggest to adapt the input graph by splitting s into two artificial vertices s_1 and s_2 and then use Yen's algorithm to find the path between s_1 and s_2 [WZR18]. Whereas the racetrack property of being at least as long as l but also being as close as possible to l enables many real-world use-cases it comes at the cost of k being not constant. By choosing l large enough the algorithm is able to find Hamiltonian cycles, which leads to an exponential worst case runtime complexity. But also k is a lower bound for the runtime complexity as at least k paths are found. Considering a grid graph as part of a road network and the shortest $s - t$ paths cross the grid graph. If l is chosen large enough, all cycles of the grid graph must be considered because Yen's algorithm finds all $s - t$ paths in ascending order. The number of cycles in a grid graph has a lower bound of $\Omega(2^n)$ [Mat17]. Therefore, also k has a

lower bound of $\Omega(2^n)$. This leads to an overall exponential worst case runtime complexity. As road networks in cities often contain grid graphs as subgraphs and l is large enough to cover some of these grid graphs this could lead to serious runtime issues.

Lewis suggests to use Bhandari's algorithm with $k = 2$ as heuristic for finding appropriate racetracks in polytime by starting at s and choosing the target vertex t far enough. The two found paths are then joined to get a cyclic racetrack. As the best t for finding the circuit whose length is closest to l cannot be determined upfront in polytime it is suggested to heuristic choose a vertex t being close to $\frac{1}{2} \cdot l$ away from s [Lew20]. As Bhandari's algorithm uses the shortest path search to determine both paths the paths are in general rather straight and have few sharp turns, which is also a desirable property of racetracks as the vehicles do not have to brake. But that does not always halt. E.g., the shortest path between two diagonal vertices in a grid graph consists of many rectangular turns. Another drawback is that Bhandari's algorithm only guarantees that the racetrack is edge-disjoint. Therefore, vertices may not be unique in the racetrack, which might lead to confusion while driving.

4.2 Requirements

The objective of the World Generation is to create an in-game world based on existing real-world data. Given a geo-spatial position it uses OpenStreetMap data as well as satellite images and public domain textures to generate a racetrack. This racetrack can then be loaded by SuperTuxKart, and the user can drive through this reproduction of the real-life. To integrate with SuperTuxKart a cyclic path based on the OpenStreetMap road network is computed and allows the user to score laps following this path through the in-game world. Along this path collectable bonus items are distributed. Objects like trees are mapped to pre-built SuperTuxKart models. Buildings are created and texturized based on their shape and their height. Table 4.1 shows the requirements set up for the World Generation project.

Table 4.1: World Generation requirements

№	Title	Definition
W1-1	Web service	The result of the project shall be a web service.
W1-2	Standalone	The service shall not require any SuperTuxKart coding to build or execute.
W1-3	Non-interactive	The service shall compute the result based on the input parameters without further user interaction.
W1-4	Performance	Each request shall be processed in at most the magnitude of one minute.
W2	Input	The service shall be able to process requests with the parameters position as pair of longitude and latitude and the size of the area that shall be replicated to the in-game world.

Continued on next page

Table 4.1: World Generation requirements (Continued)

№	Title	Definition
W3-1	Output	The output of each request shall be a SuperTuxKart track compressed as ZIP archive including all data that is needed to load and use the track in SuperTuxKart.
W3-2	World	The track shall contain a 3D model of the real-life.
W3-3	World size	The world size shall match the given size specified as parameter.
W3-4	Textures	Each structure of the 3D model shall be texturized based either on real-world data or following the SuperTuxKart art style.
W3-5	Racetrack path	Given an existing road network the service shall find a cyclic path through the road network starting and ending at the closest road segment to the input position.
W3-6	Path length	The path shall have approximately maximum length.
W3-7	Path intersections	The racetrack path shall not contain the same vertex or edge more than once.
W3-8	Path curvature	The racetrack path shall contain as few sharp turns as possible to allow the user to drive along the track with as little braking as possible.
W3-9	Sight	The world shall not contain parts that are out of sight following the racetrack path.
W3-10	Objects	Some pre-built SuperTuxKart objects are part of each SuperTuxKart installation. If there exists a natural real-world counterpart, it shall be mapped to the in-game object, i.e., a real-world tree shall be mapped to a SuperTuxKart tree object.
W3-11	Bonus items	Bonus items shall be distributed along the path through the in-game world.
W3-12	Level of detail	The in-game world shall be a compromise to be as realistic as possible but also match the performance requirements and integrate into the SuperTuxKart gameplay and art style.
W4	Real-world data	The service shall use freely accessible real-world data sources to compute the in-game world.
W5-1	Portability	The service shall operate platform independently.
W5-2	Extensibility	It shall be possible to add additional data sources without changing the existing data sources's implementation.
W5-3	Scalability	The service shall be stateless to allow easy scale-out.

4.3 Design of Solution

To fulfill the requirements W1-1, W1-2, W1-3, and W5-3 the World Generation application is implemented as a stateless service. Each World Generation is designed as request to the service. The input parameters of each request as defined by requirement W2 are position as longitude and latitude and the area that should be considered and mapped to the generated in-game world. The response and therefore result of the World Generation request as defined by requirement W3-1 is a SuperTuxKart track as ZIP archive containing all track constituents as described in section 2.3. The generated track can then be loaded into SuperTuxKart, and races can take place in a model of the real-world surrounding. As data sources OSM data and satellite images are used as foundation for the generated in-game world.

Figure 4.3 shows the process of the World Generation. The first step is to gather the OSM data for the specified position and area. The OSM data consists of the real-world road network and the positions of real-world objects. Using the OSM data as input the mapping of objects, the racetrack path computation, and the 3D model generation can be done independently and in parallel. The mapping of objects step maps the OSM objects to appropriate predefined SuperTuxKart objects. The result of the mapping is a list of in-game objects with their position, rotation, and scale. The racetrack path computation uses the road network and the real-world position of the car and finds a cycle through the street graph with a specific minimum length and no duplicated vertices or edges. Using the racetrack path as input the quads, and the checklines can be computed, and the bonus items can be placed. To create the 3D model in addition to the OSM data also the satellite images are used. The step “Gather satellite images” downloads the satellite images from a map tile server. Each tile of the map corresponds to one satellite image. All map tiles for the selected area with their satellite image as texture form the basemap for the in-game world. The step “Generate world model” generates building models for the building shapes defined in the OSM data and places them on the basemap. The combination of basemap and buildings then results in the 3D model in the SPM format as defined in section 2.2. The “Objects”, “Quads”, “Checklines”, “Bonus items”, and the “3D model” are finally packed and result in the track archive in the ZIP format.

4.3.1 Object Mapping

To fulfill requirement W3-10 suitable real-world objects of the OSM data are mapped to in-game objects. The mapping of objects is a one-to-one mapping of OSM objects to SuperTuxKart objects. The SuperTuxKart objects are predefined 3D models included in the default SuperTuxKart installation and can be placed in the in-game world in an arbitrary amount and at arbitrary positions. The object mapping step consists of filtering the OSM object nodes for objects that have an appropriate predefined SuperTuxKart counterpart, e.g., trees, and transforming the OSM object to its in-game counterpart. The result of the transformation is for each suitable real-world object a tuple consisting of the identifier of the predefined SuperTuxKart object, the position of the object transformed to the in-game coordinate system, the scaling, and the rotation of the 3D model. Finally, the resulting list of tuples is mapped to XML elements following the SuperTuxKart schema and placed in the “scene.xml” file.

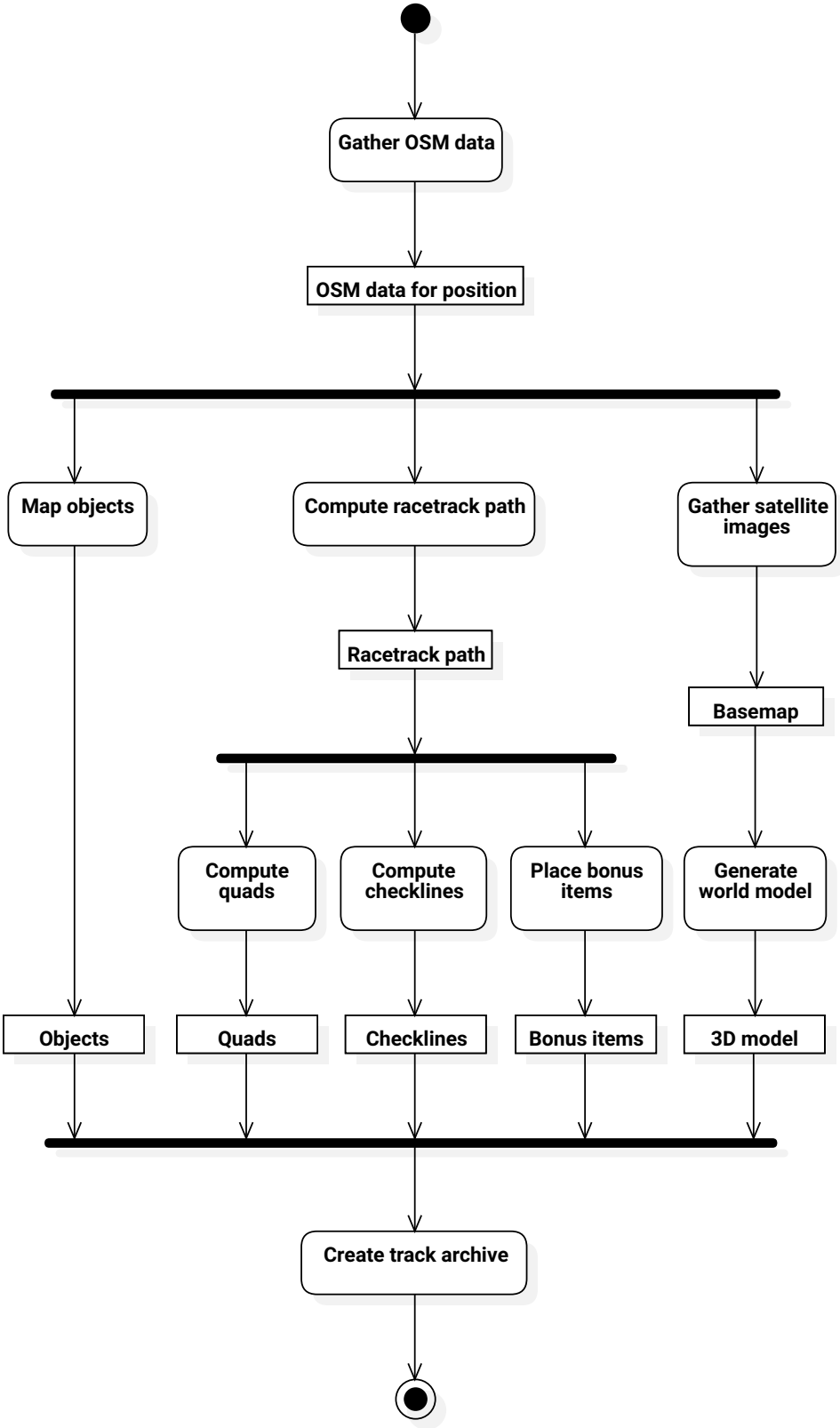

Figure 4.3: UML activity diagram of the World Generation

4.3.2 Racetrack Path

The SuperTuxKart's guidelines for gameplay define that a racetrack follows exactly one path through the world. This path should be unambiguous and clear to the user so that they always know where to go and where the surrounding terrain begins. While racing the karts follow the racetrack path to compete in the race. Many consecutive or tight curves should be avoided as they increase driving difficulty [Sup19]. Requirement W3-5 requires that such a racetrack needs to be found based on the OSM data. Therefore the OSM road network is used as input to allow karts to drive along the real-world streets, which is the natural mapping of real-world streets and avoids confusion. The usage of the road network as graph with inherent spatial meaning allows the usage of graph routing algorithms to find the racetrack path. According to requirement W3-6 the racetrack path should be heuristically as long as possible. Requirements W3-5, W3-6, and W3-7 can all be achieved by using Willems et al.'s suggestion to use Yen's algorithm, but requirement W3-8 states that sharp turns shall be avoided. Furthermore, the algorithm's runtime might be too high for road networks as required by requirement W1-4. Lewis idea of using Bhandari's algorithm to find racetrack paths fulfills W1-4, W3-5, W3-6, and W3-8, but to further reduce confusion, requirement W3-7 states that nodes must be unique in the racetrack path.

The idea to solve this problem is to use an extension of Bhandari's algorithm called Suurballe's algorithm. Suurballe's algorithm comprises all guarantees of Bhandari's algorithm and further guarantees that each vertex on the found path is unique [Suu74].

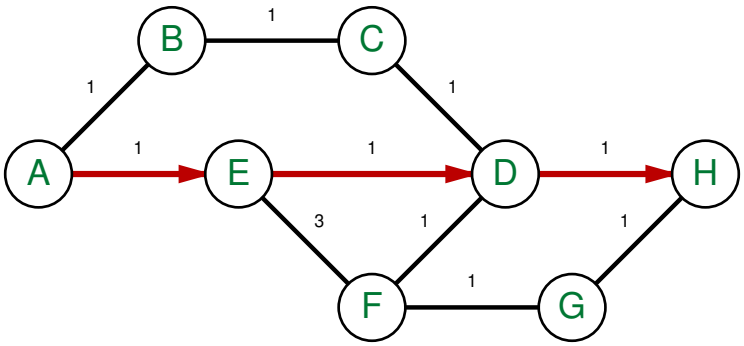

Figure 4.4: Example graph for Suurballe's algorithm [Mac15]

The differences between Bhandari's algorithm and Suurballe's algorithm are described with an example. Figure 4.4 shows the input graph. The objective is to find the two shortest $A - H$ paths, which can then be joined to a cycle. The first shortest $A - H$ path is marked in red. As in figure 2.4 the next step is the inversion of edges of the first shortest $A - H$ path and assignment of the negative weights of the original edges to the inversed edges I. After the inversion step Suurballe's algorithm splits each vertex v of the first shortest $A - H$ path into two vertices v_{in} and v_{out}.

Figure 4.5 shows the inversed edges and split vertices of figure 4.4. For each split vertex v an edge (v_{out}, v_{in}) with weight 0 is added. All original ingoing edges $(\cdot, v)$ are altered to $(\cdot, v_{in})$ and all original outgoing edges $(v, \cdot)$ are altered to $(v_{out}, \cdot)$. Each inversed edge $(v_i, v_j) \in I$ is altered to point from the "in" vertex to the "out" vertex and results therefore in the edge $(v_{i_{in}}, v_{j_{out}})$. The remaining steps of Suurballe's algorithm are identical to Bhandari's algorithm.

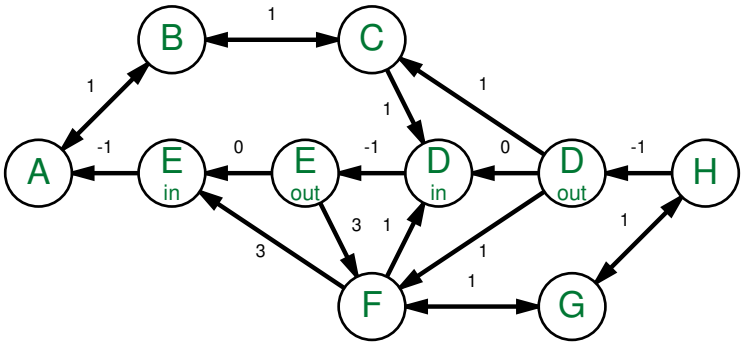

Figure 4.5: Example graph of figure 4.4 with inversed edges and split vertices [Mac15]

The steps to determine the racetrack path are therefore as follows:

1. Sort all road segments by their distance to the car's position.
2. Starting at the closest segment Dijkstra's algorithm is used to determine the shortest distance to all other vertices in the street graph.
3. Starting at the furthest vertex Suurballe's algorithm finds the corresponding second shortest path. Both paths can then be joined to form the racetrack path without duplicated vertices.
4. If there is no second path, go back to step 3 using the next furthest vertex.
5. If there is no racetrack path for the segment, go back to step 2 with the next closest segment.
6. If no racetrack path can be found in the whole street graph, the request fails.

Using this application of Suuraballe's algorithm enables finding long racetrack cycles in polytime. As the street graph is typically connected the algorithm usually terminates after the first execution of step 3. The resulting racetrack path is then used as input for further processing.

4.3.3 Quads

To allow SuperTuxKart to show the minimap of the track and enable the AI for karts to drive along the racetrack path SuperTuxKart requires the racetrack path segments to be enriched with track width information. Therefore, the displacements for a predefined track width of the racetrack path are computed and split to one quad per edge of the racetrack path. Finally, all quads are serialized by their endpoints in counterclockwise order as XML elements conforming the SuperTuxKart XML schema and stored into the file "quads.xml". The order of quads is defined in the file "graph.xml".

To adapt to curvatures, the displacements are computed by computing the angle bisector of two consecutive racetrack path segments. On the angle bisector on both sides of the intersection point an auxiliary point is added in a distance of 0.5 forming a slope segment with the length 1. Connecting two consecutive slope segments with one connection segment on each of both sides of the racetrack

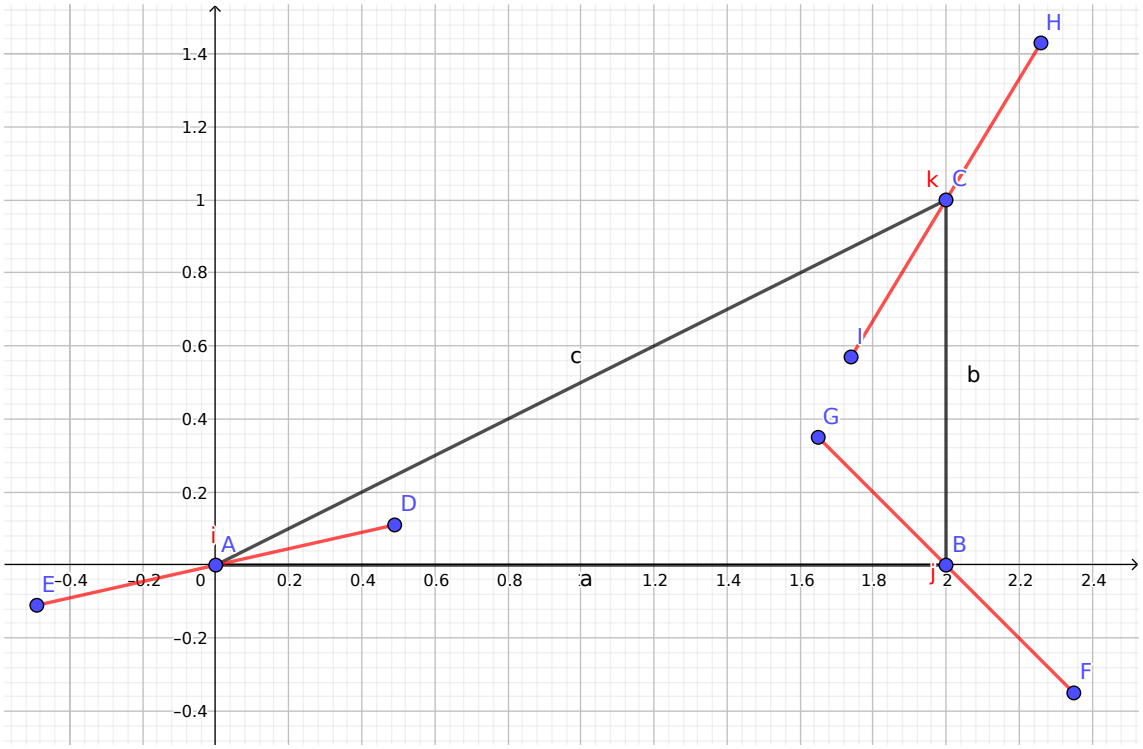

Figure 4.6: Quad construction based on triangular racetrack

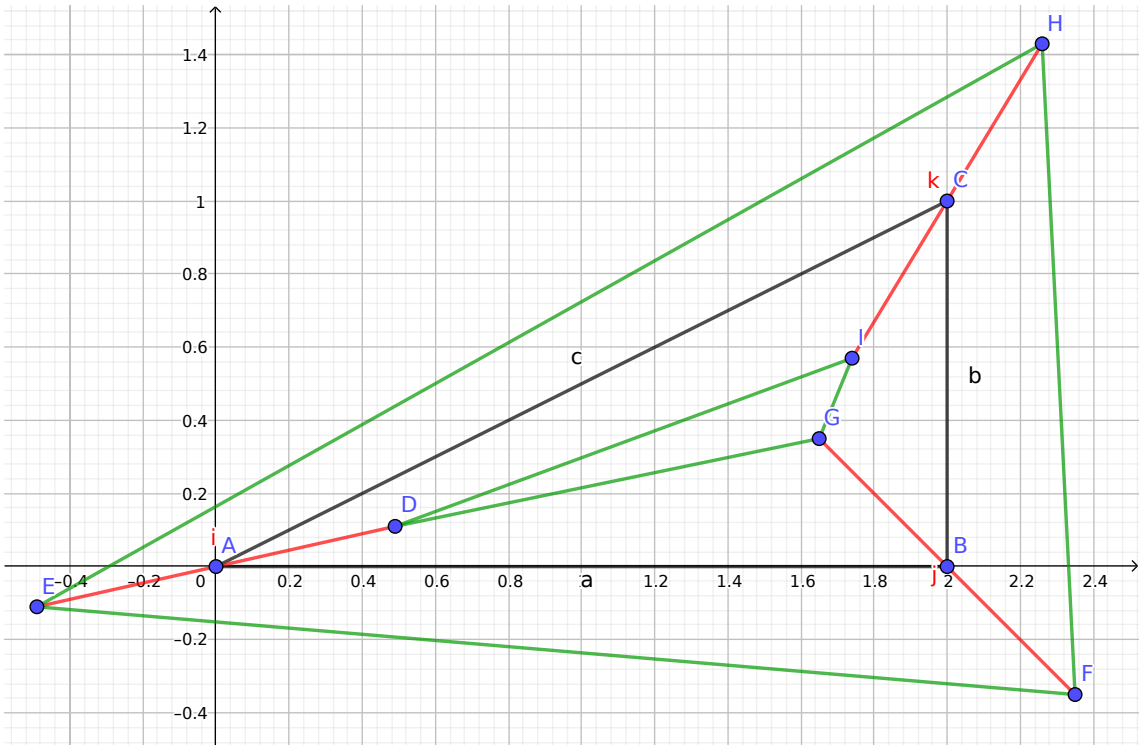

Figure 4.7: Finalize quad construction of figure 4.6 through connection segments

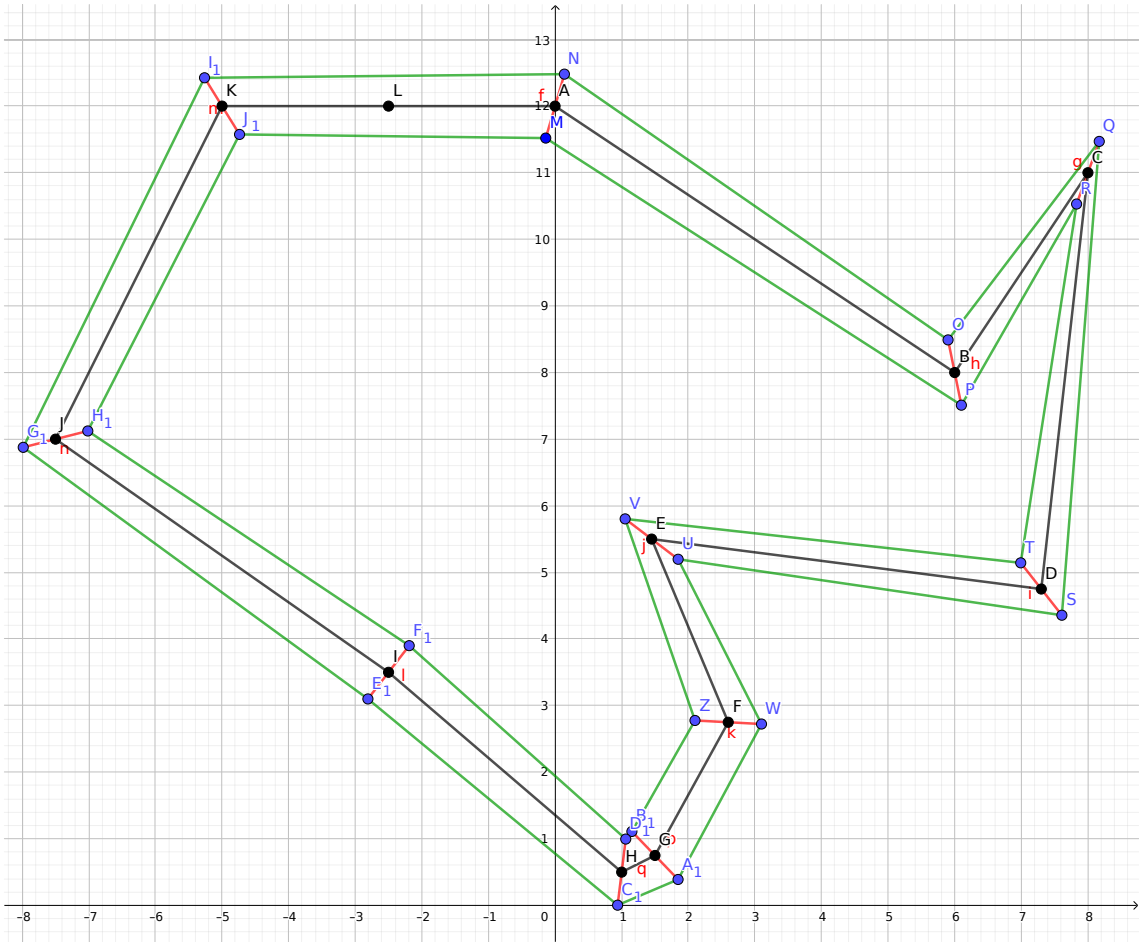

Figure 4.8: Real-world quad construction example

path segment results in the quad for the racetrack path segment. Vertices on a straight line between their adjacent vertices are skipped during quad computation as this would not create new information but just split the surrounding quad.

The displacement segment computation based on a given racetrack path is described with the example shown in figure 4.6. The racetrack path is colored in black and goes through the vertices A, B, and C. The slope segments are marked in red with their auxiliary endpoints on both sides shown in blue. The first step to compute the slope is to compute the angle α between two consecutive racetrack path segments, e.g., a and b.

(4.1) $D_i = P_i - P_{i-1}$

(4.2) $\alpha_i = \text{arctan2}(D_{i-2}, D_{i-1}) - \text{arctan2}(D_{i-1}, D_i)$

As the order of auxiliary points is important for the serialization in counterclockwise order it is necessary to compute α_i as the left angle between the segments $i-1$ and i. It is important to not calculate the smaller angle between the two segments. Therefore, α_i can be computed as described in equation (4.2) by using the arctan2 and the directional vector D_i of the segment i. The computation of D_i is described in equation (4.1) with P_i being the endpoint of the segment and

P_{i-1} being the starting point of the segment. For further calculations α_i is also shifted to be in $[-\pi, +\pi]$.

$$T(\phi) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix} \tag{4.3}$$

$$T_{left}(\alpha) = T(\frac{1}{2}\alpha + \frac{\pi}{2}) \tag{4.4}$$

$$T_{right}(\alpha) = T(\frac{1}{2}\alpha - \frac{\pi}{2}) \tag{4.5}$$

$$S_{left_i} = P_i + \frac{1}{2} \cdot T_{left}(\alpha_i) \cdot \frac{D_i}{|D_i|} \tag{4.6}$$

$$S_{right_i} = P_i + \frac{1}{2} \cdot T_{right}(\alpha_i) \cdot \frac{D_i}{|D_i|} \tag{4.7}$$

Using the counterclockwise 2D transformation matrix for the angle ϕ described in equation (4.3) the endpoints for the slope segments can be computed. The transformation matrix for an angle α along the angle bisector in the left direction is described in equation (4.4) and in right direction in equation (4.5). The computation of the left slope endpoint for segment i is defined in equation (4.6) and for the right slope endpoint in equation (4.7) by adding the normalized and transformed directional vector scaled to the length 0.5 to the endpoint of the segment. The scaling to 0.5 is used to set the length of the slope segments to 1. Connecting both endpoints of the slope segment results in the slope segment itself.

The addition of the connection segments of figure 4.6 to complete the quad construction is shown in figure 4.7. The connection segments are thereby colored in green. The resulting serialized quads in counterclockwise order are $DEFG$, $GFHI$, and $IHED$. As only the endpoints are required for the serialization the connection segment computation can be omitted as the endpoints are already determined for the slope segments and is only shown here to visualize the interpretation of the quad data in SuperTuxKart.

Figure 4.8 shows a complex example of the quad generation based on real-world data. The racetrack path is shown as black path. The computed slopes are depicted in red for each vertex of the racetrack path. The green segments represent the connection segments. Therefore, each quad consists of two red slope segments and two green connection segments of two consecutive vertices. Each quad is serialized by creating a tuple of the four endpoints of the segments in counterclockwise order. As visible for point L all vertices that are on a straight line between the previous and next vertex are omitted for quad generation.

4.3.4 Checklines

The checklines are required for lap counting and to ensure that the player does not skip parts of the track. The checklines must be crossed in consecutive order and no checkline can be omitted. Therefore, checklines are an essential constituent of each SuperTuxKart track.

The idea to implement checklines in the generated in-game model of the real-life is to use orthogonal lines to the racetrack path segments. The checkline is placed in the center of the segment with a specified width. The width of the checkline is chosen to be large enough to allow driving aside the road. The road segment must thereby be longer than a specified minimum length to allow cutting

edges as checklines must be crossed and putting a checkline on a short segment forces the kart to drive along that segment. Intersections with previous checklines must be avoided as this could lead to driving through the checklines in the wrong order while still following the racetrack path, especially in curves. Therefore, the idea is to loop through the racetrack path segments. If the segment is long enough and the checkline does not intersect the previous checkline, the checkline is added to the list of checklines. There needs to be at least one checkline to enable the lap counting functionality. Finally, each checkline including its predecessor is stored as XML element following the SuperTuxKart schema and placed in the "scene.xml" file.

The segment intersections can be computed by transforming the segments to linear equations and calculating their intersection point. If the intersection point is within the segment boundaries, the checkline is dismissed.

4.3.5 Bonus Items

To integrate into the SuperTuxKart universe and enable further gameplay opportunities like speed-up and guidance, requirement W3-11 requires bonus items to be places along the racetrack. It is thereby important to place the bonus items in the right distance between the previous and next bonus items to not interfere with previously collected bonus items but also to place enough bonus items be able to benefit from the gameplay opportunities. To realize the bonus item placement the idea is to loop over the checklines and place the bonus items uniformly on a straight line along the checkline if the distance to the previously placed bonus items matches a certain threshold. The threshold is thereby a fix distance and chosen to be approximately the distance that is needed to consume a speed-up bonus item as a compromise, which matches the SuperTuxKart gameplay for bonus items. Finally, each bonus item is mapped to an XML element following the SuperTuxKart schema and placed in the "scene.xml" file.

4.3.6 3D model

The track takes place in a 3D model of the real-life stored in the SPM format, which requires to map the real-life to texturized, or colored polygons. As in other current solutions like Blender GIS this is possible by using satellite images as basemap and generating building models based on the building shapes located in the OSM data. The basemap is thereby a plane on ground level. The satellite images are taken from a map tile server, which splits the map into squarish tiles. For each tile in the specified area around the car's location as specified in requirement W3-3 the corresponding satellite image is downloaded. As the tile is squarish it can be represented as two triangles. To match the requirements W3-2 and W3-4 tiles are texturized and assembled to form the basemap on which the karts can drive.

To further increase realism building models are created by using the shape of the building stored in the OSM data and augment it with height information. This height information is either given, or can be computed by the number of floors, or chosen randomly in a reasonable interval. Following requirement W3-4 the cuboids are then texturized by using public domain building textures of different building types and placed on the basemap at the position of the real-world building. The public domain building textures are cropped between the windows to be able to match different wide building walls. Therefore, to texturize a building, the building type is determined by the building's

height, and the walls are texturized by the wall's width with the cropped building images. Finally, the 3D model is rectangularly cropped along the outer boundaries of the racetrack path including some padding to remove parts out of sight as required by requirement W3-9 to reduce map size. The resulting SPM file is then packed into the final ZIP archive.

4.4 Realization of Solution

As the service shall be platform independent and easily extendible for others the service is implemented using the TypeScript programming language for the Node.js [Ope21a] runtime environment. Access to external data sources via HTTP is encapsulated using axios [Zab21]. Matrix operations are implemented using the math.js [Jon21] library. The resulting ZIP archive with all track constituents is packed using JSZip [KDBA21].

To implement the suggested solution the algorithm of figure 4.3 is transformed into an object-oriented application. Therefore, the class structure of the implemented application is elaborated. Furthermore, the structures of input data from OSM, and a map tile server are examined in detail.

4.4.1 Data

OSM data consists of the elements "nodes", "ways", and "relations". A node is a specific point on the map. It is identified by a unique identifier. Its position is defined by latitude and longitude. A way is a list of nodes, which expresses correlation between these nodes. For further processing the nodes of a way are linearly connected resulting in a polyline. Ways are used to represent roads, rivers, or shapes. A relation adds relation data to other elements, e.g., which set of ways form a highway.

For each of the basic elements, tags can be specified. A tag adds the intention to an element and is used to determine how to use the element. E.g., a tag specifies whether a way is a road, or a building. [Ope21b]. The OSM API allows access to all elements by specifying the bounding box of the requested area. The elements are then accessible and serialized as XML or JSON. For processing nodes can be used independently and therefore do not require other data to be accessible. As ways consist of nodes it is necessary to first process all nodes to make them accessible while processing a way. This makes the processing of OSM data a two-stage procedure.

Satellite images are available using a map tile service. The map is thereby a 2D projection of the geo-spatial data of the earth as ellipsoid in the well-known Mercator projection. As geo-data may be huge it is necessary to allow access to be restricted to a certain area or level of detail. Therefore, the map is split into tiles on different abstraction levels. Each tile corresponds to an image with the resolution 256x256. The splitting of map data on three different abstraction levels is shown in figure 4.9. As visible the first level consists of four tiles resulting in a total map size of 512x512. Each level of abstraction doubles the available resolution and therefore the accessible data. Therefore, each tile on the same abstraction level corresponds to four tiles on the next finer lever of detail.

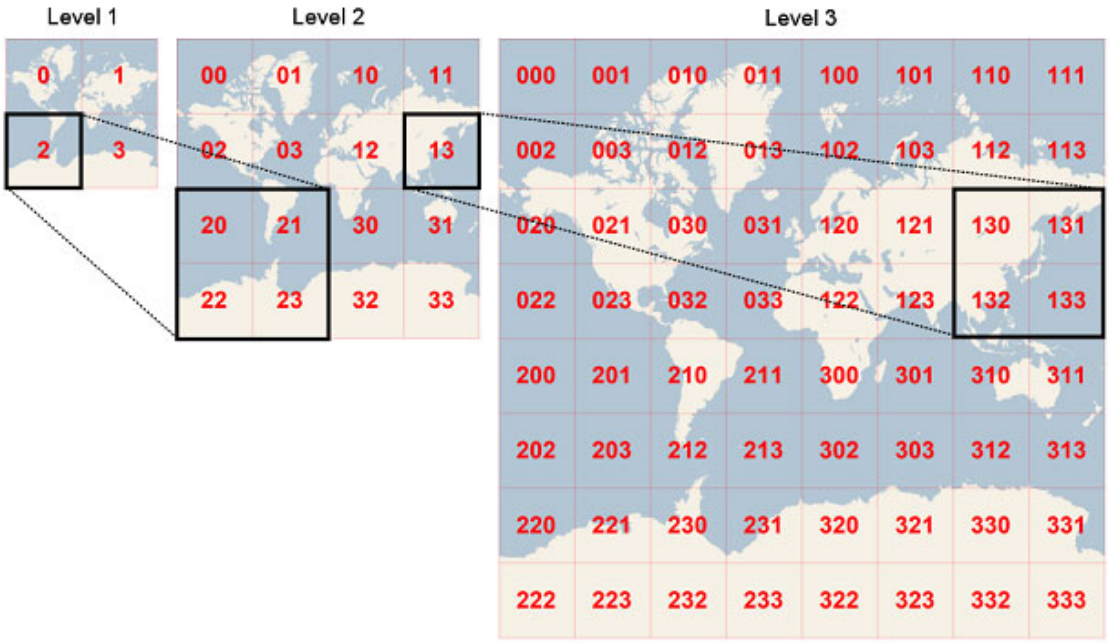

Figure 4.9: Map tile composition [Mic18]

To access a certain tile the tile needs to be addressable. Figure 4.9 shows the addressing of tiles on different abstraction levels for Bing Maps using a quadkey. The quadkey is thereby marked in red. Using the quadkey as identifier the tile data can then be downloaded as image [Mic18]. Other map tile server providers may offer different addressing schemes.

To gather the map data for a certain area on a certain level all tiles on this level intersecting the specified area are addressed and downloaded. When all tiles are downloaded, the tiles are merged and form the map data for the whole requested area.

4.4.2 Implementation

Figure 4.10 shows the implementation class diagram of the request handling for the World Generation service. The request handling is done by the `Server`. The `Server` receives a request, distributes the request to the `OsmDownloader`, the `StreetGraph`, the `TileGenerator`, the `BonusItemCreator`, the `ObjectMapper`, the `SpmCreator`, the `BuildingCreator`, and calls the `TrackConstituentCollector` to pack the track constituents to the resulting archive. The resulting archive is then packed and returned to the requester. The server endpoints listen for HTTP GET requests and expects the longitude, latitude, and the expected number of tiles as URL parameters. In case of an error, or invalid user parameters the logic of each class called by the `Server` throws an exception. The `Server` catches the exception and return the error code “400 Bad Request” in case of invalid parameters, or “500 Internal Server Error” in case of an error in the application logic. If the request is successful, the status code “200 OK” with the resulting archive as ZIP and the “Content-Type” header for ZIP archives are returned.

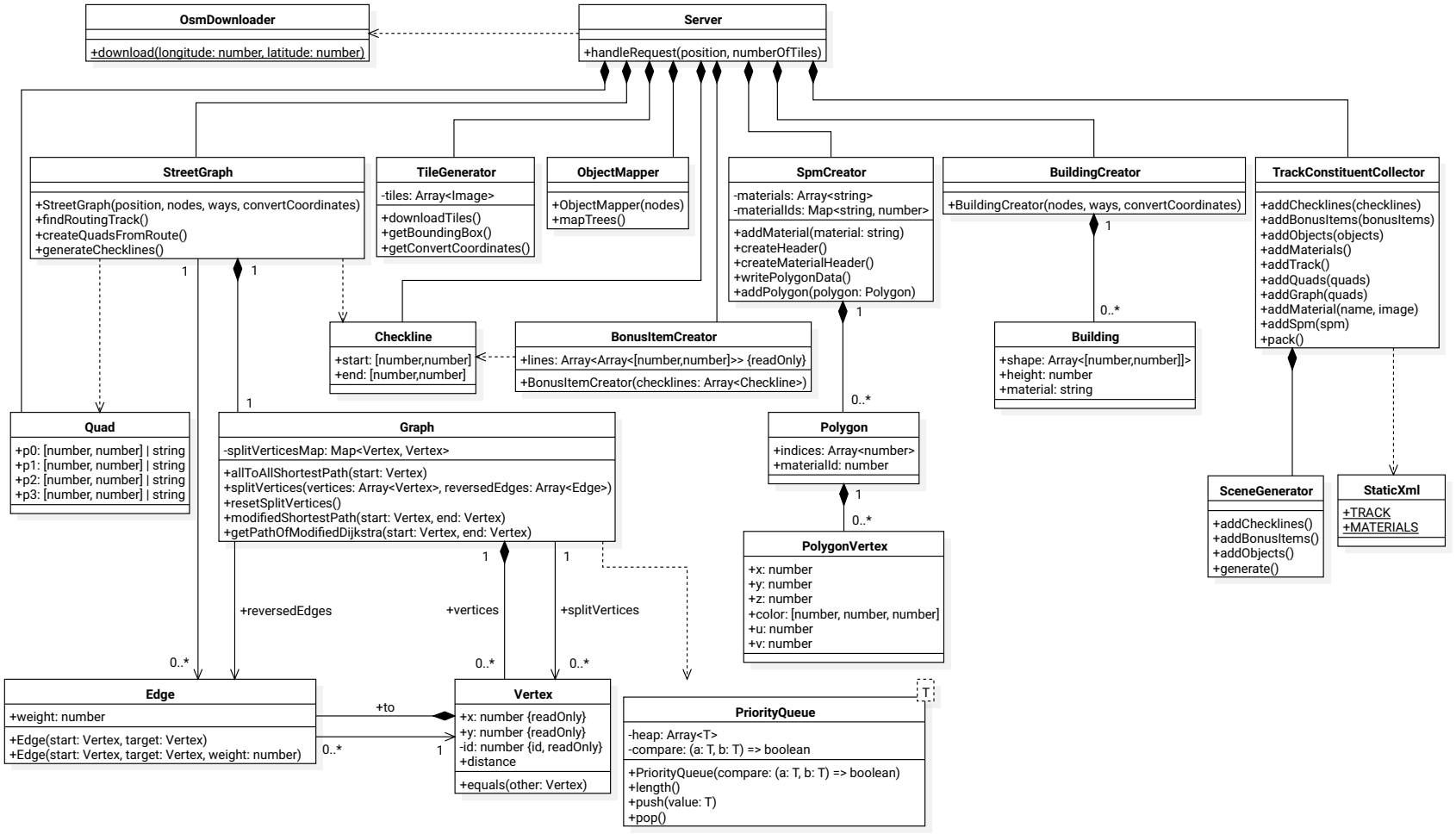

Figure 4.10: UML implementation class diagram of World Generation

Following the algorithm structure of figure 4.3 the processing is mapped to calls to the class hierarchy. The order of calls is thereby as follows:

1. The `Server` receives request for World Generation. If longitude, latitude, or the expected number of tiles are missing, the error code "400 Bad Request" is returned and the algorithm terminates.
2. The `Server` creates an instance of the `TrackConstituentCollector` to collected the computed results. The `TrackConstituentCollector` maps the computed results into the expected SuperTuxKart format. The `TrackConstituentCollector` uses thereby the `SceneGenerator`, and the `StaticXml` as helper classes. The `SceneGenerator` collects the results for the "scene.xml" file. `StaticXml` contains static XML files for "track.xml", and "materials.xml".
3. The `Server` initiates the download of the map tiles via the `TileGenerator`.
4. The `TileGenerator` download all map tiles in the specified area in parallel and computes the bounding box of all intersecting tiles.
5. The `Server` creates an instance of the `SpmCreator` for the creation of the 3D model of the in-game world. The `SpmCreator` collects materials and texturized polygons. Polygons are thereby represented as instances of `Polygon` to encapsulate vertex data and vertex correlations by a list of triangles consisting of the corresponding vertex indices. The polygon vertices are represented as instances of `PolygonVertex` to encapsulate position, color, and texture. The final 3D model in the SPM format is created by concatenating the results of `createHeader`, `createMaterialHeader`, and `writePolygonData`.
6. The downloaded tiles are added as materials to the `SpmCreator` instance.
7. The `Server` uses the static method `download` of `OsmDownloader` to gather the OSM data for the computed bounding box.
8. The OSM ways are filtered for building shapes and 3D building models are created by the `BuildingCreator`. Each building is thereby mapped to an instance of `Building` to encapsulate shape, height, and textures.
9. Using the tiles and building models the `SpmCreator` generates the final 3D model of the in-game world and stores it in the `TrackConstituentCollector`.
10. The `Server` creates a `StreetGraph` for the downloaded OSM data.
11. The `StreetGraph` converts the OSM nodes, and ways to a `Graph`. Nodes are mapped to instance of `Vertex`. Ways are mapped to instances of `Edge`. An instance of `Edge` is thereby from a `Vertex` to another `Vertex`, and has a weight stored as member.
12. The `StreetGraph` uses Suurballe's algorithm in `findRoutingTrack`. The `Graph` supports the needed operations via Dijkstra's algorithm as `allToAllShortestPath`, the modified version of Dijkstra's algorithm as `modifiedShortestPath`, the splitting and merging of vertices via `splitVertices` and `resetSplitVertices`. The method `modifiedShortestPath` thereby return the length of the shortest path or infinity if no path exists. To construct the final path the method `getPathOfModifiedDijkstra` is called. All instances of Dijkstra's algorithm and its modified version require `PriorityQueue` as a data structure for fast access of the next closest vertex and fast insertion of further vertices. `PriorityQueue` is thereby implemented as binary min-heap.

13. Using the computed racetrack path `createQuadsFromRoute` generates the quads as instances of `Quad`. Following the SuperTuxKart logic a quad consists thereby of four points either as coordinates or a reference to a previous point as formatted string.
14. The checklines for the racetrack are computed in `generateChecklines`.
15. Using the checklines as input the `BonusItemCreator` computes the bonus item placement.
16. The `ObjectMapper` maps appropriate OSM nodes to predefined SuperTuxKart objects.
17. All previously computed results are collected in the `Server`'s instance of the `TrackConstituentCollector`. The final step is the packing of the track constituents by calling `pack` of the `TrackConstituentCollector`.

4.4.3 Results

Figure 4.11 shows a race in a generated world of a small city. All buildings except one have one floor. Two different textures are used. The only taller building is the town's steeple. The basemap shows the street, some cars but also the shadows that where present while the satellite image was taken. The minimap and therefore the quads are visible in the lower left corner. As the buildings are small and numerous, large parts of the basemap are texturized as asphalt but also green plants are visible the scenery tries to convey the impression of a small city to the player.

Figure 4.12 presents a screenshot of a race taking place in a large city. The basemap shows a wide street with many lanes. The used satellite image also contains other cars visible on the right side of the player's kart. As the height information is used to generate tall buildings with skyscraper textures the scenery seems rather realistic and the kart seems to be small in-between the tall buildings. Also, no plants, or vegetation is visible, which increases the impression of being in a large city.

Figure 4.13 shows the start of a race on a generated track in a rural area. The karts are on a dirt road between fields and small buildings. The vegetation of the fields is visible through the satellite image texture. As large parts of the sky are visible and the fields are flat, the scenery creates the expression of an agriculturally used area.

Figure 4.14 visualizes the checkline placement. Each checkline is represented as 2D plane. The height and width of the checkline are visible by the extent of the planes. The currently active checkline is colored in red. All other checklines are colored in white. The checklines are one after the other along the racetrack path on street between the field and the buildings. The placement of wide checklines along the racetrack path forces the player to drive along the expected path while still being allowed to drive aside the street to a certain extent.

Figure 4.15 shows the results of the placement of bonus items and the mapping of objects on a generated track. The race takes place in a larger city between a tall building on the right, and a small building on the left. The bonus items are visible on the left side between the trees and are placed on a line orthogonal to the street. Each visible tree originates in a node with a tree tag in the OSM data. This demonstrates the mapping of real-world objects to objects of the SuperTuxKart universe while maintaining realism. The bonus items try tp ease orientation as they show where to drive while invoking bonus actions.

Figure 4.11: World Generation results: Small city

Figure 4.12: World Generation results: Large city

Figure 4.13: World Generation results: Field

Figure 4.14: World Generation results: Checkline visualization

Figure 4.15: World Generation results: Objects and bonus items

4.5 Evaluation of Solution

To evaluate the prototype and check whether the prototype satisfies all expectations the fulfillment of the requirements is validated, and the limitations of the prototype are elaborated. If there are no unfulfilled requirements and limitations within the scope of the prototype, the implementation of the prototype can be seen as successful.

Table 4.2: World Generation requirement evaluation

№	Title	Evaluation	Fulfilled
W1-1	Web service	The service is implemented as REST service using the TypeScript programming language and the Node.js runtime environment. Each request is handled as HTTP request and each response is sent as HTTP response for the corresponding request. Therefore, all requirements for a web service are fulfilled.	Yes

Continued on next page

Table 4.2: World Generation requirement evaluation (Continued)

№	Title	Evaluation	Fulfilled
W1-2	Standalone	The service only depends on the external data sources. The generated track archive follows the SuperTuxKart requirements for tracks and contains all necessary track constituents. For 3D data the SPM file format is used. Therefore, there is no need or connection to any SuperTuxKart tooling like its Blender integration or a SuperTuxKart installation itself.	Yes
W1-3	Non-interactive	Only the input parameters are considered. There is not option to interrupt or manipulate the execution after the initial start.	Yes
W1-4	Performance	The main performance relevant parts are the gathering of satellite images and the computation of the racetrack path. All satellite images are downloaded in parallel, and the usage of Suuraballe's algorithm allows computing the racetrack path in polytime. Several random samples with the number of tiles between 30 and 50 and different positions reveal a real-world runtime of around 30s. The computation is therefore within the required limit.	Yes
W2	Input	The input parameters position and area are handed over to the service with each request and considered for processing the request. As there are no further interaction mechanisms the requirement is fulfilled.	Yes
W3-1	Output	The response to a World Generation request is a ZIP archive containing the SuperTuxKart track with all required constituents. The archive can then be unzipped and loaded with SuperTuxKart to compete in the races on the generated track.	Yes
W3-2	World	The resulting track contains a 3D model of the real-life in the SPM format based on satellite images and OSM data.	Yes
W3-3	World size	The input data is restricted to the selected area and therefore also the generated in-game world.	Yes
W3-4	Textures	The basemap is texturized using real-world satellite images. Buildings are texturized using public domain image data following the SuperTuxKart comic art style. Objects and bonus items use already texturized predefined models.	Yes
W3-5	Racetrack path	Fulfilled by using Suuraballe's algorithm.	Yes

Continued on next page

Table 4.2: World Generation requirement evaluation (Continued)

№	Title	Evaluation	Fulfilled
W3-6	Path length	Fulfilled by using Suuraballe's algorithm.	Yes
W3-7	Path intersections	Fulfilled by using Suuraballe's algorithm.	Yes
W3-8	Path curvature	Fulfilled by using Suuraballe's algorithm.	Yes
W3-9	Sight	The world model is cropped to the region around the racetrack. This allows the world to be as small as possible but still containing all visible and therefore relevant elements.	Yes
W3-10	Objects	Suitable objects within the OSM data are mapped to their SuperTuxKart counterpart. This is demonstrated with trees.	Yes
W3-11	Bonus items	Bonus items are distributed in a reasonable distance on a straight line orthogonal to the racetrack.	Yes
W3-12	Level of detail	Based on the available data the basemap is texturized using real-world satellite images, building models are generated using real-world OSM data, and races take place along real-world streets, which leads to a realistic representation of the real-life. Building textures, objects, and bonus items match the SuperTuxKart art style and gameplay.	Yes
W4	Real-world data	The service uses publicly available OSM data for road and object data and accesses map tile services for satellite images.	Yes
W5-1	Portability	Fulfilled by the usage of TypeScript and Node.js and only platform-independent dependencies.	Yes
W5-2	Extensibility	Further data sources can be added and encapsulated as classes. The results are added to the appropriate files. It is therefore not necessary to alter existing coding for the other data sources.	Yes
W5-3	Scalability	The service is stateless. Therefore, it can be easily scaled.	Yes

Table 4.2 represents the evaluation and fulfillment of each requirement. Overall, all requirements set for the prototype are satisfied. Therefore, the prototype can be used as foundation for further project and to extend functionality.

Section 4.4.3 shows some results of the World Generation. Based on the figures 4.11 to 4.15, it can be estimated whether the World Generation has achieved its objectives. The success can thereby be measured by realism and gameplay. All figures show a playable SuperTuxKart track. Basic functionality like lap counting and the integration of a minimap for AI routing are implemented.

Therefore, the functional gameplay requirements are fulfilled. Further gameplay improvements are realized by placing bonus items. The largest reduction of gameplay is the missing highlighting of the racetrack path, but this would also reduce realism. Therefore, it is important for later projects to find a compromise between the creation of a distinguishable racetrack path and realism. Regarding the realism, all figures show a replication of the real world. Using satellite images as basemap improves the recognition factor. As the buildings only use public domain texture, they cannot be recognized. Also, the resolution of the satellite images and the visibility of shadows on the satellite images reduce recognizability. Therefore, the objective to create a representation of the real-life as realistic as possible is fulfilled but can be improve using better data. In conclusion, the World Generation utilizes the full capacity of the data. The generated racetrack is playable and rather realistic. Drawbacks in gameplay and realism are based on missing data or unavailable gameplay elements like elevation. This can be mitigated by the usage of sophisticated algorithms or advanced data sources in future projects. In general, this leads to the limitations out of scope.

There are two noticeable limitations going beyond the scope of the prototype. The first limitation is the availability of data, which can be used to create the in-game world. There is no publicly available database for 3D models of buildings, which would allow the creation of easily recognizable city surroundings. Instead, the prototype is limited to cuboids based on the shapes of the buildings. The issue of also not available building textures is mitigated by the usage of public domain building images or image sections as textures. Also, available OSM data may be incomplete or incorrect. E.g., building height information is often missing. This may lead to unrecognizable locations as buildings may appear different than in real-life. Also, satellite images are not available for all regions and the resolution of the available images is rather limited. This may also leads to reduced realism. The second limitation is the orientation within a race. As the generated model of the real-life is similar to the real-world surrounding there are few landmarks to accentuate the racetrack path. The prohibition of intersections of the racetrack, the generation of the minimap data, and the placement of bonus items tries to soften this issue, but orientation is sometimes still hard. Insertion of additional gameplay elements that helps to identify the racetrack helps to mitigate the orientation problem but also reduces realism.

4.6 Outlook

Whereas the implemented World Generation application fulfills all requirements it can still be improved. Major improvements can be done by using advanced data sources with detailed information about the real-life. Using a database of 3D models of buildings and important monuments would greatly improve realism, but these data sources are usually not freely available.

The in-game world can be improved by considering terrain information to include height information for the basemap. Buildings can then be placed using their real-world elevation data. The quads would have to be enriched with the height as z coordinate as well. The satellite image textures would have to be stretched along the basemap polygon.

The sky is currently unicolor. To increase realism the sky can be texturized using real-world images of the sky around the car.

The object mapping is demonstrated using trees. Further objects like benches can also be mapped to their in-game counterpart.

To improve gameplay, additional bonus items of different kinds like nitro for a speed boost could be distributed within the in-game world. Additional signs or marks would ease orientation. Using a model of the actual car as kart would increase realism and enables further integration opportunities for SuperTuxKart in connected cars.

The realism of buildings can be improved by adding a roof to the building models. This can be done by adding a slanted roof with a fix angle in a random direction to small buildings. The flat roofs for tall buildings remain unchanged.

5 REST API

The second prototype is the extension of SuperTuxKart with a REST API. Using the REST API, it is possible to extract data from and inject data into SuperTuxKart while the game is running. Therefore, the prototype realizes both parts, injection and extraction, of the vision for video games in connected cars. To provide access to SuperTuxKart following the REST architectural style, the constituents of SuperTuxKart are mapped to resources, which are accessible through HTTP. Gather and manipulation operations for a resource are thereby initiated using the appropriate HTTP verbs as define in section 2.5.1. To ease usage the OpenAPI documentation of the entire REST API is located in the REST API source code directory.

5.1 State of the Art

SuperTuxKart offers a variety of resources, which are active during a game. Also predefined resources and resource models are part of the game installation. The current possibility to dynamically adapt SuperTuxKart is scripting, which is limited to a predefined set of functionalities.

5.1.1 SuperTuxKart Resources

The fundamental resources of SuperTuxKart are tracks, karts, and objects. These are placed and used according to the SuperTuxKart guidelines for gameplay [Sup19]. The track, player, and AI karts are chosen during race setup. Objects are pre-defined 3D models and part of the track. The track objects are placed upfront by the track designer using 3D modeling tools [Sup16].

SuperTuxKart supports additional karts and tracks as add-ons [Sup21c]. All add-ons are installed into a special add-on directory and loaded with the game or while the game is running [Sup20]. Objects as parts of tracks cannot be installed separately and are instantiated and manipulated through the game logic.

Internally SuperTuxKart is implemented in C++ and resource handling is implemented in an object-oriented manner without platform specific dependencies. For storage, resources are serialized using XML in a custom SuperTuxKart schema. Both ensures that SuperTuxKart and SuperTuxKart resources can be used on multiple platforms.

SuperTuxKart allows playing over the network. Therefore, special status messages are sent to synchronize the progress of the players. For the status messages parts of the custom XML schema are reused, but its purpose is limited to the exchange of information about movable objects [Sup21d].

5.1.2 SuperTuxKart Scripting

AngelScript is an open-source, multi-platform library offering to extend applications by external scripts written in an own AngelScript scripting language. The AngelScript scripting language is based on the C++ syntax and data types but does not rely on manual memory management [Jön21]. SuperTuxKart allows AngelScript scripts to be located in the file "scripting.as" inside the track directory. During the race special functions are offered that can be called from the AngelScript coding to allow the modification of in-game state, e.g., to enable or disable an object on the track. For usage inside AngelScript, the internal data structures of SuperTuxKart are mapped to special AngelScript data structures. Also, it is possible to write event callbacks in AngelScript, which are called when the event conditions are satisfied. Callback allow dynamic reactions to special conditions, e.g., the contact of two karts. Scripting in SuperTuxKart is limited to a predefined set of functions and callbacks. Additional functions, callbacks, or libraries cannot be used. Using the scripting functionality, SuperTuxKart tracks can dynamically adapt to changing circumstances, which enables further gameplay opportunities like special game modes with custom victory conditions. Therefore, scripting in contrast to add-ons can be used to dynamically add objects [Sup17].

5.2 Requirements

The REST API for SuperTuxKart is intended as an extension to SuperTuxKart, and its source code written in C++. Therefore, it is started and terminated with SuperTuxKart to be used during the whole time the application is running. It offers additional functionality for SuperTuxKart to access and manipulate the current state of the game through an API following the well-known REST mechanisms. This additional functionality allows the access to the game without knowledge about the internal structure. Also, previous race results are stored and can be accessed for later analysis. Input parsing and the handling of the HTTP requests can be done in parallel, but to extract and maintain consistent data, the game may have to be halted and continued after all data is collected or manipulated. To not interrupt the player, the game can be halted between the computation of two frames. For a common 60Hz panel this results in around 16ms, which can be used to extract the data. Minor delays, e.g., if one frame is skipped, do not effect gameplay and are therefore acceptable. The requirements for the REST API are defined in table 5.1.

Table 5.1: REST API requirements

№	Title	Definition
R1-1	Technical foundation	The REST API shall be based on SuperTuxKart respectively its code.
R1-2	Seamless integration	The REST API shall be started with the launch and terminated with the exit of SuperTuxKart.

Continued on next page

Table 5.1: REST API requirements (Continued)

№	Title	Definition
R1-3	Multi-platform	SuperTuxKart including the REST API shall be running on the Linux, macOS, and Windows operation systems.
R2-1	Domain Model	There shall be a domain model created, which covers the major parts of the game logic.
R2-2	Identifier	Each resource of the domain model shall have its own unique identifier.
R3-1	Data exchange	The project shall offer the possibilities to exchange data in both directions, i.e., to extract or manipulate the current state of the game as an API using REST over HTTP.
R3-2	Paths	Each resource of the domain model shall be mapped to a unique endpoint using the resource's name as path.
R3-3	Endpoints	The host of the endpoints shall be the same host that runs the game.
R3-4	Path parameters	Individual resources shall be addressable using path parameters.
R3-5	Race resources	For all resources that are only available while a race is active, its endpoints shall be started when the race starts and terminated when the race is finished.
R3-6	Race resource paths	The endpoints for the resource defined in R3-5 shall reference the current race in their path.
R3-7	Read	The unabridged state of each resource shall be accessible via HTTP requests sent to the resource's endpoint using the HTTP verb "GET". The result of the query shall be the current state of the resource and the status code "200 OK" if the resource exists.
R3-8	Update	If a resource can be manipulated in the game logic, this manipulation functionality shall be accessible via requests sent to the resource's endpoint using the HTTP verb "POST". The result of the query shall be the current state of the resource and the status code "200 OK" if the resource exists.
R3-9	Create	For alterable list resources there shall be the option to create new resources via HTTP requests sent to the resource's endpoint using the HTTP verb "PUT". The result of the query shall be the newly created resource and the status code "201 Created".

Continued on next page

Table 5.1: REST API requirements (Continued)

№	Title	Definition
R3-10	Delete	If a resource can be deleted without putting the application into an inconsistent state, it shall be possible to delete the resource via a request sent to the resource's endpoint using the HTTP verb "DELETE" and the resource's unique identifier. The result of the query shall be the status code "204 No Content" and no further data.
R3-11	Consistency	After each request the application shall remain in a consistent state.
R3-12	Not found	If a requested resource does not exist, the API shall return the status code "404 Not Found" and no further data.
R4	Usability	The REST API shall be usable without knowledge of the underlying implementation and technologies by using the REST API's documentation.
R5	Extensibility	The list of resources as well as their constituents shall be extensible without changing the existing functionality.
R6-1	Input format	The application shall be able to process JSON input.
R6-2	Pre-built resources	The application shall be able to process pre-built resources packed as ZIP archive.
R6-3	Output format	The application shall be able to produce output in the JSON format.
R7-1	Errors	If any error occurs during any operation of the REST API, the application shall return the status code "500 Internal Server Error".
R7-2	Resilience	If any error occurs because of the REST API, the application shall remain functional, i.e., the application is in a consistent state and the user can continue playing without interruption.
R7-3	Input validation	If the input contains any syntactical or semantical errors, the application shall return the status code "400 Bad Request" and the reason why the request cannot be processed.
R7-4	Testability	The request handling, i.e., parsing and forwarding of the request to the game logic, shall be automatically testable.
R8-1	Persistence	Newly created resources shall be stored persistently.
R8-2	Previous races	Previous race results shall be accessible using the race's unique identifier.
R9	Performance	Each query shall not affect the user's match, i.e, each request shall pause the game for at most 16ms on average. Minor delays are acceptable as they do not effect gameplay.

5.3 Design of Solution

The main objective of the *REST API* is the exchange of information, i.e., the internal state of the game, with any other application or any user. Therefore, the internal state of the game is mapped to a set of endpoints, which then can be used consistently and without the need to know the internal structure of the game. This *REST API* can be used to adapt the game to match certain real-world conditions or to trigger actions to establish the in-game conditions in real-life.

The access to the game state is encapsulated as a service. The service is accessible using REST and HTTP. Therefore, the domain model for SuperTuxKart is extracted from the SuperTuxKart class hierarchy containing the most relevant subset of all SuperTuxKart functionality and data. The REST API for SuperTuxKart maps the constituents of the domain model to resources and offers endpoints to access the resources.

Figure 5.1: Vision and REST API correlation

Figure 5.1 shows the REST API in the context of the vision for advanced integration of video games into connected cars as presented in chapter 3. The REST API is thereby used as middleware in-between the real-life and the video game to enable the data exchange between the video game and the real-life as expected by requirement R3-1. Data injection from the real-life into the video game is send from the real-life to the REST API and then forwarded to the game. Data extraction is performed the other way around by collecting the requested data within the REST API and send the complete data to the connected car. This abstraction as middleware fulfills requirement R4 as it eases data exchange as the internals of the video game are hidden and only the easier communication patterns of the REST API need to be known.

5.3.1 Integration into the SuperTuxKart Lifecycle

Figure 5.2 presents the SuperTuxKart lifecycle with the start and termination of the REST API handlers as UML activity diagram according to requirement R1-2. The REST API handlers are thereby divided into game handlers, which are active the whole-time the game is running, and race handlers, which are only active during a race. Activities related to the game handlers are marked in green. Activities related to the race handlers are marked in orange.

The SuperTuxKart lifecycle begins with the launch of SuperTuxKart and the start of the game handlers. The user interface shows the game menu, which allows the player to select and start the race. After the user selected the race, the race is initialized. When the in-game world is ready,

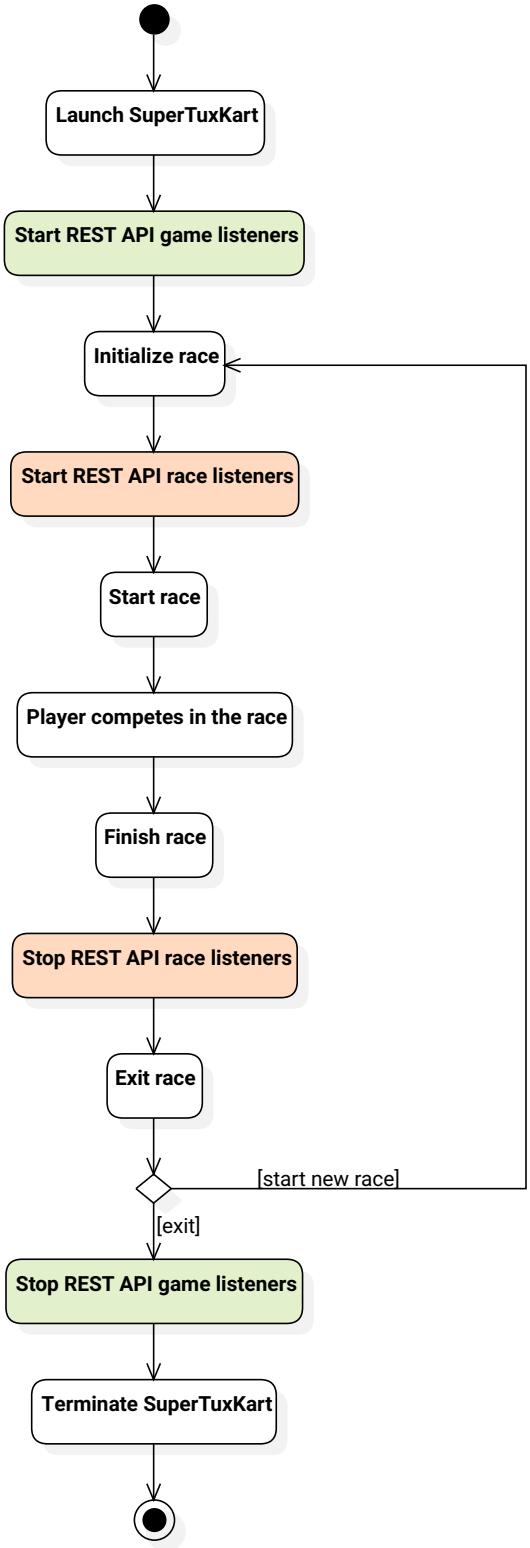

Figure 5.2: SuperTuxKart lifecycle with REST handlers as UML activity diagram

the race listeners of the REST API are started. The race starts as soon as the in-game world as well as the race listeners are ready. The next step is to wait for the player to compete in the race until all karts have reached the finish line or the race is aborted because the player wants to return to the game menu. When the race is over, the race listeners are stopped and the user interface returns to the game menu. In the game menu the player can either start a new race or exit the game. If the player chooses to start a new race, the race and therefore the race handlers are initialized again, and the player can compete in the new race. If the player wants to exit the game, the game listeners and SuperTuxKart itself are terminated. The start and termination of the race handlers in correspondence to the race start and termination fulfills requirement R3-5.

5.3.2 Domain Model

Figure 5.3 shows the most relevant parts of the domain model of SuperTuxKart as UML class diagram as required by R2-1. Attributes are thereby omitted for the sake of briefness. The coloring is in correspondence to the SuperTuxKart lifecycle shown in figure 5.2. The classes that are accessible while SuperTuxKart is running are marked in green, and the classes that are only accessible while a race is running are marked in orange. The coloring is thereby also consistent with the object composition. All classes that are part of a `Race` are marked in orange, and all other classes, which are independent of a race, are marked in green. If a race does not exist, also no orange marked objects exist.

On top there is the game `SuperTuxKart` itself. It is the entry point of the application, and if it does not exist no other object exists as well. The installed karts are represented by the `KartModel`, and the installed tracks are represented by the `TrackModel`. Sound effects are accessible through the `SfxLibrary`. The individual available sound effects are instances of `SfxLibraryEntry`. The music is stored in the `MusicLibrary`, and the individual music tracks are instances of `Music`.

The `CurrentRace` is used to indicate whether there currently is a race and which status the current race has, e.g., started or finished. The races are encapsulated in the class `Race`. Each race has `Karts` and a `Track`. Each `Kart` is based on a `KartModel`. None or a single `Music` track is played in loop while the race is running. While there is only one currently active `Race`, the results of the previous races can still be accessed. Therefore, the races of `SuperTuxKart` are modeled as a list.

The `Track` is linked to a `TrackModel`, which contains the in-game world and defines how the track constituents have to be initialized. As each `TrackModel` can be used in arbitrary `Tracks` the relationship from the `TrackModel` to the `Track` is modeled with the multiplicity of `0..*`. The track constituents are modeled as parts of the `Track`. The `Sfx` controls whether sound effects are played and manages the sound effect volume. All sound effects of the current race are modeled as `SfxSound`, which is linked to a `SfxLibraryEntry`. The checklines are represented as instances of `Checkline`. The quads are `Quads`, which also contain links to the following quads in the race graph. Therefore, it is sufficient to have the `Quad` class to generate the race graph and a potential `Graph` class can be omitted. It is important that there is at least one `Checkline` to enable lap counting, and at least one `Quad` to form a racetrack path. The in-game objects, e.g., trees, are instances of the class `Object`, which can be refined to e.g., lights or particle emitters. The materials and therefore textures are handled as `Materials`. Some materials are part of the SuperTuxKart installation, but as these

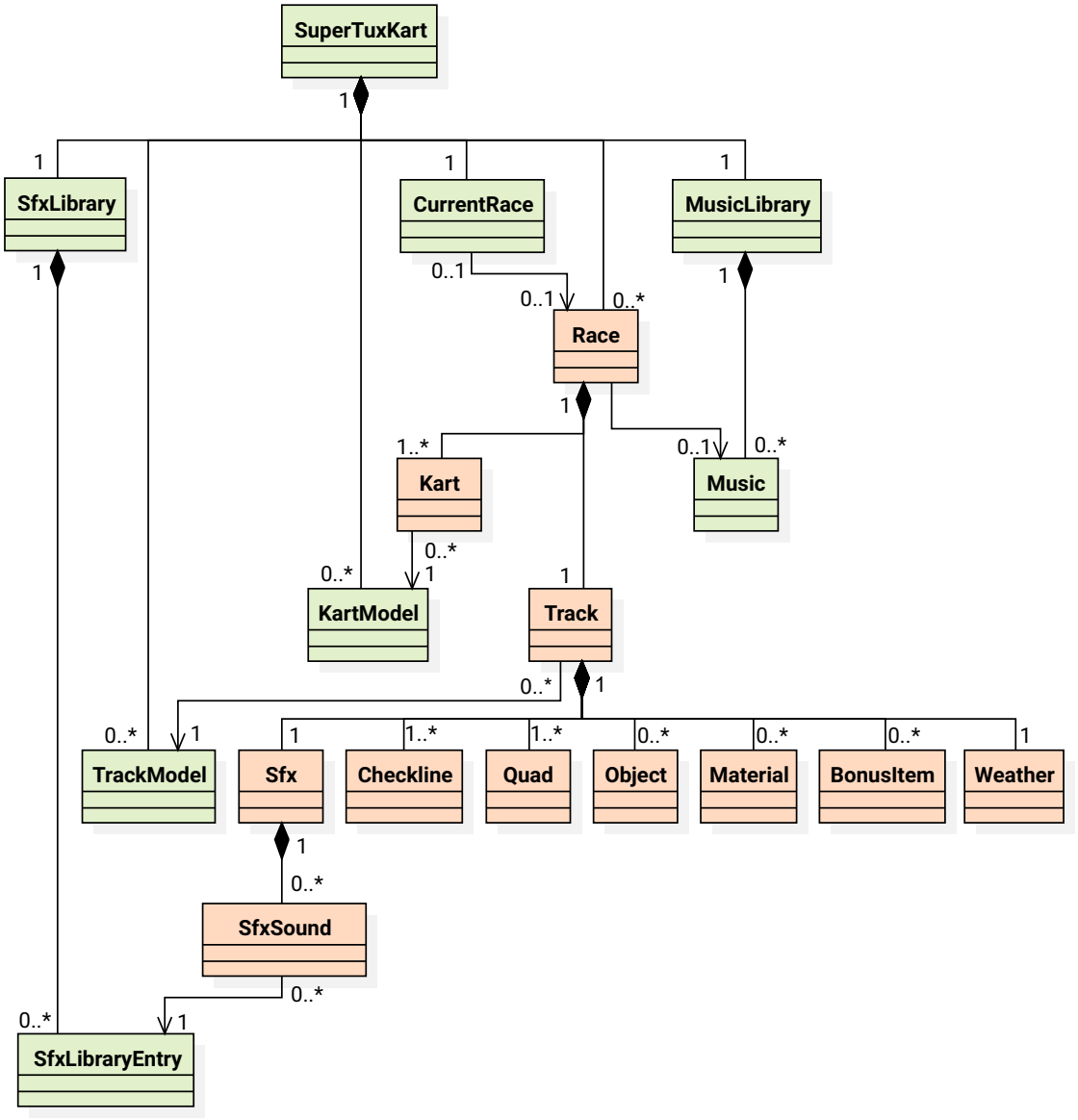

Figure 5.3: SuperTuxKart domain model as UML class diagram

preinstalled materials are merged with the track materials during the instantiation, the materials are handled as part of the `Race`. Bonus items like bonus boxes or nitro cylinders are represented as instances of the class `BonusItem`. The weather of the track is maintained in the class `Weather`.

5.3.3 Endpoints

To allow access to SuperTuxKart in a resource-oriented manner required for the REST architectural style, the available resources of SuperTuxKart need to be specified. Therefore, the domain model of figure 5.3 is used as starting point to map the internal structure of SuperTuxKart to accessible resources. Each class of the domain model can be seen as a resource. To access the resource an

Lifecycle	Resource	Endpoint
Game	Current race	`/races/{raceId}`
	Kart models	`/karts/{kartId}`
	Music library	`/music/{musicId}`
	SFX library	`/sfx/{sfxId}`
	Track models	`/tracks/{trackId}`
Race	Bonus items	`/races/{raceId}/items/{itemId}`
	Checkline	`/races/{raceId}/checklines/{checklineId}`
	Karts	`/races/{raceId}/karts/{kartId}`
	Materials	`/races/{raceId}/materials/{materialId}`
	Music	`/races/{raceId}/music`
	Objects	`/races/{raceId}/objects/{objectId}`
	Quads	`/races/{raceId}/quads/{quadId}`
	SFX	`/races/{raceId}/sfx/{sfxId}`
	Weather	`/races/{raceId}/weather`

Table 5.2: REST API mapping of resources to endpoints

endpoint for the resource is required. Table 5.2 shows the mapping of the resources of SuperTuxKart based on the domain model to the endpoints as expected by requirement R3-2. The resources are thereby divided by their existence in the SuperTuxKart lifecycle. The resources in the "Game" rows are available the whole-time the game is running and correspond to the green listeners and resources of figure 5.2 and figure 5.3. The "Race" resources are only existent while a race is running and therefore also the endpoints are only available while a race is active as they represent the resources of the running race. The race resources thereby correspond to the orange listeners and resources of figure 5.2 and figure 5.3.

For list resources the individual resources can be accessed using the resource identifier marked with braces. The addressing of individual resources thereby fulfills requirement R2-2 and R3-4. The list itself can be accessed by omitting the resource identifier and the slash character. E.g., the list of karts is accessible using the endpoint `/karts` and the kart with the identifier `kartId` is accessible using the endpoint `/karts/kartId`.

The races are also modeled as resources. If a new race is started, a new race is instantiated, and a new race identifier is generated. As the response contains the newly created race resource including the new race identifier, the user can continue querying the resource using the race identifier and the available endpoints. The race endpoints for the currently active race are accessible using the race identifier of the current race as described in requirement R3-6. The latest state of the race resources of a previous race is accessible using the endpoint `/races/{raceId}` with the `raceId` of the previous race as described by requirement R8-2. The other race endpoints are only accessible for the current

race. This allows maintaining knowledge about the past and enables the analysis of the player's in-game progress. The `Track` resource is omitted as it is only an intermediate resources, and the data is accessible using the linked resources directly.

The mapping of the SuperTuxKart resources to endpoints enables accessing the internal structure of SuperTuxKart using REST. The endpoints are the interface to the SuperTuxKart resources and therefore the REST API itself.

5.3.4 Concurrency

To not interrupt the player while playing SuperTuxKart it is important to only intercept the game execution if absolutely necessary and to not halt the game for longer than 16ms as described in requirement R9. The first parameter for performance is the usage of concurrency. The request handling of the REST API can be done in parallel to the game computations and therefore does not affect the in-game performance while still enabling performant request processing. It is only necessary to interrupt the SuperTuxKart computation for the pure data injection and extraction operations without the REST request handling overhead, which also keeps the application in a consistent state and returns only self-consistent data.

The second parameter for performance is the extraction and injection of data from and into SuperTuxKart without affecting the gameplay and still maintaining consistency as required by requirement R3-11. To enable performant request handling it is essential to understand how SuperTuxKart computes the game progress and renders the in-game world. The game progress is thereby a linear sequence of frames. The temporal distance between each frame is a timestep. When a frame is rendered, the progress of the in-game world for one timestep is computed. After the progress is computed, the game is halted until the next frame needs to be computed. This intermediate time between the computation of the progress of the in-game world and the rendering of the next frame can be used by the REST API to extract and inject data without any performance impact on the gameplay.

Figure 5.4 shows an UML sequence diagram of an exemplary interaction between the player, SuperTuxKart, the REST API, and the REST API users. In correspondence to figure 5.2 and figure 5.3 the game handlers of the REST API are colored in green, and the race handlers colored in orange. The messages are consecutively numbered in the order of their appearance. The player's interactions are shown on the left `player` lifeline. The REST API users's interactions are shown on the right `api users` lifeline. SuperTuxKart is represented by the `supertuxkart` lifeline. The game handlers are shown on the `game` lifeline, and the race handlers are shown on the `race` lifeline. The requests of each lifeline can occur and be handled in parallel. Within the SuperTuxKart application `supertuxkart` is the first thread, and the REST API as `game` and `race` is the second thread.

Consistent to figure 5.2 the first two messages are used to launch SuperTuxKart and start the game listeners. Messages 3 to 6 show a data extraction request from the `api users` to `supertuxkart` with the REST API in the form of the game listeners. Message 4 has to be answered by `supertuxkart` and has therefore a time constraint to not take longer than 16ms. To maintain consistency, it is performed in the intermediate time between two frames. As visible by the absence of the `race` lifeline at this time it would not be possible to query the race endpoints. Messages 7 to 11 represent the selection and start of a race and therefore the `race` listeners. Messages 12 to 15 show a data injection request to the `race` endpoints. As for message 4 message 13 is temporally restricted to at

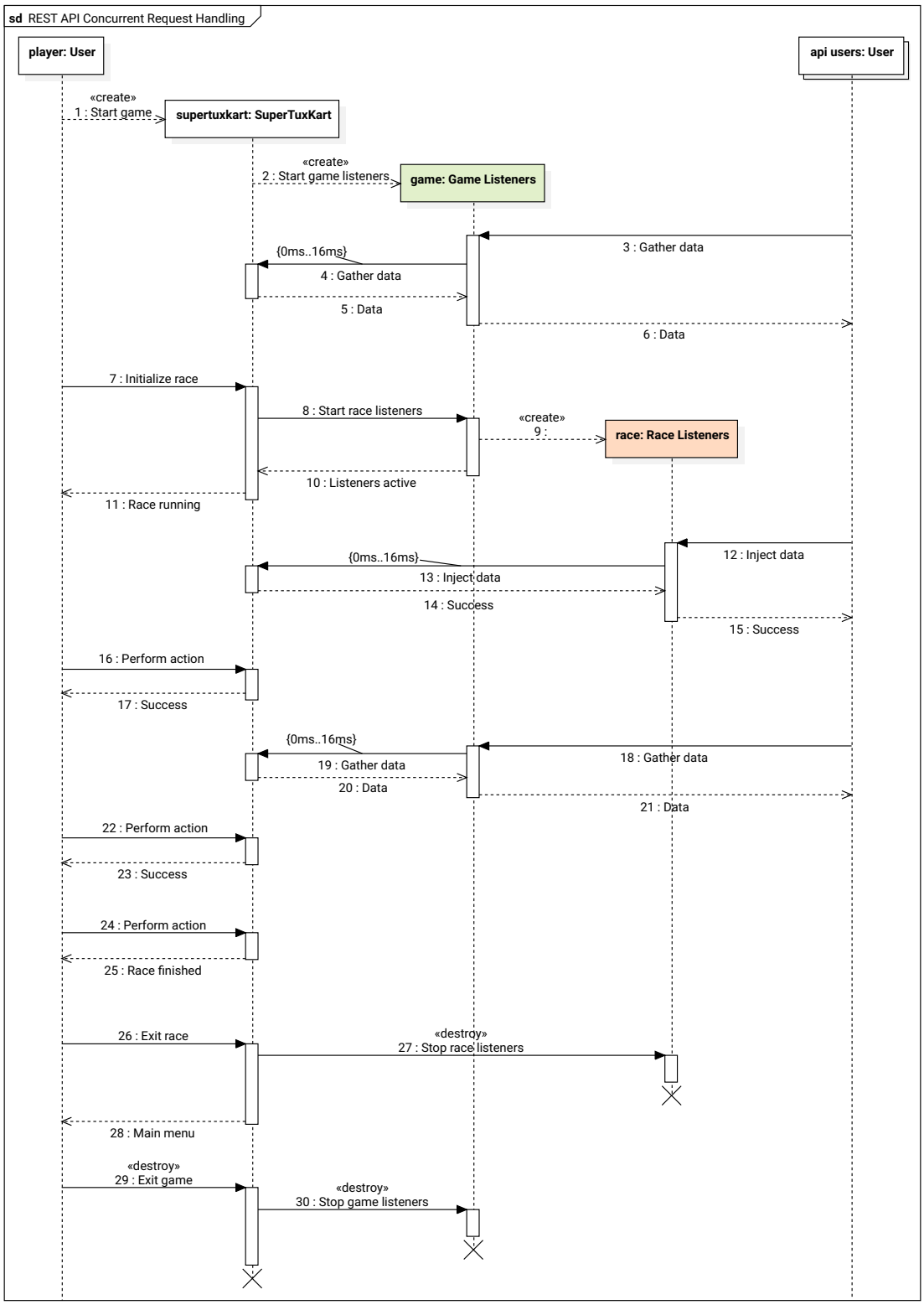

Figure 5.4: UML sequence diagram of REST API concurrent request handling

most 16ms and executed between the computation of two frames. Messages 16 and 17 symbolize the interaction of the `player` with `supertuxkart` to compete in the race. As the `game` endpoints are also accessible while the race is running, messages 18 to 21 show a data extraction request from the `api users` to the `game` endpoints. As the messages 4 and 13, message 19 is temporally restricted. Messages 22 and 23 as well as the messages 24 and 25 represent an interaction of the `player` with `supertuxkart` similar to the messages 16 and 17. In message 26 the `player` decides to quit the race, which induces the termination of the `race` handlers in message 27. As observable in message 28, the user interface returns to the game menu. In the game menu the `player` decides to quit the game in message 29, which induces the termination of `supertuxkart` and the `game` handlers.

5.4 Realization of Solution

The prototype is realized by extending the SuperTuxKart C++ code [Sup21d]. To allow REST request using HTTP the library cpp-httplib [yhi21] is used to encapsulate HTTP. To process JSON the library RapidJSON [TY16] is used. To test the handlers the C++ unit test framework GoogleTest [Goo21] is used.

5.4.1 Implementation Architecture

The implementation is based on three main components data exchange, handler, and endpoint. The integration of all three components into SuperTuxKart enables the fulfillment of the requirements.

- **Data Exchange**: The data exchange interface is the foundation for the integration into SuperTuxKart. The classes implementing the data exchange interface offer getters and setters for resource attributes. The implementation of these getters and setters access the internal structure of SuperTuxKart, and therefore encapsulate the internal structure SuperTuxKart within the REST API. In other words, an implementation of the data exchange interface represents the internal structure of a resource of SuperTuxKart.
- **Handler**: The handler class is the superclass for all request handlers. It provides methods for all HTTP operations, and encapsulates the request handling. The parameters of the HTTP operations are the request body, and if applicable, the identifier of the resource, which is extracted as path parameter of the request path. The parsing of the request body is done within the handler. To ease development, the handler superclass offers a default implementation for all request types. Therefore, only the intended operations need to be overloaded. The default implementation just returns the status code “404 Not Found” and an empty body. The result of an HTTP operation implemented by a request handler is the response body as well as the status code as specified in R3-7, R3-8, R3-9, or R3-10.
- **Endpoint**: The endpoint coordinates the matching of a request path to a handler class. It is implemented to use a list of available request handlers and therefore resources, and their path including path parameters. The endpoint is responsible for extracting the values of the path parameters if available. If a handler path matches the request path, the handler’s logic for the HTTP operation is executed.

To further elaborate the relationship between the data exchange classes, the handler classes, and the endpoints, it is important to understand the interaction between the components. Therefore, the call sequence for a request to the REST API is examined.

1. The user sends a HTTP request to an endpoint.
2. The endpoint corresponding to a path that matches the request's path is identified.
3. The endpoint calls the method of the HTTP operations on its handler. The parameters are the body of the request, and if applicable, the resource's identifier.
4. The handler parses the request.
5. The handler calls the appropriate methods of the data exchange to satisfy the user's request.
6. When the initiated modifications have taken place, the data exchange model returns control.
7. The handler generates the result body.
8. The handler returns the result body with the specified status code as response to the user.

5.4.2 Data Exchange

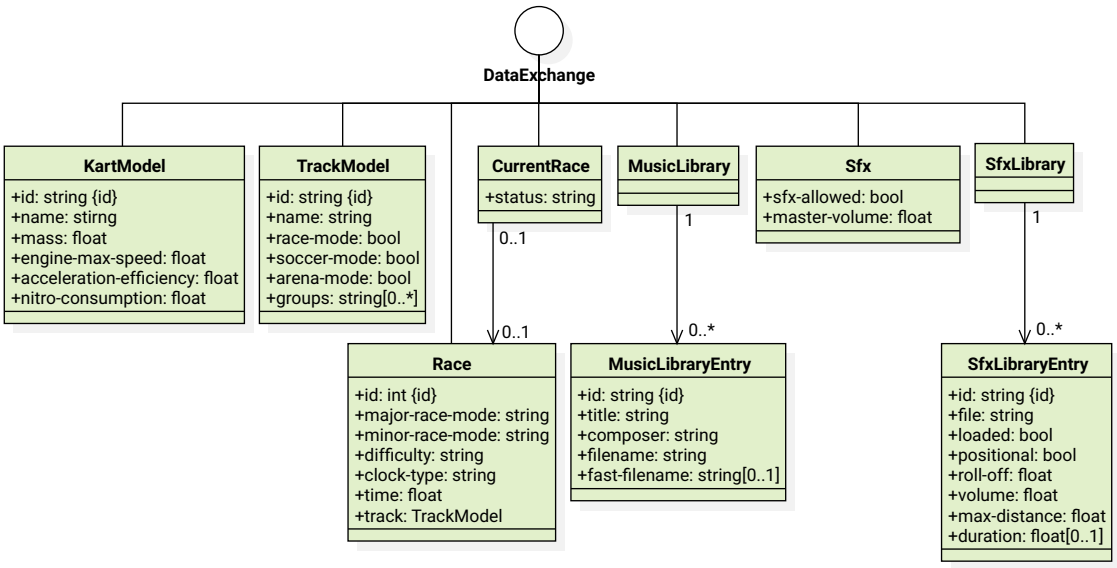

Figure 5.5: UML class diagram of game data exchange classes

Figure 5.5 presents the UML class diagram of the data exchange classes of all resources that are available independently of an active race. The classes are colored green in correspondence to the coloring of section 5.3. The class attributes conform to the attributes of the resources of the REST API without simplification. The implementation thereby does not allow access to member variables but offers getters and setters that cover the underlying SuperTuxKart integration. These getters and setters are omitted for the sake of briefness. Whereas all attributes are readable, only a specified subset is modifiable. All classes implementing the `DataExchange` interface correspond to

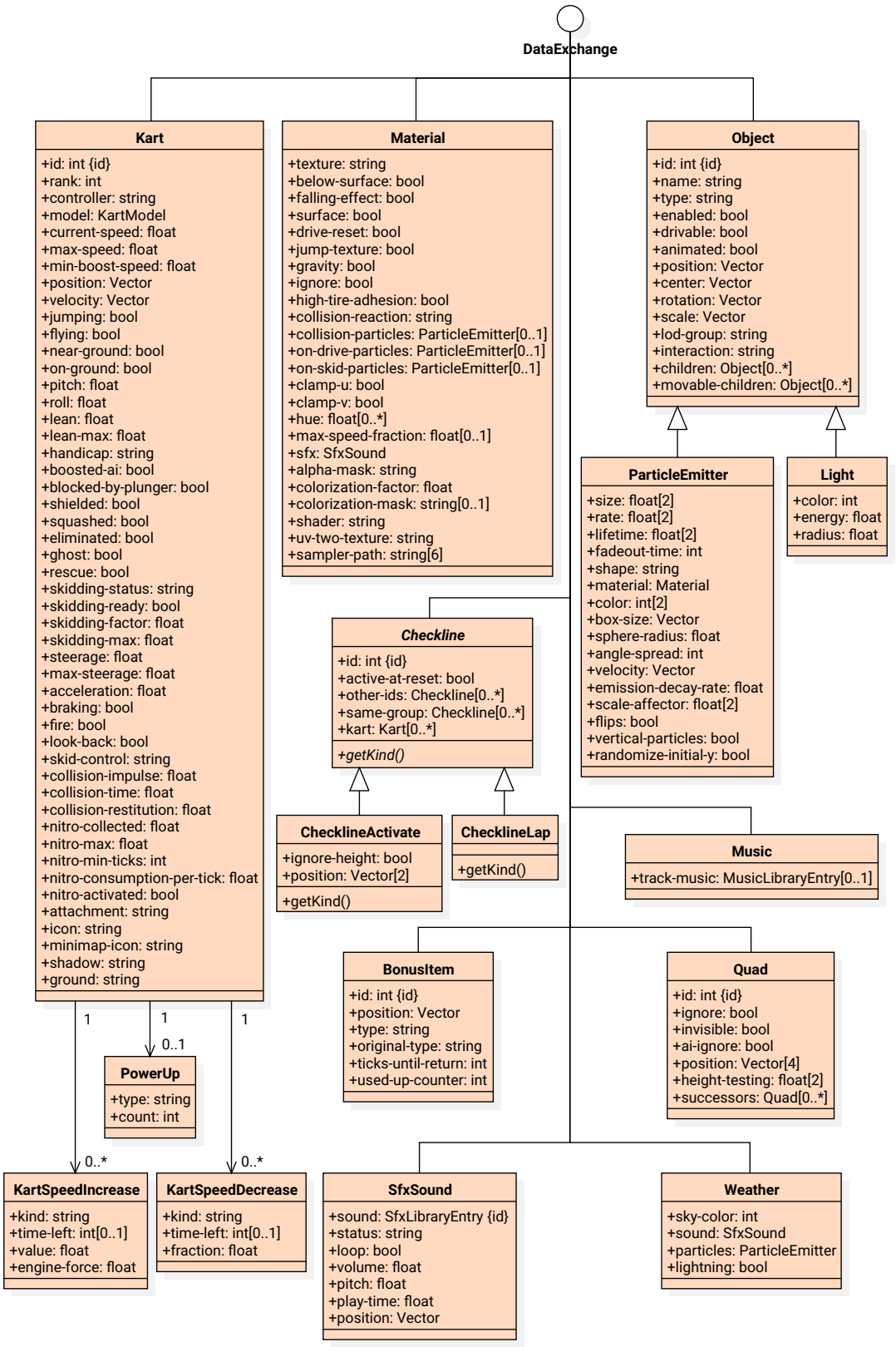

Figure 5.6: UML class diagram of race data exchange classes

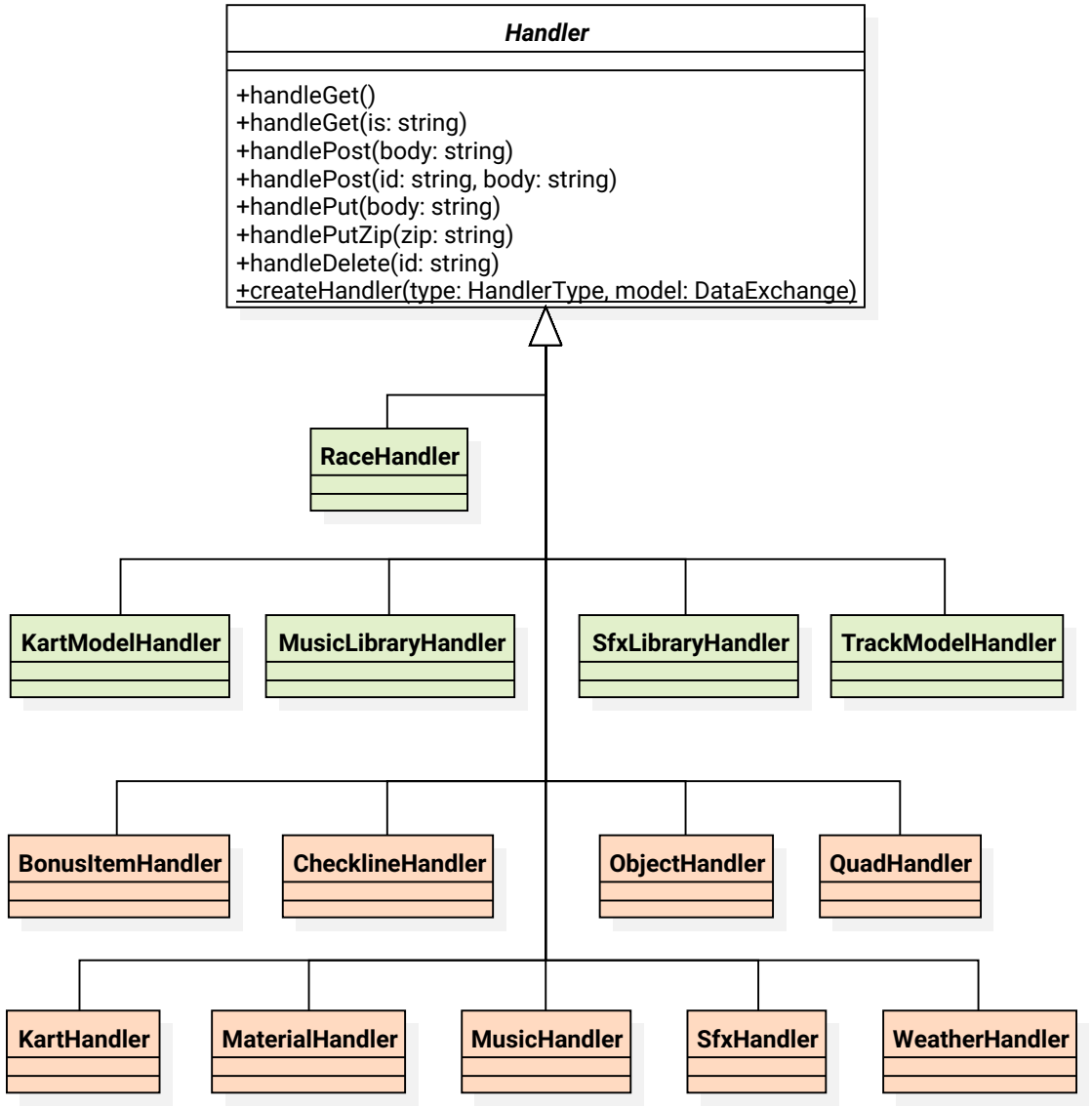

Figure 5.7: UML implementation class diagram of REST API handlers

a SuperTuxKart resource and can therefore be directly accessed using an endpoint. The directed associations result in member attributes of the data exchange classes in the response of the REST API.

Figure 5.6 contains the UML class diagram of the data exchange classes of the resources that are only available during a race. The classes are colored orange in correspondence to the coloring of section 5.3. Like in figure 5.5 getters and setters are omitted for the sake of briefness. Also, the set of attributes is complete and accessible, but only a subset can be modified to not endanger the application's consistency. Classes implementing the data exchange interface are directly accessible through an endpoint, associated classes as member attributes, and inherited classes just add their attributes to the superclass resource. The data exchange interface is the same as in figure 5.6.

5.4.3 Handler Design

Figure 5.7 presents the design of the handler component. The `Handler` class is the abstract superclass of all request handlers. The HTTP operations are realized as methods of the `Handler` class, and can take the response body, the resource identifier, or both as parameters. The static factory method `createHandler` is substitutional for one factory method for each handler implementation. To create a handler, the factory method of the handler needs to be called with the data exchange model of the resources that is represented by the handler. To encapsulate, the implementation of the handler classes stays hidden in an anonymous namespace and is therefore not accessible without calling the factory method. The coloring of handlers available the whole time the game is running in green, and of handlers only available while a race is active in orange are consistent to the coloring in section 5.3.

Another demand satisfied by the solution architecture is the realization of the testability requested in requirement R7-4. As the request handler processes the request body, typing errors and oversight can easily occur. Unit testing is a useful technique to prevent these kinds of mistakes. As other parts of SuperTuxKart do not support unit testing, the GoogleTest framework is used to maintain the tests. The testing is done by handing over a mock implementation of a data exchange class to a `Handler` class that should be tested. If the expected methods of the mock implementation are called, the implementation of the `Handler` class does not contain rough mistakes. The semantics still need to be checked by the developer.

5.4.4 Endpoint Matching

Endpoint	Regular Expression
/races/{raceId}	^\/races(\/(\d+))?$
/karts/{kartId}	^\/karts(\/(.+))?$
/music/{musicId}	^\/music(\/(.+))?$
/sfx/{sfxId}	^\/sfx(\/(.+))?$
/tracks/{trackId}	^\/tracks(\/(.+))?$
/races/{raceId}/items/{itemId}	^\/races\/(\d+)\/items(\/(\d+))?$
/races/{raceId}/checklines/{checklineId}	^\/races\/(\d+)\/checklines(\/(\d+))?$
/races/{raceId}/karts/{kartId}	^\/races\/(\d+)\/karts(\/(\d+))?$
/races/{raceId}/materials/{materialId}	^\/races\/(\d+)\/materials(\/(\d+))?$
/races/{raceId}/music	^\/races\/(\d+)\/music$
/races/{raceId}/objects/{objectId}	^\/races\/(\d+)\/objects(\/(\d+))?$
/races/{raceId}/quads/{quadId}	^\/races\/(\d+)\/quads(\/(\d+))?$
/races/{raceId}/sfx/{sfxId}	^\/races\/(\d+)\/sfx(\/(\d+))?$
/races/{raceId}/weather	^\/races\/(\d+)\/weather$

Table 5.3: REST API path to endpoint matching

The correlation between endpoints and resources is established by assigning each resource a unique path as defined in table 5.2. To allow path parameters as dynamic parts of a path each path is converted to a regular expression.

The regular expressions of the resource paths are shown in table 5.3. Each path parameter is matched as a capturing group. Numeric parameters are limited to set of numerals (\d). String parameters can use arbitrary characters (.). Each identifier must not be empty (+). To only match whole paths, the start (^) and end ($) anchors are specified for each path.

The matching is done by looping over the list of endpoints and calling the handler of the first matching endpoint. To be able to process a specific regular expression before a more general regular expression that matches the same path, the order of endpoints is maintained and needs to be considered while integrating endpoints. If the requested path does not lead to any matches, the status code “404 Not Found” is returned.

5.4.5 Race Result Persistency

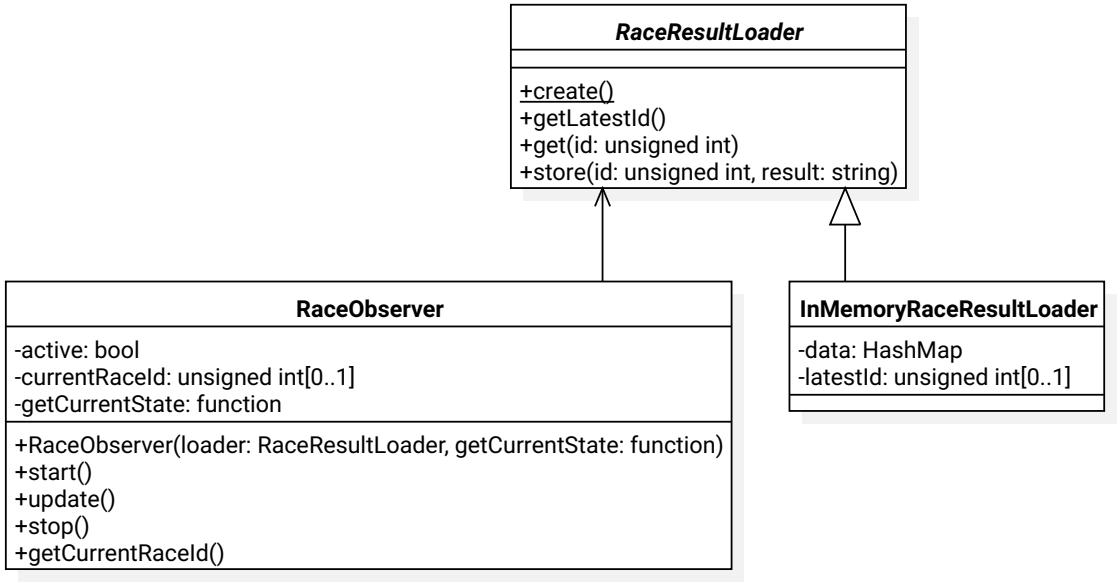

Figure 5.8: UML class diagram of race result persistency

In order to realize requirement R8-2 to allow access to previous race results a persistency mechanism is established. Figure 5.8 shows the UML class diagram of this persistency mechanism.

The `RaceResultLoader` is an abstract interface implemented by the `InMemoryRaceResultLoader`. The `InMemoryRaceResultLoader` stores the previous results in a hash map with the race identifier as key and the serialized result as value. The optional attribute `latestId` of the `InMemoryRaceResultLoader` indicates whether there is a previous result and what the identifier of the latest result is. The static factory method `create` of `RaceResultLoader` returns an instance of the `InMemoryRaceResultLoader`, which allows the `InMemoryRaceResultLoader` to stay hidden in an anonymous namespace. To use other persistency mechanisms like storage in files only the interface `RaceResultLoader` needs to be implemented and linked in the `create` method.

An instance of `RaceObserver` is created each time a new race starts. The maintenance of the `RaceObserver` is integrated into the SuperTuxKart race handling. Each instance of `RaceObserver` receives the same instance of `RaceResultLoader` to allow persistency over several races. When the status of the race changes, `start`, `update`, or `stop` is called, and the current state of the race is stored in the `RaceResultLoader` by calling `getCurrentState` to get the serialized state of the current race.

As SuperTuxKart does not maintain information about previous races, the `RaceObserver` is also used to maintain the identifier of the current race. `getCurrentRaceId` returns whether there is an active race and what the identifier of the current race is. The status if a race is active is maintained using the attribute `active`, which is set by `start` and `stop`.

5.4.6 Concurrency

As the request handling is implemented to run in parallel in a different thread than the SuperTuxKart game logic, concurrency challenges arise. Data races, race conditions, and deadlocks must be prohibited as they might lead to an inconsistent application state.

Considering that only the time between two frames can be used for data injections or extractions, a mechanism is needed that starts execution right after the processing of one frame ends and finishes before the next frame needs to be computed. The timing for execution between two frames is realized using a mutex. While a frame is calculated the mutex is locked and the REST API cannot start processing until the frame computation is finished. Also, the next frame cannot be computed until the REST API established a consistent state. As the locking and unlocking is only performed once a frame, at most one request is processed per frame, and there is a computational break between the frame, the impact on the performance is not measurable.

The second challenge is the detection when an initiated modification takes place. This is necessary to not send the response to the REST API user before the modification has been performed. A conditional variable allows a thread to wait for a specific condition satisfied by another thread without the need for further processing power as e.g., polling requires. Therefore, a conditional variable is introduced waiting on the condition that the modification has taken place. When the condition is satisfied, the modification is included in the current state, and the REST API can send the response based on the modified resource. This mechanism can also be combined with the mutex approach to maintain consistency while the response is computed.

The third challenge is the thread-dependence of some operations. E.g., it is not possible to load a texture for usage with OpenGL on a thread that is not registered as an OpenGL contributor. In order to maintain separation of concerns a data injection mechanism is established, which is synchronized using a conditional variable. The instructions for the SuperTuxKart thread are stored in a variable. The next time the SuperTuxKart thread finishes processing a frame the variable is evaluated, and the instructions are executed. When the thread continues, the condition of the conditional variable is satisfied, and the response is sent to the REST API user. This allows coordination between the thread to perform thread-dependent operations, and still know when the modification has taken place without putting extra load on the system.

5.4.7 ZIP Resources

In order to realize requirement R6-2 it is required that packed resources as ZIP archive can be uploaded in a PUT operation. The extracted resources are then stored in the SuperTuxKart add-on directory. When the extraction is done, the SuperTuxKart functionality to load the resource is called. As the add-on directory is persistent even if SuperTuxKart is restarted, the uploaded resources stay available and are loaded with the start of SuperTuxKart. It is expected that the uploaded ZIP archive contains a single directory with the name of the resource. If a resource with the same type and name as the uploaded resource already exists, the operation is aborted. When an add-on resource is deleted, the corresponding add-on directory is deleted as it would else be loaded again on the next launch of SuperTuxKart. If any error occurs during the extraction or while loading the resource, the operation is rolled back by deleting the newly created directory. This ensures consistency as expected by requirement R3-11. The response is not sent before the resource is loaded successfully or the request failed.

The supported resources for ZIP archive upload are the game resources `KartModel`, `TrackModel`, `MusicLibrary`, and the `SfxLibrary` as they maintain persistent data. The race resources are only temporary, and therefore not eligible for uploading persistent resources. The SuperTuxKart add-on directory is designed to have a designated subdirectory for each resource type. The in-game logic for handling add-on `KartModels` and `TrackModels` is already included in SuperTuxKart and just needs to be called. The logic for the `MusicLibrary` and the `SfxLibrary` needs to be added. This is done by adding a unique add-on directory for music tracks and for sound effects, which is created when the game starts and the directory does not exist yet. The logic for handling music tracks and sound effects is extended to also check the new add-on directories during start-up. When an add-on resource is deleted, the in-game functionality for removal is called to not create memory leaks.

C++ does not offer integrated support for extracting ZIP archives. Also, SuperTuxKart does not include a library for ZIP handling but includes zlib [GA17] instead, which enables deflation of single files within a ZIP archive. As parsing a ZIP archive file according to the ZIP specification [PKW20] is less effort and less error-prone than adding further dependencies, a simple ZIP archive parser is implemented. As the ZIP format is well-known it is only recapitulated briefly. The central directory with metadata like offset and filename about all files of the ZIP archive is located at the end of the ZIP archive file. It is found by reverse iterating through the file. The zlib functionality can then be used to decompress the ZIP file at the specified offset.

5.5 Evaluation of Solution

The main objective of the REST API for SuperTuxKart is to support the manipulation of in-game state via REST and to serve as easy-to-use foundation for further projects. Therefore, it is checked whether the requirements set on the REST API are fulfilled. Furthermore, it is evaluated if the REST API is suitable for future projects on the basis of a representative example. To suit advanced use cases the extensibility for further resources and therefore endpoints but also additional resource attributes are examined.

5.5.1 Requirement Analysis

Table 5.4: REST API requirements evaluation

№	Title	Evaluation	Fulfilled
R1-1	Technical foundation	Fulfilled by extending the SuperTuxKart source code.	Yes
R1-2	Seamless integration	Fulfilled by integrating launch and termination of the REST API handlers into the SuperTuxKart lifecycle as described in figure 5.2.	Yes
R1-3	Multi-platform	The SuperTuxKart source code is extended by additional C++ code and libraries that are also platform-independent. By using the installation instructions of SuperTuxKart and the added libraries, SuperTuxKart with the REST API can be built and used on the specified platforms.	Yes
R2-1	Domain Model	The domain model is located in figure 5.3.	Yes
R2-2	Identifier	By implementing the endpoints of the REST API as described in table 5.2 the individual resources can be accessed using the resource's identifier as path parameter.	Yes
R3-1	Data exchange	As described in figure 5.1 the REST API is designed as middleware between the connected car and the game. The data extraction and injection operations are implemented by using the appropriate HTTP mechanisms.	Yes
R3-2	Paths	The mapping of resources to endpoints can be found in table 5.2.	Yes
R3-3	Endpoints	The REST API is designed as single host extension of the SuperTuxKart source code. Therefore, the REST API operates on the same host as SuperTuxKart itself, and no further hosts are involved.	Yes
R3-4	Path parameters	The addressing schema for the resource identifier of R2-2 is described in table 5.2.	Yes
R3-5	Race resources	The race resources are considered in the domain model of figure 5.3 and the race handlers are only active during a race as described in figure 5.2.	Yes
R3-6	Race resource paths	The race endpoints as specified in table 5.2 require the identifier of the current race as path parameter.	Yes

Continued on next page

Table 5.4: REST API requirements evaluation (Continued)

№	Title	Evaluation	Fulfilled
R3-7	Read	If the requested resource exists, the resources handlers return the unabridged and serialized state of the requested resource as JSON together with the specified status code for “GET” requests.	Yes
R3-8	Update	“POST” requests lead to a modification of the state of SuperTuxKart. After the modification takes place the updated state of the resource is returned together with the specified status code.	Yes
R3-9	Create	New resources are created as SuperTuxKart add-on resources using a “PUT” operation, and stored in the appropriate SuperTuxKart add-on directory. If the resource was successfully created, the specified status code is returned.	Yes
R3-10	Delete	SuperTuxKart add-on resources can be deleted by using a “DELETE” request, which unloads the resources using the SuperTuxKart functionality for resource handling, and deletes the directory of the add-on resources. If the deletion was successful, the specified status code is returned.	Yes
R3-11	Consistency	By processing data extractions and injection between the computation of two frames the response data is always consistent. The internal state during and after data injections is kept consistent by changing the internal state through SuperTuxKart procedures especially designed to change state.	Yes
R3-12	Not found	The resource handlers check the existence of a resource and return the specified status code if nonexistent.	Yes
R4	Usability	The REST API is documented using OpenAPI. The internal structure of SuperTuxKart is hidden behind the resource layout and knowledge about it is therefore not required to know to use the REST API.	Yes
R5	Extensibility	Additional resource handlers and endpoints can be easily added by adding a new class for the handler and registering the endpoint via its path as regular expression. Existing handlers do not have to be changed.	Yes

Continued on next page

Table 5.4: REST API requirements evaluation (Continued)

№	Title	Evaluation	Fulfilled
R6-1	Input format	The REST API requires requests to be in the JSON format, which is processed using RapidJSON.	Yes
R6-2	Pre-built resources	Pre-built resources packed as ZIP archives can be uploaded using PUT operations. These resources are then extracted in the corresponding SuperTuxKart add-on directory and loaded by the appropriate SuperTuxKart functionality.	Yes
R6-3	Output format	All responses are in the JSON format, and built using the RapidJSON functionality for creating JSON files, which ensures syntactically correctness.	Yes
R7-1	Errors	Internal errors are encapsulated as `std::runtime_error`, and converted to the specified status code in the request handling.	Yes
R7-2	Resilience	Errors of the REST API are implemented as exceptions in the thread of the REST API. The thrown exceptions are handled in the request handling. The game thread is not affected by the REST API exceptions.	Yes
R7-3	Input validation	The input is syntactically verified by using the RapidJSON parser, and semantically validated by the request handlers. Errors are encapsulated as `std::invalid_argument` and handled accordingly in the request handling.	Yes
R7-4	Testability	By the class design to encapsulate the request handling and the separation of request handling and request execution the testing of the request handlers is possible. Existing handlers are tested using the GoogleTest unit test framework.	Yes
R8-1	Persistence	Created resources are stored persistently as SuperTuxKart add-on resources in the SuperTuxKart add-on directory.	Yes
R8-2	Previous races	When the status of a race changes, all resources are serialized and stored. These serialized results can then be accesses using the race's identifier.	Yes

Continued on next page

Table 5.4: REST API requirements evaluation (Continued)

№	Title	Evaluation	Fulfilled
R9	Performance	By using multi-threading and the intermediate time between the computation of two frames as described in section 5.3.4 the impact of the request to the REST API are in the magnitude of a few milliseconds. Therefore, no further investigations or improvements are necessary.	Yes

The fulfillment of the requirements set on the REST API is evaluated in table 5.4. Overall all requirements are fulfilled and the REST API can be used as foundation for future projects to fulfill the vision of advanced integration of SuperTuxKart into connected cars.

5.5.2 Exemplary Future Project

The main objective of the REST API is its usage as foundation for other projects. To evaluate the fulfillment of this main objective the usage of the REST API in an exemplary future project is presented. The examined project is the integration of weather data. The endpoint for weather data of the REST API is `/races/{raceId}/weather`. The supported attributes are `sky-color`, `sound`, `particles`, and `lightning`. As weather is only possible if an in-game world exists, the endpoint is only available while a race is running. Both directions of the data exchange model are examined.

The data injection direction is evaluated by injecting weather data into SuperTuxKart. The real-world weather conditions can be determined using sensor data of the connected car or an external weather data provider located in the cloud. To use the weather data within SuperTuxKart it needs to be preprocessed. Precipitation needs to be mapped to particles. Lightning can be extracted from the weather data. Sky color can be determined by the actual sky color, the surrounding light color determined by a sensor, and cloudiness. The ambient mood specified by the light color, landscape, and loudness of the environment can be mapped to an appropriate in-game sound effects. This can be done, e.g., by using artificial intelligence trained with the list of available in-game sound effect. These four values are then sent to the REST API using a `POST` request. The in-game weather is immediately changed, and the response contains the changed weather resource to allow validation of the modification. When the weather in the real-life changes, the data preprocessing and injection is performed again.

The data extraction is used to adapt the real-world surrounding to match the in-game conditions. This can be used to transfer the mood of the in-game track to the car's interior. Using a `GET` request on the weather endpoints reveals the previously mentioned four attributes of the in-game weather resource. The sky-color can be directly mapped to the color of the interior illumination. The lightning information can be used to flash the interior as well as external lights in randomized intervals, if there is lightning in-game. Lightning sound is already present through the in-game sound effects. Particle information can be used to e.g., spray windshield washer fluid on the glass, and activating the windshield wiper. In-game sound effects are already audible but can be further extended, e.g., by engine sounds of the real engine of the car.

This example shows that further sophisticated projects using the REST API as foundation are possible and increase immersion and gaming experience. The REST API eases integration of weather data in the injection as well as in the extraction direction as the SuperTuxKart internals are hidden and data exchange is reduced to the exchange of the weather resource with its four attributes. Limitations are the resource exchange via self-contained messages, which do not support continuous values and data exchange and the supported attributes of the weather resource. Continuous value exchange can be added, e.g., by adding additional exchange mechanisms for continuous values, e.g., web-sockets. Additional weather attributes or even additional resources can be realized by exploiting the REST APIs extension capabilities.

5.5.3 Extensibility

Based on the open-closed principle for application development [Mar96] software should be open for extension but closed for modification. As the resources of the domain model of figure 5.3 only cover the most relevant resources of SuperTuxKart it might be necessary to add additional resources for special use cases. Therefore, it should be possible to add new resources without large integration effort and the need to change existing program code.

Adding a new resource involves three components: a data exchange component, a handler component and an endpoint. The scope of the components is as follows:

- **DataExchange**: The DataExchange component for data handling and integration with SuperTuxKart is a class that implements the data exchange interface. It offers getters and setters for all requested attributes. As this is newly written code no further code needs to be adapted.
- **Handler**: The Handler component is a class that inherits from the handler class. It receives the request body and if applicable, the identifier of the requested resource. Therefore, it needs to implement the body parsing logic. As the handler class already offers default behavior for all request types, only the required operations need to be overwritten. To ease implementation available helper functionality for JSON parsing and ZIP handling can be used. As the data exchange component, the handler component contains only newly written code, and therefore does not require any further program code modifications.
- **Endpoint**: The endpoint matches a resource path to a resource handler. It is therefore the tuple of endpoint path and resource handler. The path matching is implemented by specifying the path as a regular expression and appending it to the list of existing endpoints. The resource handler is implemented by handing over a static factory method of the handler component to the endpoint. The other endpoints are not affected by this endpoint injection and do therefore not subject any modification.

By implementing these three components it is possible to extend the REST API by further resources. The open-closed principle is fulfilled, and extension is easily possible.

Adding further attributes to existing resources is more difficult as it involves a modification operation. The involved components of the resource are the data exchange component, and the handler component. The endpoint is not affected as the resource identity remains unchanged. The data exchange component is extended by adding new getters, setters, and integration logic for the newly

added attributes. The handler component logic needs to handle the new attributes and is therefore modified to support these new attributes in requests as well as in responses. As this is a modification, it is more difficult to integrate, but as the handler component mainly comprises parsing logic, it is still rather simple to add additional attributes to existing resources. Therefore, the open-closed principle for being open for extensions is fulfilled.

As the open-closed principle states, the application is closed for modification. It is not intended to change existing behavior and is therefore much more complex than an extension as the existing logic needs to be analyzed, understood, and altered correctly.

5.6 Outlook

Considering the vision for advanced integration of video games into connected cars and the data exchange model between the real-life and the video game the REST API is the connector that connects the real-life with the video game and enables the data exchange. The REST API itself does not realize the vision but serves as foundation for future projects in the context of advanced integration of SuperTuxKart into connected cars. Therefore, it is important to not only consider the improvements of the REST API itself but also the projects that are possible using the REST API as foundation.

The REST API can be improved using its extensibility mechanisms. Additional endpoints for additional resources as well as the mapping of additional attributes within existing resources are possible. Also, additional operations on existing or new resources can be added. Further resource types and interaction mechanisms, e.g., to control steerage by transmitting data using a WebSocket connection, are feasible.

The amount of integration possibilities of SuperTuxKart into connected cars is almost infinite. The REST API thereby supports data extraction projects, e.g., to control the interior or environment to match the in-game conditions, as well as data injection projects like the adaption of the in-game weather to match the real-world surrounding. Also, bidirectional data exchange, e.g., to realize augmented reality projects, is feasible.

The REST API can also be used to integrate the generated tracks by the World Generation application. Therefore, the World Generation application generates an in-game track. The generated track is then uploaded to the *track* endpoint and a new race on the uploaded track is started using the *race* endpoint. This enables seamless integration of multiple advanced integration approaches for SuperTuxKart and gives an impression how the realized vision may look like.

6 Conclusion

The vision of advanced integration of video games enables the usage of the additional abilities of connected cars in video games. Therefore, the data exchange model for data exchange between the real-life and the in-game world is introduced. The data exchange model covers the data injection from the real-life into the video game and the extraction of data out of the game. Based on the data exchange model many projects can be implemented to realize the advanced integration. To demonstrate the potential of this vision the World Generation prototype and the prototype for the REST API based on the kart racing game SuperTuxKart are presented.

The World Generation prototype enables the SuperTuxKart track generation based on the real-world data of the car's environment. It allows the player to compete in the races taking place in a replication of the real-life in order to create immersion. For the computation of the in-game model of the real-life a writer for the SPM file format is proposed, and the usage of Suurballe's algorithm is suggested for determining the racetrack path. Racetrack path displacements can be determined using 2D spatial computations. Whereas the prototype is able to create recognizable world models, it is limited to freely accessible data. The usage of further data sources would make the generated in-game world more realistic. The more and the better the used data is, the more realistic the generated world becomes. The World Generation demonstrates how data injection into the game can be used to increase realism and is a first step towards the disappearance of the boundaries between the real-life and the in-game world.

The REST API for SuperTuxKart is the second implemented prototype. In the data exchange model, the REST API serves as middleware for data exchange in both directions. To enable the accessing of SuperTuxKart in a resource-oriented manner the domain model is presented. The domain model comprising the most important resources of SuperTuxKart. Requests are handled in parallel to not interrupt the gameplay. Data extraction and injection operations that require consistency can be executed in the intermediate time between the computation of two frames. Special effort is placed on the ability to serve as foundation for further projects. This is archived by making the REST API easy-to-use as the internal structure of SuperTuxKart does not have to be known in order to use the REST API. Also, it is possible to extend the REST API by further resource endpoints and attributes without changing the source code of the exiting handlers by adding and registering new independent handlers.

SuperTuxKart is a good example for video games that benefit from the additional possibilities that connected cars offer. It is important for further projects to also consider the additional hardware and the cloud capabilities of connected cars. The results of this work serve thereby as foundation and leverage for the contributions of further projects to the next abstraction level, i.e., that the projects can focus on their objective by using the basis that is introduced within this work.

Video games of other categories than racing games are also covered by the vision and the data exchange model but as the natural analogy of a vehicle is missing new metaphors need to be introduced. This may also cover including all passengers into the game.

Bibliography

[Bha99a] R. Bhandari. *Survivable networks: algorithms for diverse routing*. Springer Science & Business Media, 1999, pp. 21–37. ISBN: 0-7923-8381-8 (cit. on p. 25).

[Bha99b] R. Bhandari. *Survivable networks: algorithms for diverse routing*. Springer Science & Business Media, 1999, pp. 46–67. ISBN: 0-7923-8381-8 (cit. on p. 27).

[Ble20] BlenderGIS authors. *BlenderGIS*. 2020. URL: https://github.com/domlysz/BlenderGIS (visited on 10/29/2021) (cit. on p. 40).

[Ble21] Blender Foundation. *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. 2021. URL: https://www.blender.org/ (visited on 11/17/2021) (cit. on p. 39).

[Cam11] R. H. Campbell. "Deadlocks." In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 524–527. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_282 (cit. on p. 32).

[CBHG17] D. Chalupa, P. Balaghan, K. A. Hawick, N. A. Gordon. "Computational methods for finding long simple cycles in complex networks." In: *Knowl. Based Syst.* 125 (2017), pp. 96–107. DOI: 10.1016/j.knosys.2017.03.022 (cit. on p. 23).

[CSF20] Y. Cohen, R. Stern, A. Felner. "Solving the Longest Simple Path Problem with Heuristic Search." In: *Proceedings of the International Conference on Automated Planning and Scheduling* 30.1 (June 2020), pp. 75–79. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/6647 (visited on 12/14/2021) (cit. on p. 23).

[Die17] R. Diestel. *Graph theory*. Berlin, Heidelberg: Springer Berlin Heidelberg Imprint Springer, 2017. ISBN: 978-3-662-53621-6. DOI: 10.1007/978-3-662-53622-3 (cit. on pp. 23, 24).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on pp. 24, 25).

[Fie00] R. T. Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (visited on 11/19/2021) (cit. on p. 30).

[FR14] R. T. Fielding, J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. DOI: 10.17487/RFC7231. URL: https://rfc-editor.org/rfc/rfc7231.txt (cit. on p. 31).

[GA17] J.-l. Gailly, M. Adler. *zlib Home Site*. 2017. URL: https://zlib.net/ (visited on 12/03/2021) (cit. on p. 83).

[Goo21] Google. *google/googletest: GoogleTest - Google Testing and Mocking Framework*. 2021. URL: https://github.com/google/googletest (visited on 12/03/2021) (cit. on p. 76).

[ISO17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Working draft. Geneva, Switzerland: International Organization for Standardization, 2017. URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (visited on 11/19/2021) (cit. on pp. 31, 32).

[Jon21] J. de Jong. *math.js | an extensive math library for JavaScript and Node.js*. 2021. URL: https://mathjs.org/ (visited on 11/29/2021) (cit. on p. 52).

[Jön21] A. Jönsson. *AngelScript - AngelCode.com*. 2021. URL: https://www.angelcode.com/angelscript/ (visited on 12/02/2021) (cit. on p. 66).

[KDBA21] S. Knightley, D. Duponchel, F. Buchinger, A. Afonso. *JSZip*. 2021. URL: https://stuk.github.io/jszip/ (visited on 11/29/2021) (cit. on p. 52).

[Lew20] R. Lewis. "A Heuristic Algorithm for Finding Attractive Fixed-Length Circuits in Street Maps." In: *Computational Logistics*. Ed. by E. Lalla-Ruiz, M. Mes, S. Voß. Cham: Springer International Publishing, 2020, pp. 384–395. ISBN: 978-3-030-59747-4. DOI: 10.1007/978-3-030-59747-4_25 (cit. on pp. 26, 41, 42, 46).

[Lia14] E. Liao. *Dijkstra's Shortest Path Algorithm Runtime*. 2014. URL: https://inst.eecs.berkeley.edu/~cs61bl/r//cur/graphs/dijkstra-algorithm-runtime.html?topic=lab24.topic&step=4&course= (visited on 11/14/2021) (cit. on p. 25).

[Mac15] MacFreek. *Disjoint Path Finding*. 2015. URL: http://www.macfreek.nl/memory/index.php?title=Disjoint_Path_Finding&oldid=5934 (visited on 10/28/2021) (cit. on pp. 28, 29, 46, 47).

[Mar96] R. C. Martin. "The Open-Closed Principle." In: *More C++ gems* 19.96 (1996), p. 9 (cit. on p. 88).

[Mat17] Mathematics Stack Exchange. *A lower bound for number of cycles in a grid graph*. 2017. URL: https://math.stackexchange.com/questions/3396973/a-lower-bound-for-number-of-cycles-in-a-grid-graph (visited on 11/18/2021) (cit. on p. 41).

[Mic18] Microsoft. *Bing Maps Tile System - Bing Maps | Microsoft Docs*. 2018. URL: https://docs.microsoft.com/en-us/bingmaps/articles/bing-maps-tile-system (visited on 11/28/2021) (cit. on p. 53).

[Mic21] Microsoft. *Bing Maps - Directions, trip planning, traffic cameras & more*. 2021. URL: https://www.bing.com/maps (visited on 11/16/2021) (cit. on p. 34).

[ope13] opengl-tutorials. *Tutorial 5 : A Textured Cube*. 2013. URL: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/ (visited on 11/08/2021) (cit. on p. 22).

[Ope19] OpenStreetMap Wiki. *SuperTuxKart — OpenStreetMap Wiki*. 2019. URL: https://wiki.openstreetmap.org/w/index.php?title=SuperTuxKart&oldid=1888334 (visited on 10/25/2021) (cit. on p. 40).

[Ope21a] OpenJS Foundation. *Node.js*. 2021. URL: https://nodejs.org/en/ (visited on 11/29/2021) (cit. on p. 52).

[Ope21b] OpenStreetMap Wiki. *Elements — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Elements&oldid=2157322 (visited on 11/28/2021) (cit. on p. 52).

[Ope21c] OpenStreetMap Wiki. *OSM2World — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=OSM2World&oldid=2218700 (visited on 11/17/2021) (cit. on p. 40).

[PKW20] PKWARE Inc. *APPNOTE.TXT - .ZIP File Format Specification*. 2020. URL: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT (visited on 12/11/2021) (cit. on p. 83).

[Pra11] C. von Praun. "Race Conditions." In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 1691–1697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_36 (cit. on p. 32).

[Red17] Red Hat. *What is an API?* 2017. URL: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces (visited on 11/19/2021) (cit. on p. 30).

[Red20a] Red Hat. *Stateful vs stateless*. 2020. URL: https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless (visited on 11/19/2021) (cit. on p. 30).

[Red20b] Red Hat. *What is service-oriented architecture (SOA)?* 2020. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-service-oriented-architecture (visited on 11/19/2021) (cit. on p. 30).

[SPM21] SPM author(s). *stk-code/sp_mesh_loader.cpp at master · supertuxkart/stk-code · GitHub*. 2021. URL: https://github.com/supertuxkart/stk-blender/blob/master/io_scene_spm/export_spm.py (visited on 10/28/2021) (cit. on p. 19).

[Sup16] SuperTuxKart Team. *Making Tracks - SuperTuxKart*. 2016. URL: https://supertuxkart.net/index.php?title=Making_Tracks&oldid=5268 (visited on 11/12/2021) (cit. on pp. 22, 39, 65).

[Sup17] SuperTuxKart Team. *Making Tracks - SuperTuxKart*. 2017. URL: https://supertuxkart.net/index.php?title=Scripting&oldid=5367 (visited on 11/17/2021) (cit. on p. 66).

[Sup19] SuperTuxKart Team. *Making Tracks: Notes - SuperTuxKart*. 2019. URL: https://supertuxkart.net/index.php?title=Making_Tracks:_Notes&oldid=5760#Gameplay (visited on 10/25/2021) (cit. on pp. 46, 65).

[Sup20] SuperTuxKart Team. *Installing Add-Ons - SuperTuxKart*. 2020. URL: https://supertuxkart.net/index.php?title=Installing_Add-Ons&oldid=5812 (visited on 12/02/2021) (cit. on p. 65).

[Sup21a] SuperTuxKart Team. *stk-code/sp_mesh_loader.cpp at master · supertuxkart/stk-code · GitHub*. 2021. URL: https://github.com/supertuxkart/stk-code/blob/master/src/graphics/sp_mesh_loader.cpp (visited on 10/28/2021) (cit. on p. 19).

[Sup21b] SuperTuxKart Team. *SuperTuxKart*. 2021. URL: https://supertuxkart.net/index.php?title=Main_Page&oldid=5813 (visited on 11/11/2021) (cit. on p. 19).

[Sup21c] SuperTuxKart Team. *SuperTuxKart Add-ons*. 2021. URL: https://online.supertuxkart.net/ (visited on 12/02/2021) (cit. on p. 65).

[Sup21d] SuperTuxKart Team. *supertuxkart/stk-code: The code base of supertuxkart*. 2021. URL: https://github.com/supertuxkart/stk-code/ (visited on 12/02/2021) (cit. on pp. 65, 76).

[Suu74] J. W. Suurballe. “Disjoint paths in a network.” In: *Networks* 4.2 (1974), pp. 125–145. DOI: https://doi.org/10.1002/net.3230040204. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230040204 (cit. on p. 46).

[TY16] THL A29 Limited, a Tencent company, M. Yip. *RapidJSON: Main Page*. 2016. URL: https://rapidjson.org/ (visited on 12/03/2021) (cit. on p. 76).

[WZR18] D. Willems, O. Zehner, S. Ruzika. “On a Technique for Finding Running Tracks of Specific Length in a Road Network.” In: *Operations Research Proceedings 2017*. Ed. by N. Kliewer, J. F. Ehmke, R. Borndörfer. Cham: Springer International Publishing, 2018, pp. 333–338. ISBN: 978-3-319-89920-6. DOI: 10.1007/978-3-319-89920-6_45 (cit. on pp. 41, 46).

[Yen71] J. Y. Yen. “Finding the K Shortest Loopless Paths in a Network.” In: *Management Science* 17.11 (1971), pp. 712–716. ISSN: 00251909, 15265501. DOI: 10.1287/mnsc.17.11.712 (cit. on pp. 26, 27).

[yhi21] yhirose. *yhirose/cpp-httplib: A C++ header-only HTTP/HTTPS server and client library*. 2021. URL: https://github.com/yhirose/cpp-httplib (visited on 12/03/2021) (cit. on p. 76).

[Zab21] M. Zabriskie. *axios/axios: Promise based HTTP client for the browser and node.js*. 2021. URL: https://github.com/axios/axios (visited on 11/29/2021) (cit. on p. 52).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature