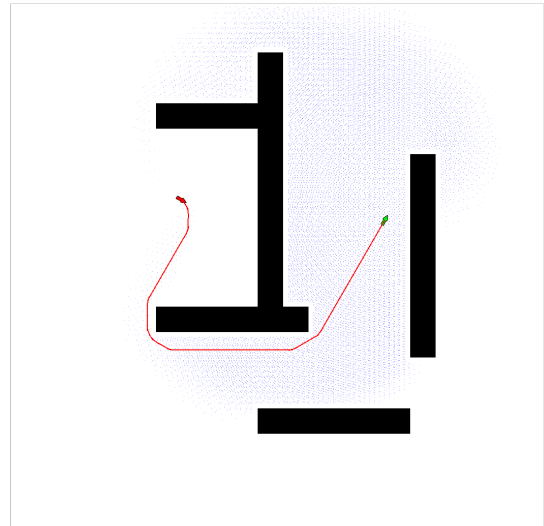
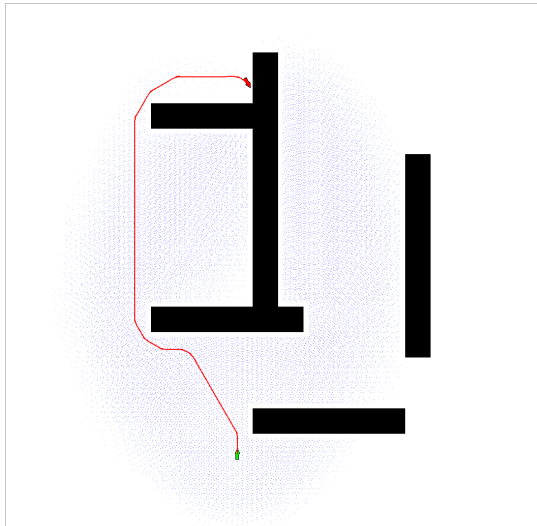

City Based CBS Documentation

<https://github.com/Faywyn/city-CBS-Astar>



1 Class Index	2
1.1 Class List	2
2 File Index	3
2.1 File List	3
3 Class Documentation	4
3.1 _aStarConflict Struct Reference	4
3.1.1 Detailed Description	5
3.2 _aStarNode Struct Reference	5
3.2.1 Detailed Description	5
3.3 _cityGraphNeighbor Struct Reference	5
3.3.1 Detailed Description	6
3.4 _cityGraphPoint Struct Reference	6
3.4.1 Detailed Description	6
3.5 _cityMapBuilding Struct Reference	6
3.5.1 Detailed Description	6
3.6 _cityMapGreenArea Struct Reference	6
3.6.1 Detailed Description	7
3.7 _cityMapIntersection Struct Reference	7
3.7.1 Detailed Description	7
3.8 _cityMapRoad Struct Reference	7
3.8.1 Detailed Description	7
3.9 _cityMapSegment Struct Reference	8
3.9.1 Detailed Description	8
3.10 _cityMapWaterArea Struct Reference	8
3.10.1 Detailed Description	8
3.11 _data Struct Reference	8
3.11.1 Detailed Description	8
3.12 _managerCBSNode Struct Reference	9
3.12.1 Detailed Description	9
3.13 AStar Class Reference	9
3.13.1 Detailed Description	9
3.13.2 Constructor & Destructor Documentation	9
3.13.3 Member Function Documentation	10
3.14 Car Class Reference	10
3.14.1 Detailed Description	11
3.14.2 Member Function Documentation	11
3.15 CityGraph Class Reference	15
3.15.1 Detailed Description	15
3.15.2 Member Function Documentation	15
3.16 CityMap Class Reference	16
3.16.1 Detailed Description	17

3.16.2 Member Function Documentation	17
3.17 ConstraintController Class Reference	19
3.17.1 Detailed Description	19
3.17.2 Member Function Documentation	20
3.18 DataManager Class Reference	22
3.18.1 Detailed Description	22
3.18.2 Constructor & Destructor Documentation	22
3.18.3 Member Function Documentation	22
3.19 Dubins Class Reference	23
3.19.1 Detailed Description	23
3.19.2 Constructor & Destructor Documentation	23
3.19.3 Member Function Documentation	24
3.20 DubinsPath Class Reference	25
3.20.1 Detailed Description	25
3.20.2 Constructor & Destructor Documentation	26
3.21 FileSelector Class Reference	26
3.21.1 Detailed Description	26
3.22 Manager Class Reference	26
3.22.1 Detailed Description	27
3.22.2 Constructor & Destructor Documentation	27
3.22.3 Member Function Documentation	28
3.23 PriorityQueue< T > Class Template Reference	30
3.23.1 Detailed Description	30
3.23.2 Constructor & Destructor Documentation	31
3.23.3 Member Function Documentation	31
3.24 Renderer Class Reference	32
3.24.1 Detailed Description	32
3.24.2 Member Function Documentation	32
3.25 Test Class Reference	33
3.25.1 Detailed Description	33
3.26 TimedAStar Class Reference	33
3.26.1 Detailed Description	33
3.26.2 Constructor & Destructor Documentation	33
3.26.3 Member Function Documentation	35
4 File Documentation	35
4.1 aStar.h File Reference	35
4.1.1 Detailed Description	35
4.2 aStar.h	36
4.3 car.h File Reference	37
4.3.1 Detailed Description	37
4.4 car.h	37

4.5 cityGraph.h File Reference	38
4.5.1 Detailed Description	38
4.6 cityGraph.h	39
4.7 cityMap.h	40
4.8 config.h File Reference	41
4.8.1 Detailed Description	41
4.9 config.h	41
4.10 dataManager.h File Reference	41
4.10.1 Detailed Description	42
4.11 dataManager.h	42
4.12 dubins.h File Reference	42
4.12.1 Detailed Description	42
4.13 dubins.h	43
4.14 fileSelector.h File Reference	43
4.14.1 Detailed Description	43
4.15 fileSelector.h	44
4.16 manager.h File Reference	44
4.16.1 Detailed Description	44
4.17 manager.h	44
4.18 priorityQueue.h File Reference	45
4.18.1 Detailed Description	45
4.19 priorityQueue.h	45
4.20 renderer.h File Reference	46
4.20.1 Detailed Description	47
4.20.2 Function Documentation	47
4.21 renderer.h	47
4.22 test.h File Reference	48
4.22.1 Detailed Description	48
4.23 test.h	48
4.24 utils.h File Reference	48
4.24.1 Detailed Description	49
4.24.2 Function Documentation	49
4.25 utils.h	51
4.26 index.py	51
4.27 aStar.cpp File Reference	53
4.27.1 Detailed Description	54
4.28 aStar.cpp	54
4.29 car.cpp File Reference	55
4.29.1 Detailed Description	55
4.30 car.cpp	55
4.31 cityGraph.cpp File Reference	57
4.31.1 Detailed Description	58

4.32 cityGraph.cpp	58
4.33 cityMap.cpp File Reference	62
4.33.1 Detailed Description	62
4.34 cityMap.cpp	62
4.35 constraintController.cpp File Reference	66
4.35.1 Detailed Description	67
4.36 constraintController.cpp	67
4.37 dataManager.cpp File Reference	68
4.37.1 Detailed Description	68
4.38 dataManager.cpp	69
4.39 dubins.cpp File Reference	70
4.39.1 Detailed Description	70
4.40 dubins.cpp	70
4.41 fileSelector.cpp File Reference	71
4.41.1 Detailed Description	72
4.42 fileSelector.cpp	72
4.43 main.cpp File Reference	73
4.43.1 Detailed Description	73
4.44 main.cpp	73
4.45 manager.cpp File Reference	74
4.45.1 Detailed Description	74
4.46 manager.cpp	74
4.47 managerCBS.cpp File Reference	75
4.47.1 Detailed Description	75
4.48 managerCBS.cpp	75
4.49 renderer.cpp File Reference	79
4.49.1 Detailed Description	80
4.50 renderer.cpp	80
4.51 test.cpp File Reference	84
4.51.1 Detailed Description	84
4.52 test.cpp	84
4.53 timedAStar.cpp File Reference	85
4.53.1 Detailed Description	85
4.54 timedAStar.cpp	85
4.55 utils.cpp File Reference	87
4.55.1 Detailed Description	88
4.55.2 Function Documentation	88
4.56 utils.cpp	89

1 Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

_aStarConflict		
A conflict for the A* algorithm		4
_aStarNode		
A node for the A* algorithm		5
_cityGraphNeighbor		
A neighbor of a point in the city graph		5
_cityGraphPoint		
A point in the city graph		6
_cityMapBuilding		
A building in the city map		6
_cityMapGreenArea		
A green area in the city map		6
_cityMapIntersection		
An intersection in the city map		7
_cityMapRoad		
A road in the city map		7
_cityMapSegment		
A segment in the city map		8
_cityMapWaterArea		
A water area in the city map		8
_data		
Data structure		8
_managerCBSNode		
A node for the CBS algorithm		9
AStar		
A* algorithm		9
Car		
A car in the city		10
CityGraph		
A graph representing the city's streets and intersections using a graph		15
CityMap		
A city map		16
ConstraintController		
Controller for constraints		19
DataManager		
Data manager		22
Dubins		
Dubins path used to calculate the path between two points in the city graph		23

DubinsPath	
Dubins path used to calculate the path between two points in the city graph	25
FileSelector	
A file selector	26
Manager	
A manager for the cars	26
PriorityQueue< T >	
Priority Queue	30
Renderer	
A renderer for the city	32
Test	
A class for testing the project	33
TimedAStar	
Timed A* algorithm	33

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

aStar.h	
A* algorithm	35
car.h	
A car in the city	37
cityGraph.h	
A graph representing the city's streets and intersections using a graph	38
cityMap.h	40
config.h	
Configuration file	41
dataManager.h	
Data manager	41
dubins.h	
Dubins path	42
fileSelector.h	
File selector	43
manager.h	
Manager for the cars	44
priorityQueue.h	
Priority Queue	45
renderer.h	
A renderer for the city	46
test.h	
A header file for the Test class	48

utils.h	
Utility functions	48
index.py	51
aStar.cpp	
A* algorithm implementation	53
car.cpp	
Car class implementation	55
cityGraph.cpp	
City graph implementation	57
cityMap.cpp	
CityMap class implementation	62
constraintController.cpp	
ConstraintController class implementation	66
dataManager.cpp	
Data manager	68
dubins.cpp	
Dubins path implementation	70
fileSelector.cpp	
File selector implementation	71
main.cpp	
Main file	73
manager.cpp	
Implementation of the Manager class	74
managerCBS.cpp	
CBS algorithm implementation	75
renderer.cpp	
Implementation of the Renderer class	79
test.cpp	
A file for testing the project	84
timedAStar.cpp	
Timed A* algorithm implementation	85
utils.cpp	
Utility functions implementation	87

3 Class Documentation

3.1 `_aStarConflict` Struct Reference

A conflict for the A* algorithm.

```
#include <aStar.h>
```

Public Attributes

- `CityGraph::point` point

The point in the graph.

- `int time`

The time of the conflict.

- `int car`

The car that caused the conflict.

3.1.1 Detailed Description

A conflict for the A* algorithm.

This struct represents a conflict for the A* algorithm. It contains the point in the graph, the time of the conflict and the car that caused the conflict.

Definition at line 41 of file [aStar.h](#).

The documentation for this struct was generated from the following file:

- [aStar.h](#)

3.2 `_aStarNode` Struct Reference

A node for the A* algorithm.

```
#include <aStar.h>
```

Public Attributes

- [CityGraph::point](#) `point`

The point in the graph.

- `double speed`

The speed of the car.

- `std::pair< CityGraph::point, CityGraph::neighbor >` `arcFrom`

The arc from which the node was reached.

3.2.1 Detailed Description

A node for the A* algorithm.

This struct represents a node for the A* algorithm. It contains the point in the graph, the speed of the car and the arc from which the node was reached.

Definition at line 20 of file [aStar.h](#).

The documentation for this struct was generated from the following file:

- [aStar.h](#)

3.3 `_cityGraphNeighbor` Struct Reference

A neighbor of a point in the city graph.

```
#include <cityGraph.h>
```

Public Attributes

- [_cityGraphPoint](#) `point`

The neighbor point.

- `double maxSpeed`

The maximum speed to reach the neighbor point.

- `double turningRadius`

The turning radius to reach the neighbor point.

- `double distance`

The distance to reach the neighbor point.

- `bool isRightWay`

If it is the right way.

3.3.1 Detailed Description

A neighbor of a point in the city graph.

This struct represents a neighbor of a point in the city graph. It contains the neighbor point, the maximum speed to reach it, the turning radius to reach it, the distance to reach it and if it is the right way.

Definition at line 43 of file [cityGraph.h](#).

The documentation for this struct was generated from the following file:

- [cityGraph.h](#)

3.4 _cityGraphPoint Struct Reference

A point in the city graph.

```
#include <cityGraph.h>
```

Public Attributes

- `sf::Vector2f` **position**
The position of the point.
- `double` **angle**
The angle of the point.

3.4.1 Detailed Description

A point in the city graph.

This struct represents a point in the city graph. It contains the position and the angle of the point.

Definition at line 20 of file [cityGraph.h](#).

The documentation for this struct was generated from the following file:

- [cityGraph.h](#)

3.5 _cityMapBuilding Struct Reference

A building in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `std::vector< sf::Vector2f >` **points**
The points of the building.

3.5.1 Detailed Description

A building in the city map.

Definition at line 34 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- [cityMap.h](#)

3.6 _cityMapGreenArea Struct Reference

A green area in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `std::vector< sf::Vector2f >` **points**
The points of the green area.
- `int` **type**
The type of the green area.

3.6.1 Detailed Description

A green area in the city map.

Definition at line 42 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- `cityMap.h`

3.7 `_cityMapIntersection` Struct Reference

An intersection in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `int id`
The id of the intersection.
- `sf::Vector2f center`
The center of the intersection.
- `double radius`
The radius of the intersection.
- `std::vector< std::pair< int, int > > roadSegmentIds`
The ids of the road segments (roadId, segmentId). The segments are the same for both directions of the road.

3.7.1 Detailed Description

An intersection in the city map.

Definition at line 59 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- `cityMap.h`

3.8 `_cityMapRoad` Struct Reference

A road in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `int id`
The id of the road.
- `std::vector< _cityMapSegment > segments`
The segments of the road.
- `double width`
The width of the road.
- `int numLanes`
The number of lanes of the road.

3.8.1 Detailed Description

A road in the city map.

Definition at line 23 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- `cityMap.h`

3.9 `_cityMapSegment` Struct Reference

A segment in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `sf::Vector2f p1`
The first point of the segment.
- `sf::Vector2f p2`
The second point of the segment.
- `sf::Vector2f p1_offset`
The offset of the first point, used for the intersection.
- `sf::Vector2f p2_offset`
The offset of the second point, used for the intersection.
- `double angle`
The angle of the segment.

3.9.1 Detailed Description

A segment in the city map.

Definition at line 11 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- [cityMap.h](#)

3.10 `_cityMapWaterArea` Struct Reference

A water area in the city map.

```
#include <cityMap.h>
```

Public Attributes

- `std::vector< sf::Vector2f > points`
The points of the water area.

3.10.1 Detailed Description

A water area in the city map.

Definition at line 51 of file [cityMap.h](#).

The documentation for this struct was generated from the following file:

- [cityMap.h](#)

3.11 `_data` Struct Reference

Data structure.

```
#include <dataManager.h>
```

3.11.1 Detailed Description

Data structure.

This struct represents the data structure.

Definition at line 18 of file [dataManager.h](#).

The documentation for this struct was generated from the following file:

- [dataManager.h](#)

3.12 `_managerCBSNode` Struct Reference

A node for the CBS algorithm.

```
#include <manager.h>
```

Public Attributes

- `std::vector< std::vector< sf::Vector2f > > paths`
The paths for all agents.
- [ConstraintController](#) `constraints`
The constraints for all agents.
- `std::vector< double > costs`
The individual path costs.
- `double cost`
The total cost.
- `int depth`
The depth in the CBS tree.
- `bool hasResolved`
If the node has resolved conflicts.

3.12.1 Detailed Description

A node for the CBS algorithm.

This struct represents a node for the CBS algorithm. It contains the paths for all agents, the constraints for all agents, the individual path costs, the total cost, the depth in the CBS tree and if the node has resolved conflicts.

Definition at line 24 of file [manager.h](#).

The documentation for this struct was generated from the following file:

- [manager.h](#)

3.13 AStar Class Reference

A* algorithm.

```
#include <aStar.h>
```

Public Member Functions

- [AStar](#) ([CityGraph::point](#) start, [CityGraph::point](#) end, const [CityGraph](#) &cityGraph)
Constructor.
- `std::vector< node > findPath ()`
Find the path.

3.13.1 Detailed Description

A* algorithm.

This class represents the A* algorithm. It is used to find the shortest path between two points in a graph.

Definition at line 74 of file [aStar.h](#).

3.13.2 Constructor & Destructor Documentation

AStar()

```
AStar::AStar (
    CityGraph::point start,
    CityGraph::point end,
    const CityGraph & cityGraph)
```

Constructor.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point
<i>cityGraph</i>	The graph

Definition at line 21 of file [aStar.cpp](#).

3.13.3 Member Function Documentation

findPath()

```
std::vector< node > AStar::findPath () [inline]
```

Find the path.

Returns

The path

Definition at line 91 of file [aStar.h](#).

The documentation for this class was generated from the following files:

- [aStar.h](#)
- [aStar.cpp](#)

3.14 Car Class Reference

A car in the city.

```
#include <car.h>
```

Public Member Functions

- **Car ()**
Constructor.
- void [assignStartEnd](#) ([CityGraph::point](#) start, [CityGraph::point](#) end)
Assign the start and end points.
- void [chooseRandomStartEndPath](#) ([CityGraph](#) &graph, [CityMap](#) &cityMap)
Choose a random start and end point in the graph.
- void [assignPath](#) (std::vector< [AStar::node](#) > path)
Assign a path to the car.
- void [assignExistingPath](#) (std::vector< sf::Vector2f > path)
Assign an existing path to the car.
- void **move** ()
Move the car, move to the next point in the path.
- void [render](#) (sf::RenderWindow &window)
Render the car.
- [CityGraph::point](#) [getStart](#) ()
Get the start point.
- [CityGraph::point](#) [getEnd](#) ()
Get the end point.
- double [getSpeed](#) ()
Get the current point in the path.
- double [getSpeedAt](#) (int index)
Get the speed at a certain index in the path.
- double [getAverageSpeed](#) ([CityGraph](#) &graph)
Get the average speed of the car.

- double `getRemainingTime ()`
Get the remaining time to reach the end point.
- double `getElapsedTime ()`
Get the elapsed time since the start of the car.
- double `getPathTime ()`
Get the time to reach the end point from the start point.
- double `getRemainingDistance ()`
Get the remaining distance to reach the end point.
- double `getElapsedDistance ()`
Get the elapsed distance since the start of the car.
- double `getPathLength ()`
Get the distance to reach the end point from the start point.
- `sf::Vector2f` `getPosition ()`
Get the position of the car.
- `std::vector< sf::Vector2f >` `getPath ()`
Get the path of the car.
- `std::vector< AStar::node >` `getAStarPath ()`
Get the path of the car from the A algorithm.*
- void `toggleDebug ()`
Toggle the debug mode. In debug mode, the path of the car is rendered and the car is rendered in red.

3.14.1 Detailed Description

A car in the city.

This class represents a car in the city. It contains the start and end points of the car, the path of the car and the current point in the path.

Definition at line 23 of file [car.h](#).

3.14.2 Member Function Documentation

`assignExistingPath()`

```
void Car::assignExistingPath (
    std::vector< sf::Vector2f > path)
```

Assign an existing path to the car.

Parameters

<i>path</i>	The path
-------------	----------

Definition at line 87 of file [car.cpp](#).

`assignPath()`

```
void Car::assignPath (
    std::vector< AStar::node > path)
```

Assign a path to the car.

Parameters

<i>path</i>	The path
-------------	----------

Definition at line 76 of file [car.cpp](#).

assignStartEnd()

```
void Car::assignStartEnd (
    CityGraph::point start,
    CityGraph::point end) [inline]
```

Assign the start and end points.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point

Definition at line 35 of file [car.h](#).

chooseRandomStartEndPath()

```
void Car::chooseRandomStartEndPath (
    CityGraph & graph,
    CityMap & cityMap)
```

Choose a random start and end point in the graph.

Parameters

<i>graph</i>	The graph
<i>cityMap</i>	The city map

Definition at line 142 of file [car.cpp](#).

getAStarPath()

```
std::vector< AStar::node > Car::getAStarPath () [inline]
```

Get the path of the car from the A* algorithm.

Returns

The path

Definition at line 154 of file [car.h](#).

getAverageSpeed()

```
double Car::getAverageSpeed (
    CityGraph & graph)
```

Get the average speed of the car.

Parameters

<i>graph</i>	The graph
--------------	-----------

Returns

The average speed

Definition at line 172 of file [car.cpp](#).

getElapsedDistance()

```
double Car::getElapsedDistance ()
```

Get the elapsed distance since the start of the car.

Returns

The elapsed distance

Definition at line 122 of file [car.cpp](#).

getElapsedTime()

```
double Car::getElapsedTime ()
```

Get the elapsed time since the start of the car.

Returns

The elapsed time

Definition at line 109 of file [car.cpp](#).

getEnd()

```
CityGraph::point Car::getEnd () [inline]
```

Get the end point.

Returns

The end point

Definition at line 80 of file [car.h](#).

getPath()

```
std::vector< sf::Vector2f > Car::getPath () [inline]
```

Get the path of the car.

Returns

The path

Definition at line 148 of file [car.h](#).

getPathLength()

```
double Car::getPathLength ()
```

Get the distance to reach the end point from the start point.

Returns

The distance

Definition at line 132 of file [car.cpp](#).

getPathTime()

```
double Car::getPathTime ()
```

Get the time to reach the end point from the start point.

Returns

The time

Definition at line 110 of file [car.cpp](#).

getPosition()

```
sf::Vector2f Car::getPosition () [inline]
```

Get the position of the car.

Returns

The position

Definition at line 142 of file [car.h](#).

getRemainingDistance()

```
double Car::getRemainingDistance ()
```

Get the remaining distance to reach the end point.

Returns

The remaining distance

Definition at line 112 of file [car.cpp](#).

getRemainingTime()

```
double Car::getRemainingTime ()
```

Get the remaining time to reach the end point.

Returns

The remaining time

Definition at line 108 of file [car.cpp](#).

getSpeed()

```
double Car::getSpeed ()
```

Get the current point in the path.

Returns

The current point in the path

Definition at line 92 of file [car.cpp](#).

getSpeedAt()

```
double Car::getSpeedAt (
    int index)
```

Get the speed at a certain index in the path.

Parameters

<i>index</i>	The index
--------------	-----------

Returns

The speed at the index

Definition at line 100 of file [car.cpp](#).

getStart()

```
CityGraph::point Car::getStart () [inline]
```

Get the start point.

Returns

The start point

Definition at line 74 of file [car.h](#).

render()

```
void Car::render (
    sf::RenderWindow & window)
```

Render the car.

Parameters

<i>window</i>	The window
---------------	------------

Definition at line 28 of file [car.cpp](#).

The documentation for this class was generated from the following files:

- [car.h](#)
- [car.cpp](#)

3.15 CityGraph Class Reference

A graph representing the city's streets and intersections using a graph.

```
#include <cityGraph.h>
```

Public Member Functions

- void [createGraph](#) (const [CityMap](#) &cityMap)
Create a city graph.
- std::unordered_map< [point](#), std::vector< [neighbor](#) > > [getNeighbors](#) () const
Get neighbors map.
- std::unordered_set< [point](#) > [getGraphPoints](#) () const
Get graph points.
- [point](#) [getRandomPoint](#) () const
Get random point.
- double [getHeight](#) () const
Get the height of the city graph.
- double [getWidth](#) () const
Get the width of the city graph.

3.15.1 Detailed Description

A graph representing the city's streets and intersections using a graph.

This class represents the city graph. It contains the neighbors of each point in the graph and the graph points.

Definition at line 82 of file [cityGraph.h](#).

3.15.2 Member Function Documentation

createGraph()

```
void CityGraph::createGraph (
    const CityMap & cityMap)
```

Create a city graph.

This constructor creates a city graph from a city map.

Parameters

<i>cityMap</i>	The city map
----------------	--------------

Definition at line 23 of file [cityGraph.cpp](#).

getGraphPoints()

```
std::unordered_set< point > CityGraph::getGraphPoints () const [inline]
```

Get graph points.

Returns

Graph points

Definition at line 106 of file [cityGraph.h](#).

getHeight()

```
double CityGraph::getHeight () const [inline]
```

Get the height of the city graph.

Returns

The height of the city graph

Definition at line 118 of file [cityGraph.h](#).

getNeighbors()

```
std::unordered_map< point, std::vector< neighbor > > CityGraph::getNeighbors () const [inline]
```

Get neighbors map.

Returns

Neighbors map

Definition at line 100 of file [cityGraph.h](#).

getRandomPoint()

```
CityGraph::point CityGraph::getRandomPoint () const
```

Get random point.

Returns

Random point

Definition at line 274 of file [cityGraph.cpp](#).

getWidth()

```
double CityGraph::getWidth () const [inline]
```

Get the width of the city graph.

Returns

The width of the city graph

Definition at line 124 of file [cityGraph.h](#).

The documentation for this class was generated from the following files:

- [cityGraph.h](#)
- [cityGraph.cpp](#)

3.16 CityMap Class Reference

A city map.

```
#include <cityMap.h>
```

Public Member Functions

- **CityMap ()**
Constructor.
- void **loadFile** (const std::string &filename)
Load a city map from a file.
- bool **isCityMapLoaded** () const
Check if the city map is loaded.
- std::vector< [road](#) > **getRoads** () const
Get the roads.

- `std::vector< intersection > getIntersections () const`
Get the intersections.
- `std::vector< building > getBuildings () const`
Get the buildings.
- `std::vector< greenArea > getGreenAreas () const`
Get the green areas.
- `std::vector< waterArea > getWaterAreas () const`
Get the water areas.
- `sf::Vector2f getMinLatLon () const`
Get the minimum latitude and longitude.
- `sf::Vector2f getMaxLatLon () const`
Get the maximum latitude and longitude.
- `int getWidth () const`
Get the width of the city map.
- `int getHeight () const`
Get the height of the city map.

3.16.1 Detailed Description

A city map.

This class represents the city map. It contains the roads, intersections, buildings, green areas and water areas of the city.

Definition at line 74 of file [cityMap.h](#).

3.16.2 Member Function Documentation

getBuildings()

```
std::vector< building > CityMap::getBuildings () const [inline]
```

Get the buildings.

Returns

The buildings

Definition at line 116 of file [cityMap.h](#).

getGreenAreas()

```
std::vector< greenArea > CityMap::getGreenAreas () const [inline]
```

Get the green areas.

Returns

The green areas

Definition at line 122 of file [cityMap.h](#).

getHeight()

```
int CityMap::getHeight () const [inline]
```

Get the height of the city map.

Returns

The height of the city map

Definition at line 152 of file [cityMap.h](#).

getIntersections()

```
std::vector< intersection > CityMap::getIntersections () const [inline]
```

Get the intersections.

Returns

The intersections

Definition at line 110 of file [cityMap.h](#).

getMaxLatLon()

```
sf::Vector2f CityMap::getMaxLatLon () const [inline]
```

Get the maximum latitude and longitude.

Returns

The maximum latitude and longitude

Definition at line 140 of file [cityMap.h](#).

getMinLatLon()

```
sf::Vector2f CityMap::getMinLatLon () const [inline]
```

Get the minimum latitude and longitude.

Returns

The minimum latitude and longitude

Definition at line 134 of file [cityMap.h](#).

getRoads()

```
std::vector< road > CityMap::getRoads () const [inline]
```

Get the roads.

Returns

The roads

Definition at line 104 of file [cityMap.h](#).

getWaterAreas()

```
std::vector< waterArea > CityMap::getWaterAreas () const [inline]
```

Get the water areas.

Returns

The water areas

Definition at line 128 of file [cityMap.h](#).

getWidth()

```
int CityMap::getWidth () const [inline]
```

Get the width of the city map.

Returns

The width of the city map

Definition at line 146 of file [cityMap.h](#).

isCityMapLoaded()

```
bool CityMap::isCityMapLoaded () const [inline]
```

Check if the city map is loaded.

Returns

True if the city map is loaded, false otherwise

Definition at line 98 of file [cityMap.h](#).

loadFile()

```
void CityMap::loadFile (
    const std::string & filename)
```

Load a city map from a file.

Parameters

<i>filename</i>	The filename
-----------------	--------------

Definition at line 23 of file [cityMap.cpp](#).

The documentation for this class was generated from the following files:

- [cityMap.h](#)
- [cityMap.cpp](#)

3.17 ConstraintController Class Reference

Controller for constraints.

```
#include <aStar.h>
```

Public Member Functions

- **ConstraintController ()**
Constructor.
- [ConstraintController copy \(\)](#)
Copy constructor.
- [ConstraintController copy \(std::vector< int > cars\)](#)
Copy constructor.
- void [addConstraint \(AStar::conflict constraints\)](#)
Add a constraint.
- bool [hasConstraint \(AStar::conflict constraint\)](#)
Check if a constraint exists.
- bool [checkConstraints](#) (int car, double speed, double newSpeed, double time, [CityGraph::point](#) from, [CityGraph::neighbor](#) to)
Check if a car can move to a certain point in the graph at a certain time.

3.17.1 Detailed Description

Controller for constraints.

This class is used to control the constraints of the A* algorithm. It is used to check if a car can move to a certain point in the graph at a certain time.

Definition at line 114 of file [aStar.h](#).

3.17.2 Member Function Documentation

addConstraint()

```
void ConstraintController::addConstraint (  
    AStar::conflict constraints)
```

Add a constraint.

Parameters

<i>constraints</i>	The constraint to add
--------------------	-----------------------

Definition at line 15 of file [constraintController.cpp](#).

checkConstraints()

```
bool ConstraintController::checkConstraints (
    int car,
    double speed,
    double newSpeed,
    double time,
    CityGraph::point from,
    CityGraph::neighbor to)
```

Check if a car can move to a certain point in the graph at a certain time.

Parameters

<i>car</i>	The car
<i>speed</i>	The speed of the car
<i>newSpeed</i>	The new speed of the car
<i>time</i>	The time
<i>from</i>	The point from which the car is moving
<i>to</i>	The point to which the car is moving

Returns

True if the car can move to the point, false otherwise

Definition at line 74 of file [constraintController.cpp](#).

copy() [1/2]

```
ConstraintController ConstraintController::copy ()
```

Copy constructor.

Returns

A copy of the object

Definition at line 52 of file [constraintController.cpp](#).

copy() [2/2]

```
ConstraintController ConstraintController::copy (
    std::vector< int > cars)
```

Copy constructor.

Parameters

<i>cars</i>	The cars to copy
-------------	------------------

Returns

A copy of the object

Definition at line 60 of file [constraintController.cpp](#).

hasConstraint()

```
bool ConstraintController::hasConstraint (
    AStar::conflict constraint)
```

Check if a constraint exists.

Parameters

<i>constraint</i>	The constraint to check
-------------------	-------------------------

Returns

True if the constraint exists, false otherwise

Definition at line 32 of file [constraintController.cpp](#).

The documentation for this class was generated from the following files:

- [aStar.h](#)
- [constraintController.cpp](#)

3.18 DataManager Class Reference

Data manager.

```
#include <dataManager.h>
```

Public Member Functions

- [DataManager](#) (std::string filename)
Constructor.
- void [createData](#) (int numData, int numCarsMin, int numCarsMax, std::string mapName)
Create data. It launches multiple simulations with different number of cars and car densities. Then, it calculates different statistics and stores them in a file.

3.18.1 Detailed Description

Data manager.

This class represents the data manager. It creates data and stores it in a file.

Definition at line 30 of file [dataManager.h](#).

3.18.2 Constructor & Destructor Documentation

DataManager()

```
DataManager::DataManager (
    std::string filename)
```

Constructor.

Parameters

<i>filename</i>	The filename
-----------------	--------------

Definition at line 20 of file [dataManager.cpp](#).

3.18.3 Member Function Documentation

createData()

```
void DataManager::createData (
    int numData,
    int numCarsMin,
```

```
int numCarsMax,
std::string mapName)
```

Create data. It launches multiple simulations with different number of cars and car densities. Then, it calculates different statistics and stores them in a file.

Parameters

<i>numData</i>	The number of data
<i>numCarsMin</i>	The minimum number of cars
<i>numCarsMax</i>	The maximum number of cars
<i>mapName</i>	The map name

Definition at line 28 of file [dataManager.cpp](#).

The documentation for this class was generated from the following files:

- [dataManager.h](#)
- [dataManager.cpp](#)

3.19 Dubins Class Reference

[Dubins](#) path used to calculate the path between two points in the city graph.

```
#include <dubins.h>
```

Public Member Functions

- [Dubins](#) ([CityGraph::point](#) start, [CityGraph::neighbor](#) end)
Constructor with start and end points.
- [Dubins](#) ([CityGraph::point](#) start, [CityGraph::neighbor](#) end, double startSpeed)
Constructor with start point, end point and start speed.
- [Dubins](#) ([CityGraph::point](#) start, [CityGraph::neighbor](#) end, double startSpeed, double endSpeed)
Constructor with start point, end point, start speed and end speed.
- [~Dubins](#) ()
Destructor.
- double [distance](#) ()
Get the distance to reach the end point.
- double [time](#) ()
Get the time to reach the end point.
- [CityGraph::point](#) [point](#) (double [time](#))
Get the point at a certain time in the path using interpolation.
- `std::vector< CityGraph::point >` [path](#) ()
Get the path using interpolation.

3.19.1 Detailed Description

[Dubins](#) path used to calculate the path between two points in the city graph.

This class represents a [Dubins](#) path used to calculate the path between two points in the city graph. Given the start and end points, it calculates the path, the distance and the time to reach the end point.

Definition at line 26 of file [dubins.h](#).

3.19.2 Constructor & Destructor Documentation

[Dubins](#)() [1/3]

```
Dubins::Dubins (
    CityGraph::point start,
    CityGraph::neighbor end)
```

Constructor with start and end points.

The class will be initialized with the start and end points. The car will run without speed limits.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point

Definition at line 11 of file [dubins.cpp](#).

Dubins() [2/3]

```
Dubins::Dubins (  
    CityGraph::point start,  
    CityGraph::neighbor end,  
    double startSpeed)
```

Constructor with start point, end point and start speed.

The class will be initialized with the start and end points and the start speed. The car will accelerate to the maximum speed.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point
<i>startSpeed</i>	The start speed

Definition at line 14 of file [dubins.cpp](#).

Dubins() [3/3]

```
Dubins::Dubins (  
    CityGraph::point start,  
    CityGraph::neighbor end,  
    double startSpeed,  
    double endSpeed)
```

Constructor with start point, end point, start speed and end speed.

The class will be initialized with the start and end points, the start and end speeds. The car will accelerate uniformly to the maximum speed.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point
<i>startSpeed</i>	The start speed
<i>endSpeed</i>	The end speed

Definition at line 34 of file [dubins.cpp](#).

3.19.3 Member Function Documentation**distance()**

```
double Dubins::distance () [inline]
```

Get the distance to reach the end point.

Returns

The distance

Definition at line 72 of file [dubins.h](#).

path()

```
std::vector< CityGraph::point > Dubins::path ()
```

Get the path using interpolation.

Returns

The path

Definition at line 85 of file [dubins.cpp](#).

point()

```
CityGraph::point Dubins::point (
    double time)
```

Get the point at a certain time in the path using interpolation.

Parameters

<i>time</i>	The time
-------------	----------

Returns

The point

Definition at line 64 of file [dubins.cpp](#).

time()

```
double Dubins::time ()
```

Get the time to reach the end point.

Returns

The time

Definition at line 62 of file [dubins.cpp](#).

The documentation for this class was generated from the following files:

- [dubins.h](#)
- [dubins.cpp](#)

3.20 DubinsPath Class Reference

[Dubins](#) path used to calculate the path between two points in the city graph.

```
#include <dubins.h>
```

Public Member Functions

- [DubinsPath](#) (std::vector< [AStar::node](#) > path)
Constructor with path.
- std::vector< [CityGraph::point](#) > path ()
Get the path.

3.20.1 Detailed Description

[Dubins](#) path used to calculate the path between two points in the city graph.

This class represents a [Dubins](#) path used to calculate the path between two points in the city graph. Given the start and end points, it calculates the path, the distance and the time to reach the end point.

Definition at line 112 of file [dubins.h](#).

3.20.2 Constructor & Destructor Documentation

DubinsPath()

```
DubinsPath::DubinsPath (
    std::vector< AStar::node > path)
```

Constructor with path.

The class will be initialized with the path.

Parameters

<i>path</i>	The path
-------------	----------

Definition at line 95 of file [dubins.cpp](#).

The documentation for this class was generated from the following files:

- [dubins.h](#)
- [dubins.cpp](#)

3.21 FileSelector Class Reference

A file selector.

```
#include <fileSelector.h>
```

3.21.1 Detailed Description

A file selector.

This class represents a file selector. It allows the user to select a file from a folder.

Definition at line 19 of file [fileSelector.h](#).

The documentation for this class was generated from the following files:

- [fileSelector.h](#)
- [fileSelector.cpp](#)

3.22 Manager Class Reference

A manager for the cars.

```
#include <manager.h>
```

Public Member Functions

- [Manager](#) (const [CityGraph](#) &cityGraph, const [CityMap](#) &[CityMap](#), bool log)
Constructor.
- [Manager](#) (const [CityGraph](#) &cityGraph, const [CityMap](#) &[CityMap](#), std::vector< [Car](#) > cars, bool log)
Constructor.
- void [createCarsAStar](#) (int numCars)
Create cars using A pathfinding, no collision avoidance.*
- std::pair< bool, [DataManager::data](#) > [createCarsCBS](#) (int numCars)
Create cars using CBS pathfinding.
- [CBSNode](#) [createSubCBS](#) ([CBSNode](#) &node, int subNodeDepth)
Create a sub-CBS node.
- [CBSNode](#) [processCBS](#) ([ConstraintController](#) constraints, int subNodeDepth)
Process a CBS node.
- bool [hasConflict](#) (std::vector< std::vector< sf::Vector2f > > paths, int *car1, int *car2, sf::Vector2f *p1, sf::Vector2f *p2, double *a1, double *a2, int *time)
Check if two cars have a conflict.
- void [moveCars](#) ()

- Move the cars to the next point in the path.*
- void `renderCars` (sf::RenderWindow &>window)
- Render the cars.*
- void `toggleCarDebug` (sf::Vector2f mousePos)
- Toggle the debug of one car.*
- int `getNumCars` ()
- Get the number of cars.*
- std::vector< `Car` > `getCars` ()
- Get the cars.*

3.22.1 Detailed Description

A manager for the cars.

The manager class is used to manage the cars during the CBS pathfinding. It creates the cars and resolves conflicts using the CBS algorithm.

Definition at line 45 of file [manager.h](#).

3.22.2 Constructor & Destructor Documentation

Manager() [1/2]

```
Manager::Manager (
    const CityGraph & cityGraph,
    const CityMap & CityMap,
    bool log) [inline]
```

Constructor.

Parameters

<i>cityGraph</i>	The city graph
<i>CityMap</i>	The city map
<i>log</i>	If the manager should log

Definition at line 55 of file [manager.h](#).

Manager() [2/2]

```
Manager::Manager (
    const CityGraph & cityGraph,
    const CityMap & CityMap,
    std::vector< Car > cars,
    bool log) [inline]
```

Constructor.

Parameters

<i>cityGraph</i>	The city graph
<i>CityMap</i>	The city map
<i>cars</i>	The cars
<i>log</i>	If the manager should log

Definition at line 66 of file [manager.h](#).

3.22.3 Member Function Documentation

createCarsAStar()

```
void Manager::createCarsAStar (  
    int numCars)
```

Create cars using A* pathfinding, no collision avoidance.

Parameters

<i>numCars</i>	The number of cars
----------------	--------------------

Definition at line 13 of file [manager.cpp](#).

createCarsCBS()

```
std::pair< bool, DataManager::data > Manager::createCarsCBS (  
    int numCars)
```

Create cars using CBS pathfinding.

Parameters

<i>numCars</i>	The number of cars
----------------	--------------------

Returns

The data for the cars (success, data)

Definition at line 16 of file [managerCBS.cpp](#).

createSubCBS()

```
Manager::CBSNode Manager::createSubCBS (  
    CBSNode & node,  
    int subNodeDepth)
```

Create a sub-CBS node.

Parameters

<i>node</i>	The parent CBS node
<i>subNodeDepth</i>	The depth of the sub-CBS node

Returns

The sub-CBS node

This function creates a sub-CBS node from a parent CBS node. It creates a new node with the same paths and constraints as the parent node, but with less agents.

Definition at line 90 of file [managerCBS.cpp](#).

getCars()

```
std::vector< Car > Manager::getCars () [inline]
```

Get the cars.

Returns

The cars

Definition at line 150 of file [manager.h](#).

getNumCars()

```
int Manager::getNumCars () [inline]
```

Get the number of cars.

Returns

The number of cars

Definition at line 144 of file [manager.h](#).

hasConflict()

```
bool Manager::hasConflict (
    std::vector< std::vector< sf::Vector2f > > paths,
    int * car1,
    int * car2,
    sf::Vector2f * p1,
    sf::Vector2f * p2,
    double * a1,
    double * a2,
    int * time)
```

Check if two cars have a conflict.

Parameters

<i>paths</i>	The paths of the cars
<i>car1</i>	The first car
<i>car2</i>	The second car
<i>p1</i>	The position of the first car
<i>p2</i>	The position of the second car
<i>a1</i>	The angle of the first car
<i>a2</i>	The angle of the second car
<i>time</i>	The time of the conflict

Returns

If the cars have a conflict

Definition at line 325 of file [managerCBS.cpp](#).

processCBS()

```
Manager::CBSNode Manager::processCBS (
    ConstraintController constraints,
    int subNodeDepth)
```

Process a CBS node.

Parameters

<i>constraints</i>	The constraints
<i>subNodeDepth</i>	The depth of the sub-CBS node

Returns

The processed CBS node

This function processes a CBS node. It resolves conflicts and returns a new CBS node with the resolved conflicts.
Definition at line 167 of file [managerCBS.cpp](#).

renderCars()

```
void Manager::renderCars (
    sf::RenderWindow & window)
```

Render the cars.

Parameters

<i>window</i>	The window
---------------	------------

Definition at line 37 of file [manager.cpp](#).

toggleCarDebug()

```
void Manager::toggleCarDebug (
    sf::Vector2f mousePos)
```

Toggle the debug of one car.

Parameters

<i>mousePos</i>	The mouse position
-----------------	--------------------

This function toggles the debug of a car. If the mouse is over a car, the debug of the car is toggled.

Definition at line 43 of file [manager.cpp](#).

The documentation for this class was generated from the following files:

- [manager.h](#)
- [manager.cpp](#)
- [managerCBS.cpp](#)

3.23 **PriorityQueue< T > Class Template Reference**

Priority Queue.

```
#include <priorityQueue.h>
```

Public Member Functions

- [PriorityQueue](#) (int size)
Constructor.
- [~PriorityQueue](#) ()
Destructor.
- void [push](#) (T e, double p)
Push an element with a priority.
- T [pop](#) ()
Pop the element with the lowest priority.
- bool [empty](#) ()
Get the element with the lowest priority.

3.23.1 Detailed Description

```
template<class T>
```

```
class PriorityQueue< T >
```

Priority Queue.

Template Parameters

<i>T</i>	Type of the elements
----------	----------------------

This class is a simple priority queue implementation. With a fixed size, it will keep the elements sorted by their priority. The elements with the lowest priority will be at the front of the queue.

Definition at line 20 of file [priorityQueue.h](#).

3.23.2 Constructor & Destructor Documentation

PriorityQueue()

```
template<class T>
PriorityQueue< T >::PriorityQueue (
    int size) [inline]
```

Constructor.

Parameters

<i>size</i>	The size of the queue
-------------	-----------------------

Definition at line 26 of file [priorityQueue.h](#).

3.23.3 Member Function Documentation

empty()

```
template<class T>
bool PriorityQueue< T >::empty () [inline]
```

Get the element with the lowest priority.

Returns

The element

Definition at line 88 of file [priorityQueue.h](#).

pop()

```
template<class T>
T PriorityQueue< T >::pop () [inline]
```

Pop the element with the lowest priority.

Returns

The element

Definition at line 69 of file [priorityQueue.h](#).

push()

```
template<class T>
void PriorityQueue< T >::push (
    T e,
    double p) [inline]
```

Push an element with a priority.

Parameters

<i>e</i>	The element
<i>p</i>	The priority

Definition at line 44 of file [priorityQueue.h](#).

The documentation for this class was generated from the following file:

- [priorityQueue.h](#)

3.24 Renderer Class Reference

A renderer for the city.

```
#include <renderer.h>
```

Public Member Functions

- void **startRender** (const [CityMap](#) &cityMap, const [CityGraph](#) &cityGraph, [Manager](#) &manager)
Start the rendering.
- void **renderCityMap** (const [CityMap](#) &cityMap)
Render the city map.
- void **renderCityGraph** (const [CityGraph](#) &cityGraph, const sf::View &view)
Render the city graph.
- void **renderManager** ([Manager](#) &manager)
Render the cars.
- void **renderTime** ()
Render the time.
- void **setConflicts** (const std::vector< [AStar::conflict](#) > &conflicts)
Render the conflicts.

3.24.1 Detailed Description

A renderer for the city.

The renderer class is used to render the city map, the city graph and the cars.

Definition at line 19 of file [renderer.h](#).

3.24.2 Member Function Documentation

renderCityGraph()

```
void Renderer::renderCityGraph (
    const CityGraph & cityGraph,
    const sf::View & view)
```

Render the city graph.

Parameters

<i>cityGraph</i>	The city graph
<i>view</i>	The view

Definition at line 250 of file [renderer.cpp](#).

renderCityMap()

```
void Renderer::renderCityMap (
    const CityMap & cityMap)
```

Render the city map.

Parameters

<i>cityMap</i>	The city map
----------------	--------------

Definition at line 147 of file [renderer.cpp](#).

renderManager()

```
void Renderer::renderManager (
    Manager & manager)
```

Render the cars.

Parameters

<i>manager</i>	The manager
----------------	-------------

Definition at line 341 of file [renderer.cpp](#).

The documentation for this class was generated from the following files:

- [renderer.h](#)
- [renderer.cpp](#)

3.25 Test Class Reference

A class for testing the project.

```
#include <test.h>
```

Public Member Functions

- void **runTests** ()
Run the tests.

3.25.1 Detailed Description

A class for testing the project.

This class is used to test the project.

Definition at line 13 of file [test.h](#).

The documentation for this class was generated from the following files:

- [test.h](#)
- [test.cpp](#)

3.26 TimedAStar Class Reference

Timed A* algorithm.

```
#include <aStar.h>
```

Public Member Functions

- **TimedAStar** ([CityGraph::point](#) start, [CityGraph::point](#) end, const [CityGraph](#) &cityGraph, [ConstraintController](#) *constraints, int carIndex)
Constructor.
- std::vector< [AStar::node](#) > **findPath** ()
Find the path.

3.26.1 Detailed Description

Timed A* algorithm.

This class represents the timed A* algorithm. It is used to find the shortest path between two points in a graph while taking into account the constraints of the cars.

Definition at line 171 of file [aStar.h](#).

3.26.2 Constructor & Destructor Documentation

TimedAStar()

```
TimedAStar::TimedAStar (
    CityGraph::point start,
    CityGraph::point end,
    const CityGraph & cityGraph,
```

```
ConstraintController * constraints,  
int carIndex)
```

Constructor.

Parameters

<i>start</i>	The start point
<i>end</i>	The end point
<i>cityGraph</i>	The graph
<i>constraints</i>	The constraints
<i>carIndex</i>	The car index

Definition at line 16 of file [timedAStar.cpp](#).

3.26.3 Member Function Documentation**findPath()**

```
std::vector< AStar::node > TimedAStar::findPath () [inline]
```

Find the path.

Returns

The path

Definition at line 188 of file [aStar.h](#).

The documentation for this class was generated from the following files:

- [aStar.h](#)
- [timedAStar.cpp](#)

4 File Documentation**4.1 aStar.h File Reference**

A* algorithm.

```
#include "cityGraph.h"
```

Classes

- struct [_aStarNode](#)
A node for the A algorithm.*
- struct [_aStarConflict](#)
A conflict for the A algorithm.*
- class [AStar](#)
A algorithm.*
- class [ConstraintController](#)
Controller for constraints.
- class [TimedAStar](#)
Timed A algorithm.*

4.1.1 Detailed Description

A* algorithm.

This file contains the A* algorithm. It is used to find the shortest path between two points in a graph. It also contains the timed A* algorithm, which is used to find the shortest path between two points in a graph while taking into account the constraints of the cars.

Definition in file [aStar.h](#).

4.2 aStar.h

[Go to the documentation of this file.](#)

```

00001
00009 #pragma once
00010
00011 #include "cityGraph.h"
00012
00020 typedef struct _aStarNode {
00021     CityGraph::point point;
00022     double speed;
00023     std::pair<CityGraph::point, CityGraph::neighbor> arcFrom;
00024
00025     bool operator==(const _aStarNode &other) const {
00026         double s = std::round(speed / SPEED_RESOLUTION);
00027         double oS = std::round(other.speed / SPEED_RESOLUTION);
00028
00029         return point == other.point && s == oS && arcFrom.first == other.arcFrom.first &&
00030             arcFrom.second == other.arcFrom.second;
00031     }
00032 } _aStarNode;
00033
00041 typedef struct _aStarConflict {
00042     CityGraph::point point;
00043     int time;
00044     int car;
00045
00046     bool operator==(const _aStarConflict &other) const {
00047         return point == other.point && time == other.time && car == other.car;
00048     }
00049 } _aStarConflict;
00050
00051 namespace std {
00052     template <> struct hash<_aStarNode> {
00053         std::size_t operator()(const _aStarNode &point) const {
00054             double s = std::round(point.speed / SPEED_RESOLUTION);
00055
00056             return std::hash<CityGraph::point>()(point.point) ^ std::hash<double>()(s) ^
00057                 std::hash<CityGraph::point>()(point.arcFrom.first) ^
00058                 std::hash<CityGraph::neighbor>()(point.arcFrom.second);
00059         };
00060     template <> struct hash<_aStarConflict> {
00061         std::size_t operator()(const _aStarConflict &conflict) const {
00062             return std::hash<CityGraph::point>()(conflict.point) ^ std::hash<int>()(conflict.time) ^
00063                 std::hash<int>()(conflict.car);
00064         };
00065     };
00066 } // namespace std
00067
00074 class AStar {
00075 public:
00076     using node = _aStarNode;
00077     using conflict = _aStarConflict;
00078
00085     AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph);
00086
00091     std::vector<node> findPath() {
00092         if (!processed)
00093             process();
00094         return path;
00095     }
00096
00097 private:
00098     bool processed = false;
00099     node start;
00100     node end;
00101     std::vector<node> path;
00102     CityGraph graph;
00103
00104     void process();
00105 };
00106
00114 class ConstraintController {
00115 public:
00119     ConstraintController() { this->constraints.clear(); }
00120
00125     ConstraintController copy();
00126
00132     ConstraintController copy(std::vector<int> cars);
00133
00138     void addConstraint(AStar::conflict constraints);
00139
00145     bool hasConstraint(AStar::conflict constraint);
00146
00157     bool checkConstraints(int car, double speed, double newSpeed, double time, CityGraph::point from,

```



```

00158             CityGraph::neighbor to);
00159
00160 private:
00161     std::vector<std::vector<std::vector<AStar::conflict>> constraints; // [car][time][constraints]
00162 };
00163
00171 class TimedAStar {
00172 public:
00181     TimedAStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph,
00182               ConstraintController *constraints, int carIndex);
00183
00188     std::vector<AStar::node> findPath() {
00189         if (!processed)
00190             process();
00191         return path;
00192     }
00193
00194 private:
00195     bool processed = false;
00196     AStar::node start;
00197     AStar::node end;
00198     std::vector<AStar::node> path;
00199     ConstraintController *conflicts;
00200     int carIndex;
00201     CityGraph graph;
00202
00203     void process();
00204 };

```

4.3 car.h File Reference

A car in the city.

```

#include <SFML/Graphics.hpp>
#include <vector>
#include "aStar.h"
#include "cityGraph.h"

```

Classes

- class [Car](#)

A car in the city.

4.3.1 Detailed Description

A car in the city.

This file contains the declaration of the [Car](#) class. This class represents a car in the city. It contains the start and end points of the car, the path of the car and the current point in the path.

Definition in file [car.h](#).

4.4 car.h

[Go to the documentation of this file.](#)

```

00001
00008 #pragma once
00009
00010 #include <SFML/Graphics.hpp>
00011 #include <vector>
00012
00013 #include "aStar.h"
00014 #include "cityGraph.h"
00015
00023 class Car {
00024 public:
00028     Car();
00029
00035     void assignStartEnd(CityGraph::point start, CityGraph::point end) {
00036         this->start = start;
00037         this->end = end;
00038     }
00039
00045     void chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap);
00046
00051     void assignPath(std::vector<AStar::node> path);

```

```

00052
00057 void assignExistingPath(std::vector<sf::Vector2f> path);
00058
00062 void move();
00063
00068 void render(sf::RenderWindow &window);
00069
00074 CityGraph::point getStart() { return start; }
00075
00080 CityGraph::point getEnd() { return end; }
00081
00086 double getSpeed();
00087
00093 double getSpeedAt(int index);
00094
00100 double getAverageSpeed(CityGraph &graph);
00101
00106 double getRemainingTime();
00107
00112 double getElapsedTime();
00113
00118 double getPathTime();
00119
00124 double getRemainingDistance();
00125
00130 double getElapsedDistance();
00131
00136 double getPathLength();
00137
00142 sf::Vector2f getPosition() { return path[currentPoint]; }
00143
00148 std::vector<sf::Vector2f> getPath() { return path; }
00149
00154 std::vector<AStar::node> getAStarPath() { return aStarPath; }
00155
00160 void toggleDebug() { debug = !debug; }
00161
00162 private:
00163 CityGraph::point start;
00164 CityGraph::point end;
00165 std::vector<sf::Vector2f> path;
00166 std::vector<AStar::node> aStarPath;
00167 int currentPoint = 0;
00168 bool debug = false;
00169 sf::Color color;
00170 };

```

4.5 cityGraph.h File Reference

A graph representing the city's streets and intersections using a graph.

```

#include <unordered_set>
#include "cityMap.h"
#include "config.h"
#include "utils.h"

```

Classes

- struct [_cityGraphPoint](#)
A point in the city graph.
- struct [_cityGraphNeighbor](#)
A neighbor of a point in the city graph.
- class [CityGraph](#)
A graph representing the city's streets and intersections using a graph.

4.5.1 Detailed Description

A graph representing the city's streets and intersections using a graph.

This file contains the definition of the [CityGraph](#) class.

Definition in file [cityGraph.h](#).

4.6 cityGraph.h

[Go to the documentation of this file.](#)

```

00001
00007 #pragma once
00008 #include <unordered_set>
00009
00010 #include "cityMap.h"
00011 #include "config.h"
00012 #include "utils.h"
00013
00020 typedef struct _cityGraphPoint {
00021     sf::Vector2f position;
00022     double angle;
00023
00024     bool operator==(const _cityGraphPoint &other) const {
00025         int x = std::round(position.x / CELL_SIZE);
00026         int y = std::round(position.y / CELL_SIZE);
00027         int a = std::round(normalizeAngle(angle) / ANGLE_RESOLUTION);
00028         int oX = std::round(other.position.x / CELL_SIZE);
00029         int oY = std::round(other.position.y / CELL_SIZE);
00030         int oA = std::round(normalizeAngle(other.angle) / ANGLE_RESOLUTION);
00031
00032         return x == oX && y == oY && a == oA;
00033     }
00034 } _cityGraphPoint;
00035
00043 typedef struct _cityGraphNeighbor {
00044     _cityGraphPoint point;
00045     double maxSpeed;
00046     double turningRadius;
00047     double distance;
00048     bool isRightWay;
00049
00050     bool operator==(const _cityGraphNeighbor &other) const {
00051         return point == other.point && maxSpeed == other.maxSpeed && turningRadius == other.turningRadius
00052         && distance == other.distance && isRightWay == other.isRightWay;
00053     }
00054 } _cityGraphNeighbor;
00055
00056 namespace std {
00057     template <> struct hash<_cityGraphPoint> {
00058         std::size_t operator()(const _cityGraphPoint &point) const {
00059             int x = std::round(point.position.x / CELL_SIZE);
00060             int y = std::round(point.position.y / CELL_SIZE);
00061             int a = std::round(normalizeAngle(point.angle) / ANGLE_RESOLUTION);
00062
00063             return std::hash<int>()(x) ^ std::hash<int>()(y) ^ std::hash<int>()(a);
00064         }
00065     };
00066
00067     template <> struct hash<_cityGraphNeighbor> {
00068         std::size_t operator()(const _cityGraphNeighbor &neighbor) const {
00069             return std::hash<_cityGraphPoint>()(neighbor.point) ^ std::hash<double>()(neighbor.maxSpeed) ^
00070             std::hash<double>()(neighbor.turningRadius) ^ std::hash<double>()(neighbor.distance) ^
00071             std::hash<bool>()(neighbor.isRightWay);
00072         }
00073     };
00074 } // namespace std
00075
00082 class CityGraph {
00083 public:
00084     using point = _cityGraphPoint;
00085     using neighbor = _cityGraphNeighbor;
00086
00094     void createGraph(const CityMap &cityMap);
00095
00100     std::unordered_map<point, std::vector<neighbor>> getNeighbors() const { return neighbors; }
00101
00106     std::unordered_set<point> getGraphPoints() const { return graphPoints; }
00107
00112     point getRandomPoint() const;
00113
00118     double getHeight() const { return height; }
00119
00124     double getWidth() const { return width; }
00125
00126 private:
00127     std::unordered_map<point, std::vector<neighbor>> neighbors;
00128     std::unordered_set<point> graphPoints;
00129
00130     void linkPoints(const point &point1, const point &point2, int direction,
00131                    bool subPoints); // direction: 0 -> point1 to point2, 1 -> point2 to point1, 2 ->
00132     both
00133     bool canLink(const point &point1, const point &point2, double speed, double *distance) const;

```

```

00133
00134     double width;
00135     double height;
00136 };

```

4.7 cityMap.h

```

00001 #pragma once
00002
00003 #include <SFML/Graphics.hpp>
00004 #include <string>
00005 #include <vector>
00006
00011 typedef struct {
00012     sf::Vector2f p1;
00013     sf::Vector2f p2;
00014     sf::Vector2f p1_offset;
00015     sf::Vector2f p2_offset;
00016     double angle;
00017 } _cityMapSegment;
00018
00023 typedef struct {
00024     int id;
00025     std::vector<_cityMapSegment> segments;
00026     double width;
00027     int numLanes;
00028 } _cityMapRoad;
00029
00034 typedef struct {
00035     std::vector<sf::Vector2f> points;
00036 } _cityMapBuilding;
00037
00042 typedef struct {
00043     std::vector<sf::Vector2f> points;
00044     int type;
00045 } _cityMapGreenArea;
00046
00051 typedef struct {
00052     std::vector<sf::Vector2f> points;
00053 } _cityMapWaterArea;
00054
00059 typedef struct {
00060     int id;
00061     sf::Vector2f center;
00062     double radius;
00063     std::vector<std::pair<int, int> roadSegmentIds;
00064 } _cityMapIntersection;
00065
00074 class CityMap {
00075 public:
00076     using segment = _cityMapSegment;
00077     using road = _cityMapRoad;
00078     using building = _cityMapBuilding;
00079     using greenArea = _cityMapGreenArea;
00080     using waterArea = _cityMapWaterArea;
00081     using intersection = _cityMapIntersection;
00082
00086     CityMap();
00087
00092     void loadFile(const std::string &filename);
00093
00098     bool isCityMapLoaded() const { return isLoaded; }
00099
00104     std::vector<road> getRoads() const { return roads; }
00105
00110     std::vector<intersection> getIntersections() const { return intersections; }
00111
00116     std::vector<building> getBuildings() const { return buildings; }
00117
00122     std::vector<greenArea> getGreenAreas() const { return greenAreas; }
00123
00128     std::vector<waterArea> getWaterAreas() const { return waterAreas; }
00129
00134     sf::Vector2f getMinLatLon() const { return minLatLon; }
00135
00140     sf::Vector2f getMaxLatLon() const { return maxLatLon; }
00141
00146     int getWidth() const { return width; }
00147
00152     int getHeight() const { return height; }
00153
00154 private:
00155     bool isLoaded = false;
00156
00157     std::vector<road> roads;

```

```

00158     std::vector<intersection> intersections;
00159     std::vector<building> buildings;
00160     std::vector<greenArea> greenAreas;
00161     std::vector<waterArea> waterAreas;
00162
00163     sf::Vector2f minLatLon;
00164     sf::Vector2f maxLatLon;
00165     double width; // in meters
00166     double height; // in meters
00167 };

```

4.8 config.h File Reference

Configuration file.

```
#include <string>
```

4.8.1 Detailed Description

Configuration file.

Definition in file [config.h](#).

4.9 config.h

[Go to the documentation of this file.](#)

```

00001
00005 #pragma once
00006
00007 #include <string>
00008
00009 constexpr int ENVIRONMENT = 0; // 0 = development, 1 = production
00010 constexpr int SCREEN_WIDTH = 2880;
00011 constexpr int SCREEN_HEIGHT = 1864;
00012 constexpr double LOG_CBS_REFRESH_RATE = 0.3; // in seconds
00013
00014 constexpr int EARTH_RADIUS = 6371000; // in meters
00015
00016 constexpr double DEFAULT_ROAD_WIDTH = 7.0; // in meters
00017 constexpr double DEFAULT_LANE_WIDTH = 3.5; // in meters
00018 constexpr double MIN_ROAD_WIDTH = 4.0; // in meters
00019 constexpr bool ROAD_ENABLE_RIGHT_HAND_TRAFFIC = false;
00020
00021 constexpr double ZOOM_SPEED = 0.1;
00022 constexpr double MOVE_SPEED = 0.01;
00023
00024 constexpr double SIM_STEP_TIME = 0.15; // in seconds
00025 constexpr int CBS_PRECISION_FACTOR = 1; // CBS_PRECISION_FACTOR * SIM_STEP_TIME must not be too high
00026 constexpr double CBS_MAX_SUB_TIME = 30; // in seconds
00027 constexpr double CBS_MAX_OPENSET_SIZE = 5;
00028
00029 // For hash functions (to reduce items that are really close to each other)
00030 constexpr double CELL_SIZE = 0.5; // in meters
00031 constexpr double SPEED_RESOLUTION = 0.3; // in m/s
00032 constexpr double ANGLE_RESOLUTION = 0.1; // in radians
00033 constexpr double TIME_RESOLUTION = SIM_STEP_TIME; // in seconds
00034
00035 constexpr double CAR_MIN_TURNING_RADIUS = 1.5; // in meters
00036 constexpr double CAR_MAX_SPEED_KM = 30.0; // in km/h
00037 constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6; // in m/s
00038 constexpr double CAR_MAX_G_FORCE = 5.0; // in m/s^2
00039 constexpr double CAR_ACCELERATION = 3.0; // in m/s^2
00040 constexpr double CAR_DECELERATION = 4.0; // in m/s^2
00041 constexpr double CAR_LENGTH = 4.2; // in meters
00042 constexpr double CAR_WIDTH = 1.6; // in meters

```

4.10 dataManager.h File Reference

Data manager.

```
#include <string>
```

```
#include <vector>
```

Classes

- struct [_data](#)
Data structure.
- class [DataManager](#)
Data manager.

4.10.1 Detailed Description

Data manager.

This file contains the data manager class.

Definition in file [dataManager.h](#).

4.11 [dataManager.h](#)

[Go to the documentation of this file.](#)

```
00001
00007 #pragma once
00008
00009 #include <string>
00010 #include <vector>
00011
00018 struct _data {
00019     double numCars;
00020     double carDensity;
00021     std::vector<double> carAvgSpeed;
00022 };
00023
00030 class DataManager {
00031 public:
00032     using data = _data;
00033
00038     DataManager(std::string filename);
00039
00048     void createData(int numData, int numCarsMin, int numCarsMax, std::string mapName);
00049
00050 private:
00051 };
```

4.12 [dubins.h](#) File Reference

[Dubins](#) path.

```
#include "aStar.h"
#include "cityGraph.h"
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
```

Classes

- class [Dubins](#)
[Dubins](#) path used to calculate the path between two points in the city graph.
- class [DubinsPath](#)
[Dubins](#) path used to calculate the path between two points in the city graph.

4.12.1 Detailed Description

[Dubins](#) path.

This file contains the [Dubins](#) class. It is used to calculate the path between two points in the city graph. It will be used to render cars in the city and check for collisions.

Definition in file [dubins.h](#).

4.13 dubins.h

[Go to the documentation of this file.](#)

```

00001
00008 #pragma once
00009
00010 #include "aStar.h"
00011 #include "cityGraph.h"
00012
00013 #include <ompl/base/State.h>
00014 #include <ompl/base/StateSpace.h>
00015 #include <ompl/base/spaces/DubinsStateSpace.h>
00016
00017 namespace ob = ompl::base;
00018
00026 class Dubins {
00027 public:
00036     Dubins(CityGraph::point start, CityGraph::neighbor end);
00037
00048     Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed);
00049
00061     Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed, double endSpeed);
00062
00066     ~Dubins();
00067
00072     double distance() { return endPoint.distance; }
00073
00078     double time();
00079
00085     CityGraph::point point(double time);
00086
00091     std::vector<CityGraph::point> path();
00092
00093 private:
00094     ob::DubinsStateSpace *space;
00095     ob::State *start;
00096     ob::State *end;
00097
00098     CityGraph::point startPoint;
00099     CityGraph::neighbor endPoint;
00100     double startSpeed;
00101     double endSpeed;
00102     double avgSpeed;
00103 };
00104
00112 class DubinsPath {
00113 public:
00121     DubinsPath(std::vector<AStar::node> path);
00122
00126     std::vector<CityGraph::point> path();
00127
00128 private:
00129     void process();
00130
00131     std::vector<AStar::node> path_;
00132     std::vector<CityGraph::point> pathProcessed_;
00133 };

```

4.14 fileSelector.h File Reference

File selector.

```

#include <iostream>
#include <termios.h>
#include <unistd.h>
#include <vector>

```

Classes

- class [FileSelector](#)
A file selector.

4.14.1 Detailed Description

File selector.

This file contains the [FileSelector](#) class. It is used to select a file from a folder.

Definition in file [fileSelector.h](#).

4.15 fileSelector.h

[Go to the documentation of this file.](#)

```

00001
00007 #pragma once
00008 #include <iostream>
00009 #include <termios.h>
00010 #include <unistd.h>
00011 #include <vector>
00012
00019 class FileSelector {
00020 private:
00021     std::string folderPath;
00022     std::vector<std::string> files;
00023     int selectedIndex;
00024
00025     void loadFiles();
00026     char getKeyPress();
00027     void moveCursorUp();
00028     void moveCursorDown();
00029     void displayFiles();
00030
00031 public:
00032     FileSelector(const std::string &path) : folderPath(path), selectedIndex(0) { loadFiles(); }
00033     ~FileSelector() { std::cout << "\033[?25h"; }
00034
00035     std::string selectFile();
00036 };

```

4.16 manager.h File Reference

[Manager](#) for the cars.

```

#include <SFML/Graphics.hpp>
#include <vector>
#include "car.h"
#include "cityGraph.h"
#include "dataManager.h"

```

Classes

- struct [_managerCBSNode](#)
A node for the CBS algorithm.
- class [Manager](#)
A manager for the cars.

4.16.1 Detailed Description

[Manager](#) for the cars.

This file contains the declaration of the [Manager](#) class. This class is used to manage the cars during the CBS pathfinding. It creates the cars and resolves conflicts using the CBS algorithm.

Definition in file [manager.h](#).

4.17 manager.h

[Go to the documentation of this file.](#)

```

00001
00008 #pragma once
00009
00010 #include <SFML/Graphics.hpp>
00011 #include <vector>
00012
00013 #include "car.h"
00014 #include "cityGraph.h"
00015 #include "dataManager.h"
00016
00024 typedef struct _managerCBSNode {
00025     std::vector<std::vector<sf::Vector2f>> paths;
00026     ConstraintController constraints;
00027     std::vector<double> costs;
00028     double cost;

```



```

00029     int depth;
00030     bool hasResolved;
00031
00032     bool operator<(const _managerCBSNode &other) const {
00033         return cost > other.cost || (cost == other.cost && depth > other.depth);
00034     }
00035
00036 } _managerCBSNode;
00037
00045 class Manager {
00046 public:
00047     using CBSNode = _managerCBSNode;
00048
00055     Manager(const CityGraph &cityGraph, const CityMap &CityMap, bool log) : graph(cityGraph),
map(CityMap) {
00056         this->log = log;
00057     }
00058
00066     Manager(const CityGraph &cityGraph, const CityMap &CityMap, std::vector<Car> cars, bool log)
: graph(cityGraph), map(CityMap), cars(cars) {
00067         this->numCars = cars.size();
00068         this->log = log;
00069     }
00070
00071
00076     void createCarsAStar(int numCars);
00077
00083     std::pair<bool, DataManager::data> createCarsCBS(int numCars);
00084
00094     CBSNode createSubCBS(CBSNode &node, int subNodeDepth);
00095
00104     CBSNode processCBS(ConstraintController constraints, int subNodeDepth);
00105
00118     bool hasConflict(std::vector<std::vector<sf::Vector2f> paths, int *car1, int *car2, sf::Vector2f
*pl,
00119                     sf::Vector2f *p2, double *a1, double *a2, int *time);
00120
00124     void moveCars();
00125
00130     void renderCars(sf::RenderWindow &window);
00131
00138     void toggleCarDebug(sf::Vector2f mousePos);
00139
00144     int getNumCars() { return numCars; }
00145
00150     std::vector<Car> getCars() { return cars; }
00151
00152 private:
00153     int numCars;
00154     std::vector<Car> cars;
00155     CityGraph graph;
00156     CityMap map;
00157     bool log;
00158 };

```

4.18 priorityQueue.h File Reference

Priority Queue.

```
#include <iostream>
```

Classes

- class [PriorityQueue< T >](#)
Priority Queue.

4.18.1 Detailed Description

Priority Queue.

This file contains the declaration of the [PriorityQueue](#) class. This class is a simple priority queue implementation. With a fixed size, it will keep the elements sorted by their priority. The elements with the lowest priority will be at the front of the queue.

Definition in file [priorityQueue.h](#).

4.19 priorityQueue.h

[Go to the documentation of this file.](#)

```

00001
00009 #pragma once
00010 #include <iostream>
00011
00020 template <class T> class PriorityQueue {
00021 public:
00026   PriorityQueue(int size) {
00027       this->size = size;
00028       elements = (T *)malloc(size * sizeof(T));
00029       priorities = (double *)malloc(size * sizeof(double));
00030   }
00034   ~PriorityQueue() {
00035       free(elements);
00036       free(priorities);
00037   }
00038
00044   void push(T e, double p) {
00045       if (count < size) {
00046           elements[count] = e;
00047           priorities[count] = p;
00048           count++;
00049           return;
00050       }
00051
00052       for (int i = 0; i < size; i++) {
00053           if (p < priorities[i]) {
00054               for (int j = size - 1; j > i; j--) {
00055                   elements[j] = elements[j - 1];
00056                   priorities[j] = priorities[j - 1];
00057               }
00058               elements[i] = e;
00059               priorities[i] = p;
00060               return;
00061           }
00062       }
00063   }
00064
00069   T pop() {
00070       if (count == 0) {
00071           std::cerr « "PriorityQueue is empty" « std::endl;
00072           exit(1);
00073       }
00074
00075       T e = elements[0];
00076       for (int i = 1; i < size; i++) {
00077           elements[i - 1] = elements[i];
00078           priorities[i - 1] = priorities[i];
00079       }
00080       count--;
00081       return e;
00082   }
00083
00088   bool empty() { return count == 0; }
00089
00090 private:
00091   T *elements;
00092   double *priorities;
00093   int size;
00094   int count;
00095 };

```

4.20 renderer.h File Reference

A renderer for the city.

```

#include <SFML/Graphics.hpp>
#include "cityGraph.h"
#include "cityMap.h"
#include "manager.h"

```

Classes

- class [Renderer](#)

A renderer for the city.

Functions

- void [drawArrow](#) (sf::RenderWindow &window, sf::Vector2f position, double rotation, double length, double thickness, sf::Color color=sf::Color::Red, bool outline=false)

Draw an arrow.

4.20.1 Detailed Description

A renderer for the city.

Definition in file [renderer.h](#).

4.20.2 Function Documentation

drawArrow()

```
void drawArrow (
    sf::RenderWindow & window,
    sf::Vector2f position,
    double rotation,
    double length,
    double thickness,
    sf::Color color = sf::Color::Red,
    bool outline = false) [inline]
```

Draw an arrow.

Parameters

<i>window</i>	The window
<i>position</i>	The position
<i>rotation</i>	The rotation
<i>length</i>	The length
<i>thickness</i>	The thickness
<i>color</i>	The color
<i>outline</i>	If the arrow should have an outline

Definition at line 74 of file [renderer.h](#).

4.21 renderer.h

[Go to the documentation of this file.](#)

```
00001
00005 #pragma once
00006
00007 #include <SFML/Graphics.hpp>
00008
00009 #include "cityGraph.h"
00010 #include "cityMap.h"
00011 #include "manager.h"
00012
00019 class Renderer {
00020 public:
00024     void startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager);
00025
00030     void renderCityMap(const CityMap &cityMap);
00031
00037     void renderCityGraph(const CityGraph &cityGraph, const sf::View &view);
00038
00043     void renderManager(Manager &manager);
00044
00048     void renderTime();
00049
00053     void setConflicts(const std::vector<AStar::conflict> &conflicts) { this->conflicts = conflicts; }
00054
00055 private:
00056     sf::RenderWindow window;
00057     double time;
00058
```

```

00059     std::vector<AStar::conflict> conflicts;
00060
00061     bool debug = false;
00062 };
00063
00074 inline void drawArrow(sf::RenderWindow &window, sf::Vector2f position, double rotation, double length,
double thickness,
00075                     sf::Color color = sf::Color::Red, bool outline = false) {
00076     sf::ConvexShape arrow;
00077
00078     arrow.setFillColor(color);
00079     arrow.setOrigin(-length / 2, 0);
00080     arrow.setPosition(position);
00081     arrow.setRotation(rotation);
00082
00083     arrow.setPointCount(7);
00084     arrow.setPoint(0, sf::Vector2f(0, 0));
00085     arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
00086     arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
00087     arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
00088     arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
00089     arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
00090     arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));
00091
00092     if (outline) {
00093         arrow.setOutlineThickness(thickness / 10);
00094         arrow.setOutlineColor(sf::Color::Black);
00095     }
00096
00097     window.draw(arrow);
00098 }

```

4.22 test.h File Reference

A header file for the [Test](#) class.

Classes

- class [Test](#)
A class for testing the project.

4.22.1 Detailed Description

A header file for the [Test](#) class.

Definition in file [test.h](#).

4.23 test.h

[Go to the documentation of this file.](#)

```

00001
00005 #pragma once
00006
00013 class Test {
00014 public:
00018     void runTests();
00019
00020 private:
00021     void testSpdlog();
00022     void testTinyXML2();
00023     void testSFML();
00024 };

```

4.24 utils.h File Reference

Utility functions.

```

#include "config.h"
#include <SFML/Graphics.hpp>

```

Functions

- sf::Vector2f [latLonToXY](#) (double lat, double lon)

- Convert latitude and longitude to x and y.*
- double [distance](#) (sf::Vector2f p1, sf::Vector2f p2)
Get the distance between two points.
- double [normalizeAngle](#) (double angle)
Normalize an angle to -PI to PI.
- double [turningRadius](#) (double speed)
Get the turning radius from the speed.
- double [turningRadiusToSpeed](#) (double radius)
Get the speed from the turning radius.
- bool [carsCollided](#) (Car car1, Car car2, int time)
- bool [carConflict](#) (sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle)
Check if two cars have a conflict.
- sf::Font [loadFont](#) ()
Load a font.

4.24.1 Detailed Description

Utility functions.

Definition in file [utils.h](#).

4.24.2 Function Documentation

carConflict()

```
bool carConflict (
    sf::Vector2f carPos,
    double carAngle,
    sf::Vector2f confPos,
    double confAngle)
```

Check if two cars have a conflict.

Parameters

<i>carPos</i>	The position of the car
<i>carAngle</i>	The angle of the car
<i>confPos</i>	The position of the conflicting car
<i>confAngle</i>	The angle of the conflicting car

Returns

If the cars have a conflict

Definition at line 49 of file [utils.cpp](#).

carsCollided()

```
bool carsCollided (
    Car car1,
    Car car2,
    int time)
```

@bref Check if two cars collided

Parameters

<i>car1</i>	The first car
<i>car2</i>	The second car

Definition at line 23 of file [utils.cpp](#).

distance()

```
double distance (  
    sf::Vector2f p1,  
    sf::Vector2f p2) [inline]
```

Get the distance between two points.

Parameters

<i>p1</i>	The first point
<i>p2</i>	The second point

Definition at line [29](#) of file [utils.h](#).

latLonToXY()

```
sf::Vector2f latLonToXY (  
    double lat,  
    double lon) [inline]
```

Convert latitude and longitude to x and y.

Parameters

<i>lat</i>	The latitude
<i>lon</i>	The longitude

Returns

The x and y

Definition at line [17](#) of file [utils.h](#).

loadFont()

```
sf::Font loadFont ()
```

Load a font.

Returns

The font

Definition at line [13](#) of file [utils.cpp](#).

normalizeAngle()

```
double normalizeAngle (  
    double angle) [inline]
```

Normalize an angle to -PI to PI.

Parameters

<i>angle</i>	The angle
--------------	-----------

Definition at line [37](#) of file [utils.h](#).

turningRadius()

```
double turningRadius (  
    double speed) [inline]
```

Get the turning radius from the speed.

Parameters

<i>speed</i>	The speed
--------------	-----------

Returns

The turning radius

Definition at line 52 of file [utils.h](#).

turningRadiusToSpeed()

```
double turningRadiusToSpeed (
    double radius) [inline]
```

Get the speed from the turning radius.

Parameters

<i>radius</i>	The turning radius
---------------	--------------------

Returns

The speed

Definition at line 59 of file [utils.h](#).

4.25 utils.h

[Go to the documentation of this file.](#)

```
00001
00005 #pragma once
00006 #include "config.h"
00007 #include <SFML/Graphics.hpp>
00008
00009 class Car;
00010
00017 inline sf::Vector2f latLonToXY(double lat, double lon) {
00018     sf::Vector2f xy;
00019     xy.x = EARTH_RADIUS * lon * M_PI / 180;
00020     xy.y = EARTH_RADIUS * std::log(std::tan((90.0f + lat) * M_PI / 360.0f));
00021     return xy;
00022 }
00023
00029 inline double distance(sf::Vector2f p1, sf::Vector2f p2) {
00030     return std::sqrt(std::pow(p2.x - p1.x, 2) + std::pow(p2.y - p1.y, 2));
00031 }
00032
00037 inline double normalizeAngle(double angle) { // -PI to PI
00038     while (angle > M_PI) {
00039         angle -= 2 * M_PI;
00040     }
00041     while (angle <= -M_PI) {
00042         angle += 2 * M_PI;
00043     }
00044     return angle;
00045 }
00046
00052 inline double turningRadius(double speed) { return speed * speed / CAR_MAX_G_FORCE; }
00053
00059 inline double turningRadiusToSpeed(double radius) { return std::sqrt(radius * CAR_MAX_G_FORCE); }
00060
00066 bool carsCollided(Car car1, Car car2, int time);
00067
00076 bool carConflict(sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle);
00077
00082 sf::Font loadFont();
```

4.26 index.py

```
00001
00014 import sys
```

```

00015 import os
00016 import matplotlib.pyplot as plt
00017 import numpy as np
00018 from collections import defaultdict
00019
00020 # =====
00021 # User-Configurable Parameters
00022 # =====
00023
00024 # Parameters for the vertical bars representing individual data points
00025 BAR_COLOR = 'blue' # Color of the vertical bars
00026 BAR_WIDTH = 1 # Width of the bars (in data units)
00027 BAR_VERTICAL_OFFSET = 0.3 # Vertical offset: each bar spans from (y - offset) to (y + offset)
00028 BAR_ALPHA = 0.1 # Opacity of the bars (0.0 to 1.0)
00029
00030 # Parameters for the mean speed trend line
00031 MEAN_LINE_COLOR = 'red' # Color of the mean speed line
00032 MEAN_LINE_STYLE = '-' # Style of the mean speed line
00033 MEAN_LINE_WIDTH = 2 # Thickness of the mean speed line
00034
00035 # Parameters for the trend line (interpolation)
00036 TREND_LINE_COLOR = 'green' # Color of the trend line
00037 TREND_LINE_STYLE = '-' # Style of the trend line
00038 TREND_LINE_WIDTH = 2 # Thickness of the trend line
00039 TREND_DEGREE = 1 # Degree of the polynomial for the trend line (1 = linear)
00040
00041 # Parameters for the standard deviation bands
00042 STD_LINE_COLOR = 'purple' # Color of the standard deviation lines
00043 STD_LINE_STYLE = '--' # Style of the standard deviation lines
00044 STD_LINE_WIDTH = 1.5 # Thickness of the standard deviation lines
00045
00046 # Parameters for the x-axis labels
00047 X_LABEL_STEP = 4
00048
00049 # =====
00050 # Main Code
00051 # =====
00052
00053 def main():
00054     # Check if a filename is provided as a command-line argument
00055     if len(sys.argv) < 2:
00056         print("Usage: python script.py <filename>")
00057         sys.exit(1)
00058
00059     filename = sys.argv[1]
00060
00061     # Validate that the file exists
00062     if not os.path.isfile(filename):
00063         print(f"Error: File '{filename}' does not exist.")
00064         sys.exit(1)
00065
00066     # Lists to store data points
00067     x_points = [] # Number of vehicles (numCar)
00068     y_points = [] # Converted speeds (km/h)
00069     speed_data = defaultdict(list) # Dictionary mapping numCar to a list of speeds
00070     density_mapping = {} # Store density values for each numCar
00071
00072     # Read the file
00073     with open(filename, 'r', encoding='utf-8') as file:
00074         for line_number, line in enumerate(file, start=1):
00075             line = line.strip()
00076             if not line:
00077                 continue
00078
00079             # Split the line using ';' as the delimiter and remove empty tokens
00080             tokens = [token.strip() for token in line.split(';') if token.strip()]
00081
00082             if len(tokens) < 2:
00083                 continue
00084
00085             # Parse numCar (the number of vehicles) and density
00086             try:
00087                 num_car = int(tokens[0])
00088                 density = float(tokens[1]) # Store density for the x-axis label
00089                 density_mapping[num_car] = density # Ensure each numCar has a unique density mapping
00090             except ValueError:
00091                 print(f"Error on line {line_number}: Cannot parse numCar or density '{tokens[2:]}'")
00092                 continue
00093
00094             expected_token_count = 2 + num_car
00095             if len(tokens) < expected_token_count:
00096                 print(f"Error on line {line_number}: Expected {expected_token_count} values, found {len(tokens)}")
00097                 continue
00098
00099             # Process each speed value (tokens from index 2 onward), converting from m/s to km/h
00100             for token in tokens[2:]:

```



```

00101         try:
00102             speed_kmh = float(token) * 3.6
00103             x_points.append(num_car)           # Use numCar as the x-value
00104             y_points.append(speed_kmh)         # Store the speed (km/h)
00105             speed_data[num_car].append(speed_kmh) # Group speeds by numCar for averaging
00106         except ValueError:
00107             print(f"Error on line {line_number}: Cannot convert '{token}' to float.")
00108             continue
00109
00110     if not x_points:
00111         print("No valid data found. Exiting.")
00112         sys.exit(1)
00113
00114     # Compute the mean speed and standard deviation for each unique numCar
00115     unique_x = sorted(speed_data.keys())
00116     mean_y = [np.mean(speed_data[num]) for num in unique_x]
00117     std_y = [np.std(speed_data[num]) for num in unique_x] # Compute standard deviation
00118
00119     # Compute upper and lower bounds ( $\pm 1$  standard deviation)
00120     upper_y = [mean + std for mean, std in zip(mean_y, std_y)]
00121     lower_y = [mean - std for mean, std in zip(mean_y, std_y)]
00122
00123     # Fit a linear trend line (degree 1)
00124     trend_poly = np.polyfit(unique_x, mean_y, TREND_DEGREE)
00125     trend_func = np.poly1d(trend_poly)
00126
00127     # Generate smooth x values for plotting the trend curve
00128     x_smooth = np.linspace(min(unique_x), max(unique_x), 300)
00129     y_smooth = trend_func(x_smooth)
00130
00131     # Create the plot
00132     fig, ax = plt.subplots(figsize=(10, 6))
00133
00134     # Plot the individual data points as vertical bars using plt.bar
00135     bottoms = [y - BAR_VERTICAL_OFFSET for y in y_points]
00136     heights = [2 * BAR_VERTICAL_OFFSET for _ in y_points]
00137     ax.bar(x_points, heights, width=BAR_WIDTH, bottom=bottoms, color=BAR_COLOR,
00138           alpha=BAR_ALPHA, align='center')
00139
00140     # Plot the mean speed as a continuous red line
00141     ax.plot(unique_x, mean_y, color=MEAN_LINE_COLOR, linestyle=MEAN_LINE_STYLE,
00142           linewidth=MEAN_LINE_WIDTH, label="Mean Speed")
00143
00144     # Plot the trend (interpolation) curve
00145     ax.plot(x_smooth, y_smooth, color=TREND_LINE_COLOR, linestyle=TREND_LINE_STYLE,
00146           linewidth=TREND_LINE_WIDTH, label="Trend Curve")
00147
00148     # Plot  $\pm 1$  standard deviation lines
00149     ax.plot(unique_x, upper_y, color=STD_LINE_COLOR, linestyle=STD_LINE_STYLE,
00150           linewidth=STD_LINE_WIDTH, label="+1 Std Dev")
00151     ax.plot(unique_x, lower_y, color=STD_LINE_COLOR, linestyle=STD_LINE_STYLE,
00152           linewidth=STD_LINE_WIDTH, label="-1 Std Dev")
00153
00154     # Set x-axis labels with "numCar (density)"
00155     x_labels = [f"{num} ({density_mapping[num]:.0f})" if i % X_LABEL_STEP == 0 else ""
00156               for i, num in enumerate(unique_x)]
00157     ax.set_xticks(unique_x)
00158     ax.set_xticklabels(x_labels, rotation=45, ha='right') # Rotate for better readability
00159     ax.set_xlim(min(unique_x)-0.5, max(unique_x)+0.5)
00160
00161     ax.set_xlabel("Number of Vehicles (Density)")
00162     ax.set_ylabel("Average Speed (km/h)")
00163     ax.set_title("Number of Vehicles vs Average Speeds with Std Deviation")
00164     ax.legend()
00165
00166     # Display the plot (grid is not added)
00167     plt.show()
00168
00169 if __name__ == '__main__':
00170     main()

```

4.27 aStar.cpp File Reference

A* algorithm implementation.

```

#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>

```

```
#include <spdlog/spdlog.h>
#include <unordered_set>
```

4.27.1 Detailed Description

A* algorithm implementation.

This file contains the implementation of the A* algorithm. It is used to find the shortest path between two points in a graph.

Definition in file [aStar.cpp](#).

4.28 aStar.cpp

[Go to the documentation of this file.](#)

```
00001
00008 #include "aStar.h"
00009 #include "config.h"
00010 #include "dubins.h"
00011 #include "utils.h"
00012
00013 #include <ompl/base/State.h>
00014 #include <ompl/base/StateSpace.h>
00015 #include <ompl/base/spaces/DubinsStateSpace.h>
00016 #include <spdlog/spdlog.h>
00017 #include <unordered_set>
00018
00019 namespace ob = ompl::base;
00020
00021 AStar::AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph) {
00022     this->start.point = start;
00023     this->start.speed = 0;
00024     this->end.point = end;
00025     this->end.speed = 0;
00026     this->graph = cityGraph;
00027 }
00028
00029 void AStar::process() {
00030     path.clear();
00031
00032     std::unordered_map<AStar::node, AStar::node> cameFrom;
00033     std::unordered_map<AStar::node, double> gScore;
00034     std::unordered_map<AStar::node, double> fScore;
00035
00036     auto heuristic = [&](const AStar::node &a) {
00037         CityGraph::neighbor end_;
00038         end_.point = end.point;
00039         end_.maxSpeed = 0;
00040         end_.turningRadius = CAR_MIN_TURNING_RADIUS;
00041         Dubins dubins(a.point, end_);
00042         return dubins.distance();
00043     };
00044     auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };
00045
00046     std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSet(compare);
00047     std::unordered_set<AStar::node> isInOpenSet;
00048
00049     openSet.push(start);
00050     gScore[start] = 0;
00051     fScore[start] = heuristic(start);
00052
00053     auto neighbors = graph.getNeighbors();
00054
00055     int nbIterations = 0;
00056     while (!openSet.empty() && nbIterations++ < 1e5) {
00057         AStar::node current = openSet.top();
00058         openSet.pop();
00059         isInOpenSet.erase(current);
00060
00061         if (current.point == end.point) {
00062             AStar::node currentCopy = current;
00063
00064             while (!(currentCopy == start)) {
00065                 path.push_back(currentCopy);
00066                 currentCopy = cameFrom[currentCopy];
00067             }
00068             path.push_back(currentCopy);
00069             std::reverse(path.begin(), path.end());
00070             break;
00071         }
00072     }
```

```

00073     for (const auto &neighborGraphPoint : neighbors[current.point]) {
00074         AStar::node neighbor;
00075         neighbor.point = neighborGraphPoint.point;
00076         neighbor.speed = neighborGraphPoint.maxSpeed;
00077         neighbor.arcFrom = {current.point, neighborGraphPoint};
00078
00079         double tentativeGScore = gScore[current] + neighborGraphPoint.distance;
00080
00081         if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
00082             cameFrom[neighbor] = current;
00083             gScore[neighbor] = tentativeGScore;
00084             fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00085
00086             if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00087                 openSet.push(neighbor);
00088                 isInOpenSet.insert(neighbor);
00089             }
00090         }
00091     }
00092 }
00093 }

```

4.29 car.cpp File Reference

[Car](#) class implementation.

```

#include "car.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <iostream>
#include <random>

```

4.29.1 Detailed Description

[Car](#) class implementation.

This file contains the implementation of the [Car](#) class.

Definition in file [car.cpp](#).

4.30 car.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include "car.h"
00008 #include "config.h"
00009 #include "dubins.h"
00010 #include "utils.h"
00011
00012 #include <iostream>
00013 #include <random>
00014
00015 Car::Car() {
00016     std::vector<sf::Color> colors = {sf::Color(50, 120, 190), sf::Color(183, 132, 144), sf::Color(105,
00017     101, 89),
00018                                     sf::Color(182, 18, 34), sf::Color(24, 25, 24), sf::Color(17,
00019     86, 122)};
00020     color = colors[rand() % colors.size()];
00021 }
00022
00021 void Car::move() {
00022     if (currentPoint >= (int)path.size())
00023         return;
00024
00025     currentPoint++;
00026 }
00027
00028 void Car::render(sf::RenderWindow &window) {
00029     if (1 + currentPoint >= (int)path.size())
00030         return;
00031
00032     sf::Vector2f point = path[currentPoint];
00033     sf::Vector2f nextPoint = path[currentPoint + 1];
00034
00035     sf::RectangleShape shape(sf::Vector2f(CAR_LENGTH, CAR_WIDTH));
00036     shape.setOrigin(CAR_LENGTH / 2.0f, CAR_WIDTH / 2.0f);
00037     shape.setPosition(point);

```

```

00038     shape.setRotation(atan2(nextPoint.y - point.y, nextPoint.x - point.x) * 180.0f / M_PI);
00039     if (debug)
00040         shape.setFillColor(sf::Color(255, 0, 0));
00041     else
00042         shape.setFillColor(color);
00043     window.draw(shape);
00044
00045     if (!debug)
00046         return;
00047
00048     // Render speed, elapsed time, remaining time, and distance
00049     int speed = (int)(getSpeed() * 3.6f);
00050     int dSpeed = (getSpeed() * 3.6f - (double)speed) * 100;
00051     sf::Font font = loadFont();
00052     sf::Text text;
00053     text.setFont(font);
00054     text.setCharacterSize(24);
00055     text.setFillColor(sf::Color::White);
00056     text.setPosition(getPosition());
00057     text.setString(std::to_string(speed) + "." + std::to_string(dSpeed) + " km/h" + "\n" +
00058         std::to_string((int)getElapsedTime()) + "s / " +
00059         std::to_string((int)getRemainingTime()) + "s" + "\n" +
00060         std::to_string((int)getElapsedDistance()) + "m / " +
00061         std::to_string((int)getRemainingDistance()) +
00062         "m");
00063     text.setOutlineColor(sf::Color::Black);
00064     text.setOutlineThickness(1.0f);
00065     text.scale(0.1f, 0.1f);
00066     text.setOrigin(text.getLocalBounds().width / 2.0f, text.getLocalBounds().height / 2.0f);
00067     window.draw(text);
00068
00069     // Render path
00070     for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00071         sf::Vertex line[] = {sf::Vertex(path[i]), sf::Vertex(path[i + 1])};
00072         line[0].color = sf::Color(255, 255, 255);
00073         line[1].color = sf::Color(255, 255, 255);
00074         window.draw(line, 2, sf::Lines);
00075     }
00076
00077 void Car::assignPath(std::vector<AStar::node> path) {
00078     this->path.clear();
00079     this->aStarPath = path;
00080     DubinsPath dubins(path);
00081     std::vector<CityGraph::point> dubinsPath_ = dubins.path();
00082     for (CityGraph::point point : dubinsPath_) {
00083         this->path.push_back(point.position);
00084     }
00085     currentPoint = 0;
00086 }
00087
00088 void Car::assignExistingPath(std::vector<sf::Vector2f> path) {
00089     this->path = path;
00090     currentPoint = 0;
00091 }
00092
00093 double Car::getSpeed() {
00094     if (currentPoint >= (int)path.size() - 1)
00095         return 0;
00096
00097     sf::Vector2f diff = path[currentPoint + 1] - path[currentPoint];
00098     return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00099 }
00100
00101 double Car::getSpeedAt(int index) {
00102     if (index >= (int)path.size() - 1)
00103         return 0;
00104
00105     sf::Vector2f diff = path[index + 1] - path[index];
00106     return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00107 }
00108
00109 double Car::getRemainingTime() { return (double)(path.size() - currentPoint) * SIM_STEP_TIME; }
00110 double Car::getElapsedTime() { return (double)currentPoint * SIM_STEP_TIME; }
00111 double Car::getPathTime() { return (double)path.size() * SIM_STEP_TIME; }
00112
00113 double Car::getRemainingDistance() {
00114     double dist = 0;
00115     for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00116         sf::Vector2f diff = path[i + 1] - path[i];
00117         dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00118     }
00119     return dist;
00120 }
00121
00122 double Car::getElapsedDistance() {

```

```

00123     double dist = 0;
00124     for (int i = 0; i < currentPoint; i++) {
00125         sf::Vector2f diff = path[i + 1] - path[i];
00126         dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00127     }
00128
00129     return dist;
00130 }
00131
00132 double Car::getPathLength() {
00133     double dist = 0;
00134     for (int i = 0; i < (int)path.size() - 1; i++) {
00135         sf::Vector2f diff = path[i + 1] - path[i];
00136         dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00137     }
00138
00139     return dist;
00140 }
00141
00142 void Car::chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap) {
00143     CityGraph::point start;
00144     CityGraph::point end;
00145
00146     double minDistance = std::max(graph.getWidth(), graph.getHeight()) / 2.0;
00147     std::vector<AStar::node> path;
00148
00149     do {
00150         path.clear();
00151         start = graph.getRandomPoint();
00152         end = graph.getRandomPoint();
00153
00154         if (std::sqrt(std::pow(start.position.x - end.position.x, 2) + std::pow(start.position.y -
00155 end.position.y, 2)) <
00156             minDistance)
00157             continue;
00158
00159         AStar aStar(start, end, graph);
00160         path = aStar.findPath();
00161
00162         if (!path.empty() && (int)path.size() >= 3) {
00163             TimedAStar timedAStar(start, end, graph, nullptr, 0);
00164             path.clear();
00165             path = timedAStar.findPath();
00166         } while (path.empty() || (int)path.size() < 3);
00167
00168         this->assignStartEnd(start, end);
00169         this->assignPath(path);
00170     }
00171
00172 double Car::getAverageSpeed(CityGraph &graph) {
00173     double dist = 0;
00174     double time = 0;
00175     auto outOfBounds = [&](sf::Vector2f p) {
00176         return p.x < 0 || p.y < 0 || p.x > graph.getWidth() || p.y > graph.getHeight();
00177     };
00178
00179     for (int i = 0; i < (int)path.size() - 1; i++) {
00180         if (outOfBounds(path[i]) || outOfBounds(path[i + 1]))
00181             continue;
00182
00183         sf::Vector2f diff = path[i + 1] - path[i];
00184         dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00185         time += SIM_STEP_TIME;
00186     }
00187
00188     if (time == 0)
00189         return 0;
00190
00191     return dist / time;
00192 }

```

4.31 cityGraph.cpp File Reference

City graph implementation.

```

#include <iostream>
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <random>

```

```
#include <spdlog/spdlog.h>
#include "cityGraph.h"
#include "config.h"
#include "utils.h"
```

4.31.1 Detailed Description

City graph implementation.

This file contains the implementation of the [CityGraph](#) class. This class represents the graph of the city. It contains the points of the graph and the neighbors of each point.

Definition in file [cityGraph.cpp](#).

4.32 cityGraph.cpp

[Go to the documentation of this file.](#)

```
00001
00008 #include <iostream>
00009 #include <ompl/base/State.h>
00010 #include <ompl/base/StateSpace.h>
00011 #include <ompl/base/spaces/DubinsStateSpace.h>
00012 #include <ompl/geometric/SimpleSetup.h>
00013 #include <ompl/geometric/planners/rrt/RRT.h>
00014 #include <random>
00015 #include <spdlog/spdlog.h>
00016
00017 #include "cityGraph.h"
00018 #include "config.h"
00019 #include "utils.h"
00020
00021 namespace ob = ompl::base;
00022
00023 void CityGraph::createGraph(const CityMap &cityMap) {
00024     auto roads = cityMap.getRoads();
00025     auto intersections = cityMap.getIntersections();
00026
00027     this->height = cityMap.getHeight();
00028     this->width = cityMap.getWidth();
00029
00030     // Graph's points are evenly distributed along a road segment
00031     for (const auto &road : roads) {
00032         if (road.segments.empty()) {
00033             continue;
00034         }
00035
00036         int numSeg = 0;
00037         for (const auto &segment : road.segments) {
00038             if (numSeg > 0) { // Link to the previous one
00039                 for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
00040                     double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width /
road.numLanes;
00041                     offset += road.width / (2 * road.numLanes);
00042
00043                     point point1;
00044                     point1.angle = road.segments[numSeg - 1].angle;
00045                     point1.position =
00046                         sf::Vector2f(road.segments[numSeg - 1].p2_offset.x + offset * sin(road.segments[numSeg -
1].angle),
00047                                     road.segments[numSeg - 1].p2_offset.y + offset * -cos(road.segments[numSeg
- 1].angle));
00048
00049                     point point2;
00050                     point2.angle = road.segments[numSeg].angle;
00051                     point2.position =
00052                         sf::Vector2f(road.segments[numSeg].p1_offset.x + offset *
sin(road.segments[numSeg].angle),
00053                                     road.segments[numSeg].p1_offset.y + offset *
-cos(road.segments[numSeg].angle));
00054
00055                     linkPoints(point1, point2, 2, true);
00056                 }
00057             }
00058             numSeg++;
00059
00060             double segmentLength =
00061                 sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y -
segment.p1_offset.y, 2));
00062             double pointDistance = 15;
00063             int numPoints = segmentLength / pointDistance;
```

```

00064     double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
00065     double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
00066     double dx_a = sin(segment.angle);
00067     double dy_a = -cos(segment.angle);
00068
00069     if (dx_a < 0) {
00070         dx_a = -dx_a;
00071         dy_a = -dy_a;
00072     }
00073
00074     for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
00075         double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
00076         offset += road.width / (2 * road.numLanes);
00077
00078         if (numPoints == 0) {
00079             point point1;
00080             point1.angle = segment.angle;
00081             point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y +
offset * dy_a);
00082
00083             point point2;
00084             point2.angle = segment.angle;
00085             point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y +
offset * dy_a);
00086
00087             linkPoints(point1, point2, 2, true);
00088             continue;
00089         }
00090
00091         for (int i = 0; i <= numPoints; i++) {
00092             point point1;
00093             point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
segment.p1_offset.y + i * dy_s + offset * dy_a);
00094             point1.angle = segment.angle;
00095
00096             if (i > 0) {
00097                 for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
00098                     double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width /
road.numLanes;
00099                     offset2 += road.width / (2 * road.numLanes);
00100
00101                     point point2;
00102                     point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
00103                     point2.angle = segment.angle;
00104
00105                     int direction = 2;
00106                     double a = normalizeAngle(atan2(dy_a, dx_a));
00107                     if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
00108                         if (dy_s >= 0) {
00109                             direction = offset > 0 ? 0 : 1;
00110                         } else {
00111                             direction = offset > 0 ? 1 : 0;
00112                         }
00113                         linkPoints(point1, point2, direction, offset == offset2);
00114                     } else {
00115                         if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
00116                             linkPoints(point1, point2, 2, true);
00117                         }
00118                     }
00119                 }
00120             }
00121         }
00122     }
00123 }
00124 }
00125 }
00126 }
00127
00128 // Connect the intersections
00129 for (const auto &intersection : intersections) {
00130     for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
00131         for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
00132             const auto &road1 = roads[roadSegmentId1.first];
00133             const auto &road2 = roads[roadSegmentId2.first];
00134             const auto &segment1 = road1.segments[roadSegmentId1.second];
00135             const auto &segment2 = road2.segments[roadSegmentId2.second];
00136
00137             // Find the point of the segment2 closest to the intersection
00138             point point1;
00139             point1.angle = segment1.angle;
00140             point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2,
intersection.center))
? segment1.p1_offset
: segment1.p2_offset;
00141
00142             point point2;
00143             point2.angle = segment2.angle;
00144             point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2,

```

```

        intersection.center))
00147                                     ? segment2.p1_offset
00148                                     : segment2.p2_offset;
00149
00150         for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
00151             double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width /
road1.numLanes;
00152             offset1 += road1.width / (2 * road1.numLanes);
00153
00154             for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
00155                 double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width /
road2.numLanes;
00156                 offset2 += road2.width / (2 * road2.numLanes);
00157
00158                 point point1_offset;
00159                 point1_offset.angle = segment1.angle;
00160                 point1_offset.position = sf::Vector2f(point1.position.x + offset1 * sin(segment1.angle),
00161                                                         point1.position.y + offset1 * -cos(segment1.angle));
00162
00163                 point point2_offset;
00164                 point2_offset.angle = segment2.angle;
00165                 point2_offset.position = sf::Vector2f(point2.position.x + offset2 * sin(segment2.angle),
00166                                                         point2.position.y + offset2 * -cos(segment2.angle));
00167
00168                 linkPoints(point1_offset, point2_offset, 2, true);
00169             }
00170         }
00171     }
00172 }
00173 }
00174
00175 spdlog::info("Graph created with {} points", graphPoints.size());
00176
00177 // Remove all the neighbors that need to turn too much
00178 for (auto &point : graphPoints) {
00179     std::vector<neighbor> newNeighbors;
00180     double distance;
00181     for (auto &neighbor : neighbors[point]) {
00182         double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
00183         bool can = canLink(point, neighbor.point, speed, &distance);
00184
00185         if (!can)
00186             continue;
00187
00188         while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
00189             speed += 0.1;
00190             if (speed >= CAR_MAX_SPEED_MS) {
00191                 speed = CAR_MAX_SPEED_MS;
00192                 break;
00193             }
00194         }
00195
00196         if (can) {
00197             neighbor.maxSpeed = speed - 0.1;
00198             neighbor.distance = std::sqrt(std::pow(neighbor.point.position.x - point.position.x, 2) +
00199                                           std::pow(neighbor.point.position.y - point.position.y, 2));
00200
00201             neighbor.turningRadius = turningRadius(speed);
00202             newNeighbors.push_back(neighbor);
00203         }
00204     }
00205
00206     neighbors[point].clear();
00207     for (const auto &neighbor : newNeighbors) {
00208         neighbors[point].push_back(neighbor);
00209     }
00210 }
00211 }
00212
00213 void CityGraph::linkPoints(const point &p, const point &n, int direction, bool subPoints) {
00214     std::vector<double> anglesPoint = {normalizeAngle(p.angle), normalizeAngle(p.angle + M_PI)};
00215     std::vector<double> anglesNeighbor = {normalizeAngle(n.angle), normalizeAngle(n.angle + M_PI)};
00216
00217     point copyPoint = p;
00218     point copyNeighbor = n;
00219
00220     bool isRiP = direction == 2 || direction == 0;
00221     bool isRiN = direction == 2 || direction == 1;
00222     bool isStraight = direction != 2;
00223     isStraight &= (anglesPoint[0] == anglesNeighbor[0] || anglesPoint[0] == anglesNeighbor[1] ||
00224                   anglesPoint[1] == anglesNeighbor[0] || anglesPoint[1] == anglesNeighbor[1]);
00225     isStraight &= subPoints;
00226
00227     if (!isStraight) {
00228         for (const auto &anglePoint : anglesPoint) {
00229             for (const auto &angleNeighbor : anglesNeighbor) {
00230                 copyPoint.angle = anglePoint;

```



```

00231         copyNeighbor.angle = angleNeighbor;
00232
00233         neighbors[copyPoint].push_back({copyNeighbor, 0, 0, 0, isRiP}); // This fields will be updated
    later
00234         neighbors[copyNeighbor].push_back({copyPoint, 0, 0, 0, isRiN});
00235
00236         graphPoints.insert(copyPoint);
00237         graphPoints.insert(copyNeighbor);
00238     }
00239 }
00240     return;
00241 }
00242
00243 // Link adding points in the middle
00244 double pointDistance = 3;
00245 double distance = std::sqrt(std::pow(n.position.x - p.position.x, 2) + std::pow(n.position.y -
p.position.y, 2));
00246 int numPoints = distance / pointDistance;
00247 double dx = (n.position.x - p.position.x) / numPoints;
00248 double dy = (n.position.y - p.position.y) / numPoints;
00249
00250 for (const auto &anglePoint : anglesPoint) {
00251     for (const auto &angleNeighbor : anglesNeighbor) {
00252         point previousPoint = p;
00253         previousPoint.angle = anglePoint;
00254
00255         for (int i = 1; i <= numPoints; i++) {
00256             point newPoint;
00257             newPoint.position = sf::Vector2f(p.position.x + i * dx, p.position.y + i * dy);
00258             newPoint.angle = anglePoint;
00259
00260             neighbors[previousPoint].push_back({newPoint, 0, 0, 0, isRiP}); // This fields will be updated
    later
00261             neighbors[newPoint].push_back({previousPoint, 0, 0, 0, isRiN});
00262
00263             previousPoint = newPoint;
00264
00265             graphPoints.insert(newPoint);
00266         }
00267
00268         // Add the last point
00269         neighbors[previousPoint].push_back({n, 0, 0, 0, isRiP}); // This fields will be updated later
00270     }
00271 }
00272 }
00273
00274 CityGraph::point CityGraph::getRandomPoint() const {
00275     std::vector<point> graphPointsOut;
00276     for (const auto &point : graphPoints) {
00277         if (point.position.x + CAR_LENGTH < 0 || point.position.x - CAR_LENGTH > width ||
00278             point.position.y + CAR_LENGTH < 0 || point.position.y - CAR_LENGTH > height)
00279             graphPointsOut.push_back(point);
00280     }
00281
00282     auto it = graphPointsOut.begin();
00283     std::random_device rd;
00284     std::mt19937 gen(rd());
00285     std::uniform_int_distribution<> dis(0, graphPointsOut.size() - 1);
00286
00287     std::advance(it, dis(gen));
00288
00289     return *it;
00290 }
00291
00292 bool CityGraph::canLink(const point &point1, const point &point2, double speed, double *distance)
const {
00293     double radius = turningRadius(speed);
00294
00295     ob::DubinsStateSpace space(radius);
00296
00297     ob::State *start = space.allocState();
00298     ob::State *end = space.allocState();
00299
00300     start->as<ob::DubinsStateSpace::StateType>()->setXY(point1.position.x, point1.position.y);
00301     start->as<ob::DubinsStateSpace::StateType>()->setYaw(point1.angle);
00302
00303     end->as<ob::DubinsStateSpace::StateType>()->setXY(point2.position.x, point2.position.y);
00304     end->as<ob::DubinsStateSpace::StateType>()->setYaw(point2.angle);
00305
00306     double total = 0;
00307
00308     // Extract the path
00309     ob::DubinsStateSpace::DubinsPath path = space.dubins(start, end);
00310     for (unsigned int i = 0; i < 3; ++i) // Dubins path has up to 3 segments
00311     {
00312         auto type = path.type_[i];
00313         if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_LEFT) {

```

```

00314         total += std::abs(path.length_[i]);
00315     } else if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_RIGHT) {
00316         total += std::abs(path.length_[i]);
00317     }
00318 }
00319
00320 *distance = space.distance(start, end);
00321 return total < M_PI * 0.75f;
00322 }

```

4.33 cityMap.cpp File Reference

[CityMap](#) class implementation.

```

#include <iostream>
#include <math.h>
#include <set>
#include "spdlog/spdlog.h"
#include "tinycl2.h"
#include "cityMap.h"
#include "utils.h"

```

4.33.1 Detailed Description

[CityMap](#) class implementation.

This file contains the implementation of the [CityMap](#) class.

Definition in file [cityMap.cpp](#).

4.34 cityMap.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include <iostream>
00008 #include <math.h>
00009 #include <set>
00010
00011 #include "spdlog/spdlog.h"
00012 #include "tinycl2.h"
00013
00014 #include "cityMap.h"
00015 #include "utils.h"
00016
00017 CityMap::CityMap() {
00018     roads.clear();
00019     intersections.clear();
00020     minLatLon.x = minLatLon.y = maxLatLon.x = maxLatLon.y = 0;
00021 }
00022
00023 void CityMap::loadFile(const std::string &filename) {
00024     spdlog::info("Loading file: {}", filename);
00025
00026     tinycl2::XMLDocument doc;
00027     // Load the XML file
00028     if (doc.LoadFile(filename.c_str()) != tinycl2::XML_SUCCESS) {
00029         spdlog::error("Failed to load file: {}", filename);
00030         return;
00031     }
00032
00033     // Extract the bounds of the map
00034     tinycl2::XMLElement *bounds = doc.FirstChildElement("osm")->FirstChildElement("bounds");
00035     if (!bounds) {
00036         spdlog::error("Failed to extract bounds from file: {}", filename);
00037         return;
00038     }
00039
00040     minLatLon.x = bounds->FloatAttribute("minlon");
00041     minLatLon.y = bounds->FloatAttribute("minlat");
00042     maxLatLon.x = bounds->FloatAttribute("maxlon");
00043     maxLatLon.y = bounds->FloatAttribute("maxlat");
00044
00045     // Define the width and height of the map
00046     width = latLonToXY(minLatLon.y, minLatLon.x).x - latLonToXY(maxLatLon.y, maxLatLon.x).x;
00047     height = latLonToXY(minLatLon.y, minLatLon.x).y - latLonToXY(maxLatLon.y, maxLatLon.x).y;
00048     width = std::abs(width);
00049     height = std::abs(height);
00050

```

```

00051     std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
00052     spdlog::info("Loading roads and buildings ...");
00053
00054     // List of highway types to exclude
00055     std::set<std::string> excludedHighways = {"footway", "path", "pedestrian", "cycleway",
00056                                              "steps", "track", "bridleway", "service"};
00057
00058     // List of highway types to include
00059     std::set<std::string> includedHighways = {
00060         "motorway", "trunk", "primary", "secondary", "tertiary",
00061         "unclassified", "residential",
00062         "living_street", "motorway_link", "trunk_link", "primary_link", "secondary_link",
00063         "tertiary_link"};
00064
00065     // Extract the roads
00066     tinyxml2::XMLElement *way = doc.FirstChildElement("osm")->FirstChildElement("way");
00067     int roadId = 0;
00068     while (way) {
00069         road r;
00070         building b;
00071         greenArea g;
00072         waterArea w;
00073         r.width = DEFAULT_ROAD_WIDTH;
00074         r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00075         r.id = roadId;
00076
00077         tinyxml2::XMLElement *nd = way->FirstChildElement("nd");
00078         while (nd) {
00079             tinyxml2::XMLElement *node = doc.FirstChildElement("osm")->FirstChildElement("node");
00080             while (node) {
00081                 if (node->IntAttribute("id") == nd->IntAttribute("ref")) {
00082                     sf::Vector2f p;
00083                     p.x = node->FloatAttribute("lon");
00084                     p.y = node->FloatAttribute("lat");
00085
00086                     if (r.segments.size() > 0) {
00087                         segment s;
00088                         s.p1 = r.segments.back().p2;
00089                         s.p2 = p;
00090                         r.segments.push_back(s);
00091                     } else {
00092                         segment s;
00093                         s.p1 = p;
00094                         s.p2 = p;
00095                         r.segments.push_back(s);
00096                     }
00097
00098                     b.points.push_back(p);
00099                     g.points.push_back(p);
00100                     w.points.push_back(p);
00101                     break;
00102                 }
00103                 node = node->NextSiblingElement("node");
00104             }
00105             nd = nd->NextSiblingElement("nd");
00106         }
00107
00108         // Remove the first segment (it has the same p1 and p2)
00109         r.segments.erase(r.segments.begin());
00110
00111         std::string highwayType;
00112         bool isHighway = false;
00113         bool isBuilding = false;
00114         bool isUnderground = false;
00115         bool isGreenArea = false;
00116         bool isWaterArea = false;
00117         bool widthSet = false;
00118         bool lanesSet = false;
00119         tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
00120         while (tag) {
00121             if (strcmp(tag->Attribute("k"), "width") == 0) {
00122                 r.width = tag->FloatAttribute("v");
00123                 widthSet = true;
00124             } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
00125                 r.numLanes = tag->IntAttribute("v");
00126                 lanesSet = true;
00127             } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
00128                 highwayType = tag->Attribute("v");
00129                 isHighway = true;
00130             } else if (strcmp(tag->Attribute("k"), "building") == 0) {
00131                 isBuilding = true;
00132             } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
00133                 int layerValue = tag->IntAttribute("v");
00134                 if (layerValue < 0) {
00135                     isUnderground = true;
00136                 }
00137             } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
00138

```

```

00136         if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
00137             strcmp(tag->Attribute("v"), "meadow") == 0) {
00138             isGreenArea = true;
00139             g.type = 0;
00140         }
00141     } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
00142         if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
00143             isGreenArea = true;
00144             g.type = 1;
00145         }
00146     } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
00147         (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream")
00148 == 0 ||
00149             strcmp(tag->Attribute("v"), "canal") == 0)) {
00149         isWaterArea = true;
00150     } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
00151         (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland")
00152 == 0)) {
00152         isWaterArea = true;
00153     } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
00154         (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0
00155 ||
00156             strcmp(tag->Attribute("v"), "river") == 0)) {
00156         isWaterArea = true;
00157     }
00158     tag = tag->NextSiblingElement("tag");
00159 }
00160 if (!widthSet && !lanesSet) {
00161     r.width = DEFAULT_ROAD_WIDTH;
00162     r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00163 } else if (!widthSet) {
00164     r.width = r.numLanes * DEFAULT_LANE_WIDTH;
00165 } else if (!lanesSet) {
00166     r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00167 }
00168 r.width = std::max(r.width, MIN_ROAD_WIDTH);
00169 r.numLanes = std::max(r.numLanes, 1);
00170
00171 if (isUnderground) {
00172     way = way->NextSiblingElement("way");
00173     continue;
00174 }
00175 if (isBuilding) {
00176     buildings.push_back(b);
00177     way = way->NextSiblingElement("way");
00178     continue;
00179 }
00180 if (isGreenArea) {
00181     greenAreas.push_back(g);
00182     way = way->NextSiblingElement("way");
00183     continue;
00184 }
00185 if (isWaterArea) {
00186     waterAreas.push_back(w);
00187     way = way->NextSiblingElement("way");
00188     continue;
00189 }
00190 if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
00191     way = way->NextSiblingElement("way");
00192     continue;
00193 }
00194 if (includedHighways.find(highwayType) != includedHighways.end()) {
00195     roads.push_back(r);
00196     roadId++;
00197 }
00198 way = way->NextSiblingElement("way");
00199 }
00200 }
00201
00202 // Convert lat/lon to meters (using the upper-left corner as origin)
00203 sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
00204 sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
00205 for (auto &r : roads) {
00206     for (auto &s : r.segments) {
00207         s.pl = latLonToXY(s.pl.y, s.pl.x);
00208         s.p2 = latLonToXY(s.p2.y, s.p2.x);
00209
00210         s.pl.x -= minXY.x;
00211         s.pl.y -= minXY.y;
00212         s.p2.x -= minXY.x;
00213         s.p2.y -= minXY.y;
00214
00215         // Symetri to the x-axis
00216         s.pl.y = maxXY.y - minXY.y - s.pl.y;
00217         s.p2.y = maxXY.y - minXY.y - s.p2.y;
00218
00219         s.pl_offset = s.pl;

```

```

00220         s.p2_offset = s.p2;
00221
00222         s.angle = std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x);
00223     }
00224 }
00225 for (auto &b : buildings) {
00226     for (auto &p : b.points) {
00227         p = latLonToXY(p.y, p.x);
00228
00229         p.x -= minXY.x;
00230         p.y -= minXY.y;
00231
00232         // Symetri to the x-axis
00233         p.y = maxXY.y - minXY.y - p.y;
00234     }
00235 }
00236 for (auto &g : greenAreas) {
00237     for (auto &p : g.points) {
00238         p = latLonToXY(p.y, p.x);
00239
00240         p.x -= minXY.x;
00241         p.y -= minXY.y;
00242
00243         // Symetri to the x-axis
00244         p.y = maxXY.y - minXY.y - p.y;
00245     }
00246 }
00247 for (auto &w : waterAreas) {
00248     for (auto &p : w.points) {
00249         p = latLonToXY(p.y, p.x);
00250
00251         p.x -= minXY.x;
00252         p.y -= minXY.y;
00253
00254         // Symetri to the x-axis
00255         p.y = maxXY.y - minXY.y - p.y;
00256     }
00257 }
00258
00259 std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
00260 spdlog::info("Roads and buildings loaded ({} ms)",
00261             std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());
00262
00263 spdlog::info("Loading intersections ...");
00264
00265 // Intersections are at any roads' points if they are near another one
00266 // First add the intersections for each node point
00267 // Then merge the intersections that are close to each other
00268 intersections.clear();
00269 int intersectionId = 0;
00270
00271 // Add the intersections for each road segment
00272 spdlog::debug("Adding intersections ...");
00273 for (auto r : roads) {
00274     for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
00275         segment s = r.segments[s_id];
00276         std::vector<sf::Vector2f> points = {s.p1, s.p2};
00277         for (auto p : points) {
00278             intersection i = {intersectionId++, p, r.width / 2, {}};
00279             i.roadSegmentIds.push_back({r.id, s_id});
00280             intersections.push_back(i);
00281         }
00282     }
00283 }
00284 spdlog::debug("Intersections added");
00285
00286 // Merge the intersections that are close to each other
00287 spdlog::debug("Merging intersections ...");
00288 for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
00289     for (int i = 0; i < (int)intersections.size(); i++) {
00290         for (int j = i + 1; j < (int)intersections.size(); j++) {
00291             bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();
00292
00293             if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
00294                 is_i = intersections[i].id < intersections[j].id;
00295             }
00296
00297             double minSpace = intersections[i].radius + intersections[j].radius;
00298             minSpace /= distCoef;
00299
00300             if (distance(intersections[i].center, intersections[j].center) < minSpace) {
00301                 // Merge the intersections to i or j (depending on is_i)
00302                 int index_from = is_i ? j : i;
00303                 int index_to = is_i ? i : j;
00304
00305                 for (auto &r : intersections[index_from].roadSegmentIds) {
00306                     intersections[index_to].roadSegmentIds.push_back(r);

```

```

00307         }
00308
00309         intersections.erase(intersections.begin() + index_from);
00310         i -= 1;
00311         break;
00312     }
00313 }
00314 }
00315 }
00316 spdlog::debug("Intersections merged");
00317
00318 // Make the road point to be outside the intersection
00319 spdlog::debug("Adding offsets to the roads ...");
00320 for (auto &i : intersections) {
00321     for (auto &roadInfo : i.roadSegmentIds) {
00322         double dx =
00323             roads[roadInfo.first].segments[roadInfo.second].p2.x -
roads[roadInfo.first].segments[roadInfo.second].p1.x;
00324         double dy =
00325             roads[roadInfo.first].segments[roadInfo.second].p2.y -
roads[roadInfo.first].segments[roadInfo.second].p1.y;
00326         double dd = distance({0, 0}, {(float)dx, (float)dy});
00327         dx /= dd;
00328         dy /= dd;
00329
00330         double radius = i.radius;
00331
00332         if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
00333             roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
00334             roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
00335         } else {
00336             dx = -dx;
00337             dy = -dy;
00338             roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
00339             roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
00340         }
00341     }
00342 }
00343 }
00344 spdlog::debug("Offsets added");
00345
00346 // Remove the intersections that link the same road
00347 spdlog::debug("Removing intersections that link the same road ...");
00348 for (int i = 0; i < (int)intersections.size(); i++) {
00349     if (intersections[i].roadSegmentIds.size() != 2)
00350         continue;
00351
00352     if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
00353         intersections.erase(intersections.begin() + i);
00354         i -= 1;
00355     }
00356 }
00357 spdlog::debug("Intersections removed");
00358
00359 // Log all the intersections and roads
00360 for (auto r : roads) {
00361     spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes,
r.segments.size());
00362 }
00363 for (auto i : intersections) {
00364     spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id,
i.center.x, i.center.y,
00365         i.radius, i.roadSegmentIds.size());
00366 }
00367
00368 std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
00369 spdlog::info("Intersections loaded ({} ms)",
00370     std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());
00371
00372 spdlog::info("Number of roads: {}", roads.size());
00373 spdlog::info("Number of buildings: {}", buildings.size());
00374 spdlog::info("Number of intersections: {}", intersections.size());
00375
00376 spdlog::info("Width: {} m", width);
00377 spdlog::info("Height: {} m", height);
00378
00379 isLoaded = true;
00380 }

```

4.35 constraintController.cpp File Reference

[ConstraintController](#) class implementation.

```

#include <iostream>
#include <spdlog/spdlog.h>

```

```
#include "aStar.h"
#include "dubins.h"
```

4.35.1 Detailed Description

[ConstraintController](#) class implementation.

This file contains the implementation of the [ConstraintController](#) class. This class is used to control the constraints of the A* algorithm. It is used to check if a car can move to a certain point in the graph at a certain time.

Definition in file [constraintController.cpp](#).

4.36 constraintController.cpp

[Go to the documentation of this file.](#)

```
00001
00009 #include <iostream>
00010 #include <spdlog/spdlog.h>
00011
00012 #include "aStar.h"
00013 #include "dubins.h"
00014
00015 void ConstraintController::addConstraint(AStar::conflict constraint) {
00016     if (constraint.car < 0) {
00017         spdlog::error("Invalid car index for constraint");
00018         throw std::runtime_error("Invalid car index for constraint");
00019     }
00020
00021     while (constraints.size() <= constraint.car) {
00022         constraints.push_back(std::vector<std::vector<AStar::conflict>>());
00023     }
00024
00025     while (constraints[constraint.car].size() <= constraint.time) {
00026         constraints[constraint.car].push_back(std::vector<AStar::conflict>());
00027     }
00028
00029     constraints[constraint.car][constraint.time].push_back(constraint);
00030 }
00031
00032 bool ConstraintController::hasConstraint(AStar::conflict constraint) {
00033     if (constraint.car < 0) {
00034         spdlog::error("Invalid car index for constraint");
00035         throw std::runtime_error("Invalid car index for constraint");
00036     }
00037     if (constraints.size() <= constraint.car) {
00038         return false;
00039     }
00040     for (int t = std::max(0, constraint.time - 1);
00041          t < std::min((int)constraints[constraint.car].size(), constraint.time + 1); t++) {
00042         for (auto c : constraints[constraint.car][t]) {
00043             if (c == constraint) {
00044                 return true;
00045             }
00046         }
00047     }
00048
00049     return false;
00050 }
00051
00052 ConstraintController ConstraintController::copy() {
00053     ConstraintController cc;
00054
00055     cc.constraints = constraints;
00056
00057     return cc;
00058 }
00059
00060 ConstraintController ConstraintController::copy(std::vector<int> cars) {
00061     ConstraintController cc;
00062
00063     for (int car : cars) {
00064         if (constraints.size() > car) {
00065             cc.constraints.push_back(constraints[car]);
00066         } else {
00067             cc.constraints.push_back(std::vector<std::vector<AStar::conflict>>());
00068         }
00069     }
00070
00071     return cc;
00072 }
00073
```

```

00074 bool ConstraintController::checkConstraints(int car, double speed, double newSpeed, double time,
CityGraph::point from,
00075                                     CityGraph::neighbor to) {
00076
00077     Dubins dubin = Dubins(from, to, speed, newSpeed);
00078     float duration = 2 * to.distance / (speed + newSpeed);
00079
00080     int tMin = std::round(time / SIM_STEP_TIME);
00081     tMin = tMin < 0 ? 0 : tMin;
00082     int tMax = std::round((time + duration) / SIM_STEP_TIME);
00083     if (constraints.size() > car && constraints[car].size() < tMax) {
00084         tMax = constraints[car].size();
00085     }
00086
00087     if (constraints.size() <= car || constraints[car].size() <= tMin) {
00088         return false;
00089     }
00090
00091     double distance = to.distance;
00092     double acc = (std::pow(newSpeed, 2) - std::pow(speed, 2)) / (2 * distance);
00093     auto xFun = [&](double t) { return (0.5 * acc * t * t + speed * t) / distance; };
00094
00095     for (int t = tMin; t < tMax; t += 1) {
00096         bool precise = false;
00097         sf::Vector2f pos = from.position + (to.point.position - from.position) * (float)xFun(t *
SIM_STEP_TIME - time);
00098         CityGraph::point point;
00099         if (constraints[car].size() <= t)
00100             continue;
00101
00102         for (auto c : constraints[car][t]) {
00103             if (precise) {
00104                 if (carConflict(point.position, point.angle, c.point.position, c.point.angle)) {
00105                     return true;
00106                 }
00107             } else {
00108                 sf::Vector2f diff = pos - c.point.position;
00109                 double dist = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00110                 if (dist < 2 * CAR_LENGTH) {
00111                     precise = true;
00112                     point = dubin.point(t * SIM_STEP_TIME - time);
00113                     if (carConflict(point.position, point.angle, c.point.position, c.point.angle)) {
00114                         return true;
00115                     }
00116                 }
00117             }
00118         }
00119     }
00120
00121     return false;
00122 }

```

4.37 dataManager.cpp File Reference

Data manager.

```

#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <spdlog/spdlog.h>
#include "cityGraph.h"
#include "cityMap.h"
#include "dataManager.h"
#include "manager.h"

```

4.37.1 Detailed Description

Data manager.

This file contains the implementation of the [DataManager](#) class.

Definition in file [dataManager.cpp](#).

4.38 dataManager.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include <filesystem>
00008 #include <fstream>
00009 #include <iostream>
00010 #include <random>
00011 #include <string>
00012
00013 #include <spdlog/spdlog.h>
00014
00015 #include "cityGraph.h"
00016 #include "cityMap.h"
00017 #include "dataManager.h"
00018 #include "manager.h"
00019
00020 DataManager::DataManager(std::string filename) {
00021     // Create /data folder if it doesn't exist
00022     if (!std::filesystem::exists("data")) {
00023         spdlog::debug("Creating data folder");
00024         std::filesystem::create_directory("data");
00025     }
00026 }
00027
00028 void DataManager::createData(int numData, int numCarsMin, int numCarsMax, std::string mapName) {
00029     // If numData is less than 1, default to a very high number (as in your original code).
00030     numData = numData < 1 ? INT_MAX : numData;
00031
00032     // Remove file extension from mapName to construct the output filename.
00033     std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
00034     std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
00035         (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";
00036
00037     // Load the city map.
00038     CityMap cityMap;
00039     cityMap.loadFile("assets/map/" + mapName);
00040
00041     // Create the city graph.
00042     CityGraph cityGraph;
00043     cityGraph.createGraph(cityMap);
00044
00045     // Open the output file in append mode.
00046     std::ofstream file;
00047     file.open(filename, std::ios::app);
00048     if (!file.is_open()) {
00049         spdlog::error("Failed to open file {}", filename);
00050         return;
00051     }
00052
00053     std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
00054     std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);
00055
00056     for (int i = 0; i < numData; i += 1) {
00057         int numCars = dist(rng);
00058
00059         Manager manager(cityGraph, cityMap, false);
00060         auto resData = manager.createCarsCBS(numCars);
00061         if (!resData.first) {
00062             spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
00063             i--;
00064             continue;
00065         }
00066
00067         data validResData = resData.second;
00068
00069         file << validResData.numCars << " " << validResData.carDensity;
00070         for (auto speed : validResData.carAvgSpeed) {
00071             file << " " << speed;
00072         }
00073         file << std::endl;
00074
00075         if (numData == INT_MAX) {
00076             spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars,
00077                 validResData.carDensity);
00078         } else {
00079             spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData, validResData.numCars,
00080                 validResData.carDensity);
00081         }
00082     }
00083     file.close();
00084 }

```

4.39 dubins.cpp File Reference

Dubins path implementation.

```
#include "dubins.h"
#include "utils.h"
```

4.39.1 Detailed Description

Dubins path implementation.

This file contains the implementation of the [Dubins](#) class. It is used to calculate the path between two points in the city graph. It will be used to render cars in the city and check for collisions.

Definition in file [dubins.cpp](#).

4.40 dubins.cpp

[Go to the documentation of this file.](#)

```
00001
00008 #include "dubins.h"
00009 #include "utils.h"
00010
00011 Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end)
00012     : Dubins(start, end, CAR_MAX_SPEED_MS, CAR_MAX_SPEED_MS) {}
00013
00014 Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed)
00015     : Dubins(start, end, startSpeed, CAR_MAX_SPEED_MS) {}
00016
00017 // The distance needed to reach the maximum speed
00018 double distanceToMaxSpeed = (std::pow(CAR_MAX_SPEED_MS, 2) - std::pow(startSpeed, 2)) / (2 *
CAR_ACCELERATION);
00019
00020 double dist = distance();
00021 if (dist == 0) {
00022     this->avgSpeed = CAR_MAX_SPEED_MS;
00023     return;
00024 }
00025
00026 if (dist < distanceToMaxSpeed) {
00027     this->avgSpeed = CAR_MAX_SPEED_MS;
00028 } else {
00029     double avg = (startSpeed + CAR_MAX_SPEED_MS) / 2;
00030     this->avgSpeed = (avg * distanceToMaxSpeed + CAR_MAX_SPEED_MS * (dist - distanceToMaxSpeed)) /
dist;
00031 }
00032 }
00033
00034 Dubins::Dubins(CityGraph::point start, CityGraph::neighbor end, double startSpeed, double endSpeed) {
00035     this->startPoint = start;
00036     this->endPoint = end;
00037     this->startSpeed = startSpeed;
00038     this->endSpeed = endSpeed;
00039     this->avgSpeed = (startSpeed + endSpeed) / 2;
00040
00041     this->space = new ob::DubinsStateSpace(this->endPoint.turningRadius);
00042
00043     ob::RealVectorBounds bounds(2);
00044     space->setBounds(bounds);
00045
00046     this->start = space->allocState();
00047     this->end = space->allocState();
00048
00049     this->start->as<ob::DubinsStateSpace::StateType>()->setXY(start.position.x, start.position.y);
00050     this->start->as<ob::DubinsStateSpace::StateType>()->setYaw(start.angle);
00051
00052     this->end->as<ob::DubinsStateSpace::StateType>()->setXY(end.point.position.x, end.point.position.y);
00053     this->end->as<ob::DubinsStateSpace::StateType>()->setYaw(end.point.angle);
00054 }
00055
00056 Dubins::~Dubins() {
00057     space->freeState(start);
00058     space->freeState(end);
00059     delete space;
00060 }
00061
00062 double Dubins::time() { return this->distance() / avgSpeed; }
00063
00064 CityGraph::point Dubins::point(double time) {
00065     double distance = this->distance();
00066     double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);
```

```

00067     auto xFun = [distance, acc, this](double t) { return (0.5 * acc * t * t + this->startSpeed * t) /
distance; };
00068
00069     ob::State *state = space->allocState();
00070     space->interpolate(start, end, xFun(time), state);
00071
00072     double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
00073     double y = state->as<ob::DubinsStateSpace::StateType>()->getY();
00074     double yaw = state->as<ob::DubinsStateSpace::StateType>()->getYaw();
00075
00076     space->freeState(state);
00077
00078     CityGraph::point point;
00079     point.position = {(float)x, (float)y};
00080     point.angle = yaw;
00081
00082     return point;
00083 }
00084
00085 std::vector<CityGraph::point> Dubins::path() {
00086     std::vector<CityGraph::point> path;
00087     double time = this->time();
00088     for (double t = 0; t < time; t += SIM_STEP_TIME) {
00089         path.push_back(this->point(t));
00090     }
00091
00092     return path;
00093 }
00094
00095 DubinsPath::DubinsPath(std::vector<AStar::node> path) : path_(path) {}
00096
00097 std::vector<CityGraph::point> DubinsPath::path() {
00098     if (pathProcessed_.empty())
00099         process();
00100
00101     return pathProcessed_;
00102 }
00103
00104 void DubinsPath::process() {
00105     pathProcessed_.clear();
00106     double t = 0;
00107     double prevTime = 0;
00108
00109     for (int i = 1; i < (int)path_.size(); i++) {
00110         AStar::node prevNode = path_[i - 1];
00111         AStar::node node = path_[i];
00112
00113         CityGraph::point start = node.arcFrom.first;
00114         CityGraph::neighbor end = node.arcFrom.second;
00115
00116         Dubins dubins(start, end, prevNode.speed, node.speed);
00117         double time = dubins.time();
00118
00119         if (t >= prevTime + time) {
00120             continue;
00121         }
00122
00123         while (t < prevTime + time) {
00124             pathProcessed_.push_back(dubins.point(t - prevTime));
00125             t += SIM_STEP_TIME;
00126         }
00127
00128         prevTime += time;
00129     }
00130 }

```

4.41 fileSelector.cpp File Reference

File selector implementation.

```

#include "fileSelector.h"
#include <filesystem>
#include <iostream>
#include <spdlog/spdlog.h>
#include <termios.h>
#include <unistd.h>
#include <vector>

```

4.41.1 Detailed Description

File selector implementation.

This file contains the implementation of the [FileSelector](#) class. It is used to select a file from a folder.

Definition in file [fileSelector.cpp](#).

4.42 fileSelector.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include "fileSelector.h"
00008
00009 #include <filesystem>
00010 #include <iostream>
00011 #include <spdlog/spdlog.h>
00012 #include <termios.h>
00013 #include <unistd.h>
00014 #include <vector>
00015
00016 namespace fs = std::filesystem;
00017
00018 void FileSelector::loadFiles() {
00019     files.clear();
00020     for (const auto &entry : fs::directory_iterator(folderPath)) {
00021         if (entry.is_regular_file() && entry.path().extension() == ".osm") {
00022             files.push_back(entry.path().filename().string());
00023         }
00024     }
00025     std::sort(files.begin(), files.end());
00026 }
00027
00028 char FileSelector::getKeyPress() {
00029     struct termios oldt, newt;
00030     char ch;
00031     tcgetattr(STDIN_FILENO, &oldt);
00032     newt = oldt;
00033     newt.c_lflag &= ~(ICANON | ECHO);
00034     tcsetattr(STDIN_FILENO, TCSANOW, &newt);
00035     ch = getchar();
00036     tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
00037     return ch;
00038 }
00039
00040 void FileSelector::moveCursorUp() {
00041     if (selectedIndex > 0) {
00042         std::cout << "\033[2K\r" << files[selectedIndex] << std::flush;
00043         selectedIndex--;
00044         std::cout << "\033[A\033[2K\r" << files[selectedIndex] << std::flush;
00045     }
00046 }
00047
00048 void FileSelector::moveCursorDown() {
00049     if (selectedIndex < files.size() - 1) {
00050         std::cout << "\033[2K\r" << files[selectedIndex] << std::flush;
00051         selectedIndex++;
00052         std::cout << "\033[B\033[2K\r" << files[selectedIndex] << std::flush;
00053     }
00054 }
00055
00056 void FileSelector::displayFiles() {
00057     std::cout << "Use UP/DOWN arrow keys to navigate, ENTER to select:\n";
00058     for (size_t i = 0; i < files.size(); i++) {
00059         if (i == selectedIndex) {
00060             std::cout << ">" << files[i] << "\n";
00061         } else {
00062             std::cout << " " << files[i] << "\n";
00063         }
00064     }
00065     std::cout << "\033[" << files.size() << "A";
00066 }
00067
00068 std::string FileSelector::selectFile() {
00069     std::cout << "\033[?25l";
00070     if (files.empty()) {
00071         spdlog::error("No .osm files found in the folder: {}", folderPath);
00072         return "";
00073     }
00074     displayFiles();
00075     while (true) {
00076         char key = getKeyPress();
00077         if (key == 27) {

```

```

00080         if (getKeyPress() == '[') {
00081             switch (getKeyPress()) {
00082                 case 'A':
00083                     moveCursorUp();
00084                     break;
00085                 case 'B':
00086                     moveCursorDown();
00087                     break;
00088             }
00089         }
00090     } else if (key == '\n') {
00091         std::cout << "\033[" << selectedIndex + 1 << "A\033[2K\r" << std::flush;
00092         std::cout << "\033[?25h";
00093         spdlog::info("Selected file: {}", files[selectedIndex]);
00094         return files[selectedIndex];
00095     }
00096 }
00097 }

```

4.43 main.cpp File Reference

Main file.

```

#include "spdlog/spdlog.h"
#include <SFML/Graphics.hpp>
#include "cityMap.h"
#include "config.h"
#include "dataManager.h"
#include "fileSelector.h"
#include "manager.h"
#include "renderer.h"
#include "test.h"

```

4.43.1 Detailed Description

Main file.

This file contains the main function of the project. It is used to run the simulation and create data.

Definition in file [main.cpp](#).

4.44 main.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include "spdlog/spdlog.h"
00008 #include <SFML/Graphics.hpp>
00009
00010 #include "cityMap.h"
00011 #include "config.h"
00012 #include "dataManager.h"
00013 #include "fileSelector.h"
00014 #include "manager.h"
00015 #include "renderer.h"
00016 #include "test.h"
00017
00018 int main(int nArgs, char **args) {
00019     srand(time(NULL));
00020     spdlog::set_pattern("[%d-%m-%C %H:%M:%S.%e] [%^%l%$] [thread %t] %v");
00021
00022     if (nArgs < 1) {
00023         spdlog::error("Usage: {} \"data\" [numCarsMin] [numCarsMax] [numData] || {} \"run\" [numCars]",
00024             args[0]);
00025         return 1;
00026     }
00027     bool data = args[1] == std::string("data");
00028     int runNumCars = 10;
00029     int dataNumCarsMin = 10;
00030     int dataNumCarsMax = 15;
00031     int dataNumData = -1;
00032
00033     if (nArgs > 2) {
00034         runNumCars = std::stoi(args[2]);
00035         dataNumCarsMin = std::stoi(args[2]);
00036     }
00037     if (nArgs > 3) {

```

```

00038     dataNumCarsMax = std::stoi(args[3]);
00039 }
00040 if (nArgs > 4) {
00041     dataNumData = std::stoi(args[4]);
00042 }
00043
00044 FileSelector fileSelector("assets/map");
00045 std::string mapFile = fileSelector.selectFile();
00046 // std::string mapFile = "small01.osm";
00047
00048 if (ENVIRONMENT == 0 && false) {
00049     spdlog::set_level(spdlog::level::debug);
00050     Test test;
00051     test.runTests();
00052 } else {
00053     spdlog::set_level(spdlog::level::info);
00054 }
00055
00056 if (data) {
00057     spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
00058 dataNumData,
00059         dataNumCarsMin, dataNumCarsMax);
00060
00061     DataManager dataManager(mapFile);
00062     dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
00063 } else {
00064     spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);
00065
00066     CityMap cityMap;
00067     cityMap.loadFile("assets/map/" + mapFile);
00068
00069     CityGraph cityGraph;
00070     cityGraph.createGraph(cityMap);
00071
00072     Manager manager(cityGraph, cityMap, true);
00073     manager.createCarsCBS(runNumCars);
00074
00075     Renderer renderer;
00076     renderer.startRender(cityMap, cityGraph, manager);
00077 }
00078 return 0;
00079 }

```

4.45 manager.cpp File Reference

Implementation of the [Manager](#) class.

```

#include "manager.h"
#include "aStar.h"
#include <iostream>
#include <spdlog/spdlog.h>

```

4.45.1 Detailed Description

Implementation of the [Manager](#) class.

This file contains the implementation of the [Manager](#) class.

Definition in file [manager.cpp](#).

4.46 manager.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include "manager.h"
00008 #include "aStar.h"
00009
00010 #include <iostream>
00011 #include <spdlog/spdlog.h>
00012
00013 void Manager::createCarsAStar(int numCars) {
00014     if (log)
00015         spdlog::info("Creating {} AStar cars", numCars);
00016     for (int i = 0; i < numCars; i++) {
00017         Car car;
00018         cars.push_back(car);
00019     }
00020 }

```

```

00021 // Create a path for each car (random start and end points)
00022 for (int i = 0; i < numCars; i++) {
00023     bool valid = false;
00024     cars[i].chooseRandomStartEndPath(graph, map);
00025
00026     if (log)
00027         spdlog::info("Car {} assigned path with {} points", i, cars[i].getPath().size());
00028 }
00029 }
00030
00031 void Manager::moveCars() {
00032     for (Car &car : cars) {
00033         car.move();
00034     }
00035 }
00036
00037 void Manager::renderCars(sf::RenderWindow &window) {
00038     for (Car &car : cars) {
00039         car.render(window);
00040     }
00041 }
00042
00043 void Manager::toggleCarDebug(sf::Vector2f mousePos) {
00044     for (Car &car : cars) {
00045         sf::Vector2f carPos = car.getPosition();
00046         double distance = sqrt(pow(mousePos.x - carPos.x, 2) + pow(mousePos.y - carPos.y, 2));
00047         if (distance < 5.0f) {
00048             car.toggleDebug();
00049         }
00050     }
00051 }

```

4.47 managerCBS.cpp File Reference

CBS algorithm implementation.

```

#include "manager.h"
#include "priorityQueue.h"
#include "renderer.h"
#include "utils.h"
#include <iostream>
#include <numeric>
#include <spdlog/spdlog.h>

```

4.47.1 Detailed Description

CBS algorithm implementation.

This file contains the implementation of the CBS algorithm. It is used to resolve conflicts between cars.

Definition in file [managerCBS.cpp](#).

4.48 managerCBS.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include "manager.h"
00008 #include "priorityQueue.h"
00009 #include "renderer.h"
00010 #include "utils.h"
00011
00012 #include <iostream>
00013 #include <numeric>
00014 #include <spdlog/spdlog.h>
00015
00016 std::pair<bool, DataManager::data> Manager::createCarsCBS(int numCars) {
00017     this->createCarsAStar(numCars);
00018     this->numCars = numCars;
00019     bool valid = true;
00020
00021     ConstraintController constraints;
00022
00023     if (log)
00024         spdlog::info("Creating {} CBS cars", numCars);
00025
00026     CBSNode node = processCBS(constraints, 0);
00027     if (!node.hasResolved) {
00028         if (log)

```

```

00029         spdlog::error("CBS could not resolve all conflicts");
00030         return std::make_pair(false, DataManager::data());
00031     } else {
00032         if (log)
00033             spdlog::info("CBS resolved all conflicts");
00034     }
00035
00036     // Check if conflicts remain
00037     for (int i = 0; i < numCars; i++) {
00038         for (int j = i + 1; j < numCars; j++) {
00039             int tMin = std::min(cars[i].getPath().size(), cars[j].getPath().size());
00040             for (int t = 0; t < tMin; t++) {
00041                 sf::Vector2f diff = cars[i].getPath()[t] - cars[j].getPath()[t];
00042
00043                 double width = graph.getWidth();
00044                 double height = graph.getHeight();
00045                 auto outOfBounds = [&](sf::Vector2f p) {
00046                     return p.x + CAR_LENGTH < 0 || p.y + CAR_LENGTH < 0 || p.x > width + CAR_LENGTH || p.y >
height + CAR_LENGTH;
00047                 };
00048
00049                 if (outOfBounds(cars[i].getPath()[t]) || outOfBounds(cars[j].getPath()[t])) {
00050                     continue;
00051                 }
00052
00053                 if (std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1) {
00054                     if (log)
00055                         spdlog::error("Cars {} and {} still have a conflict at time {} ({}), ({}),", i, j, t *
SIM_STEP_TIME,
cars[i].getPath()[t].x, cars[i].getPath()[t].y);
00056
00057                     valid = false;
00058                 }
00059             }
00060         }
00061     }
00062
00063     if (!valid) {
00064         return std::make_pair(false, DataManager::data());
00065     }
00066
00067     DataManager::data data;
00068     data.numCars = 0;
00069     data.carAvgSpeed.clear();
00070
00071     for (int i = 0; i < numCars; i++) {
00072         double avgSpeed = cars[i].getAverageSpeed(graph);
00073         if (avgSpeed <= 0.01)
00074             continue;
00075
00076         data.carAvgSpeed.push_back(avgSpeed);
00077         data.numCars++;
00078     }
00079
00080     if (data.numCars == 0) {
00081         return std::make_pair(false, DataManager::data());
00082     }
00083
00084     data.carDensity = 1000000 * data.numCars / (graph.getWidth() * graph.getHeight());
00085
00086     return std::make_pair(true, data);
00087 }
00088
00089 // Split the node into 2 subnodes
00090 Manager::CBSNode Manager::createSubCBS(CBSNode &node, int subNodeDepth) {
00091     int numCars = (int)node.paths.size();
00092     int numCars1 = numCars / 2;
00093     int numCars2 = numCars - numCars1;
00094
00095     std::vector<Car> cars1;
00096     std::vector<Car> cars2;
00097
00098     std::vector<int> cars1Index;
00099     std::vector<int> cars2Index;
00100
00101     for (int i = 0; i < numCars1; i++) {
00102         cars1.push_back(cars[i]);
00103         cars1Index.push_back(i);
00104     }
00105     for (int i = numCars1; i < numCars; i++) {
00106         cars2.push_back(cars[i]);
00107         cars2Index.push_back(i);
00108     }
00109
00110     ConstraintController constraints1 = node.constraints.copy(cars1Index);
00111     ConstraintController constraints2 = node.constraints.copy(cars2Index);
00112
00113     Manager manager1(graph, map, cars1, log);

```



```

00114   Manager manager2(graph, map, cars2, log);
00115
00116   CBSNode node1 = manager1.processCBS(constraints1, subNodeDepth + 1);
00117   if (!node1.hasResolved) {
00118       return node1;
00119   }
00120
00121   // Push all manager1 cars pos to manager2 constraints
00122   for (int i = 0; i < numCars1; i++) {
00123       std::vector<sf::Vector2f> path = node1.paths[i];
00124       for (int j = 0; j < (int)path.size(); j += CBS_PRECISION_FACTOR) {
00125           AStar::conflict conflict;
00126           conflict.point.position = path[j];
00127           conflict.point.angle = 0;
00128           conflict.time = j;
00129
00130           if (conflict.point.position.x < -CAR_LENGTH || conflict.point.position.y < -CAR_LENGTH ||
00131               conflict.point.position.x > graph.getWidth() + CAR_LENGTH ||
00132               conflict.point.position.y > graph.getHeight() + CAR_LENGTH) {
00133               continue;
00134           }
00135
00136           for (int k = 0; k < numCars2; k++) {
00137               conflict.car = k;
00138               constraints2.addConstraint(conflict);
00139           }
00140       }
00141   }
00142
00143   CBSNode node2 = manager2.processCBS(constraints2, subNodeDepth + 1);
00144   if (!node2.hasResolved) {
00145       return node2;
00146   }
00147
00148   // Merge the 2 managers
00149   for (int i = 0; i < numCars1; i++) {
00150       node.costs[i] = node1.costs[i];
00151       node.paths[i] = node1.paths[i];
00152       cars[i].assignExistingPath(node1.paths[i]);
00153   }
00154   for (int i = numCars1; i < numCars; i++) {
00155       node.costs[i] = node2.costs[i - numCars1];
00156       node.paths[i] = node2.paths[i - numCars1];
00157       cars[i].assignExistingPath(node2.paths[i - numCars1]);
00158   }
00159
00160   node.cost = node1.cost + node2.cost;
00161   node.depth = std::max(node1.depth, node2.depth);
00162   node.hasResolved = node1.hasResolved && node2.hasResolved;
00163
00164   return node;
00165 }
00166
00167 Manager::CBSNode Manager::processCBS(ConstraintController constraints, int subNodeDepth) {
00168     PriorityQueue<CBSNode> openSet = PriorityQueue<CBSNode>(CBS_MAX_OPENSET_SIZE);
00169
00170     CBSNode startNode;
00171     startNode.paths.resize(numCars);
00172     startNode.constraints = constraints;
00173     startNode.costs.clear();
00174     startNode.costs.resize(numCars);
00175     startNode.cost = 0;
00176     startNode.depth = 0;
00177     startNode.hasResolved = false;
00178
00179     double maxCarCost = 0;
00180
00181     for (int i = 0; i < numCars; i++) {
00182         TimedAStar aStar(cars[i].getStart(), cars[i].getEnd(), graph, &constraints, i);
00183         std::vector<AStar::node> newPath = aStar.findPath();
00184
00185         cars[i].assignPath(newPath);
00186
00187         startNode.paths[i] = cars[i].getPath();
00188
00189         double carCost = cars[i].getPathTime();
00190         startNode.costs[i] = carCost;
00191         startNode.cost += carCost;
00192
00193         maxCarCost = std::max(maxCarCost, carCost);
00194     }
00195
00196     openSet.push(startNode, startNode.cost);
00197
00198     // For logs
00199     std::vector<double> meanCosts;
00200     std::vector<double> meanDepths;

```

```

00201     std::vector<double> meanTimes;
00202     auto start = std::chrono::system_clock::now();
00203     double clockLastRefresh = 0;
00204     int numNodeProcessed = 0;
00205
00206     // While there are conflicts in the paths, resolve them
00207     while (!openSet.empty()) {
00208         auto duration =
00209             std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now() -
start).count() /
00210             1000.0;
00211
00212         numNodeProcessed++;
00213         CBSNode node = openSet.pop();
00214
00215         // if (duration > CBS_MAX_SUB_TIME) {
00216         //     CBSNode resSub = createSubCBS(node, subNodeDepth);
00217         //     if (resSub.hasResolved) {
00218         //         return resSub;
00219         //     }
00220         // }
00221
00222         std::vector<std::vector<sf::Vector2f>> paths = node.paths;
00223         double cost = node.cost;
00224         int depth = node.depth;
00225
00226         int car1, car2;
00227         sf::Vector2f p1, p2;
00228
00229         double a1, a2;
00230         int time;
00231         bool conflict = hasConflict(paths, &car1, &car2, &p1, &p2, &a1, &a2, &time);
00232
00233         if (!conflict) {
00234             for (int i = 0; i < numCars; i++) {
00235                 cars[i].assignExistingPath(node.paths[i]);
00236             }
00237             node.hasResolved = true;
00238             return node;
00239         }
00240
00241         meanCosts.push_back(cost);
00242         meanDepths.push_back(depth);
00243         meanTimes.push_back(time);
00244
00245         if (clockLastRefresh + LOG_CBS_REFRESHRATE < duration) {
00246             double meanCost = std::accumulate(meanCosts.begin(), meanCosts.end(), 0.0) / meanCosts.size();
00247             double meanDepth = std::accumulate(meanDepths.begin(), meanDepths.end(), 0.0) /
meanDepths.size();
00248             double meanTime = std::accumulate(meanTimes.begin(), meanTimes.end(), 0.0) / meanTimes.size();
00249             meanTime = meanTime * SIM_STEP_TIME;
00250
00251             double remainingTime = (maxCarCost - meanTime) * (duration / meanTime);
00252             double processPerSecond = numNodeProcessed / (duration - clockLastRefresh);
00253
00254             if (log) {
00255                 spdlog::info("Node C: {:0>6.5} | D: {:0>6.5} | CT: {:0>6.5} | SD: {} | ET: {}s | "
00256                     "ETR: ~{}s | Processed nodes: ~{:0>4.5}/s",
00257                     meanCost, meanDepth, meanTime, subNodeDepth, (int)duration, (int)remainingTime,
processPerSecond);
00258                 std::cout << "\033[A\033[2K\r";
00259             }
00260
00261             clockLastRefresh = duration;
00262             numNodeProcessed = 0;
00263
00264             meanCosts.clear();
00265             meanDepths.clear();
00266             meanTimes.clear();
00267         }
00268
00269         // Resolve conflict
00270         for (int iCar = 0; iCar < 2; iCar++) {
00271             int car = iCar == 0 ? car1 : car2;
00272
00273             AStar::conflict newConflict;
00274             newConflict.point.position = iCar == 0 ? p2 : p1;
00275             newConflict.point.angle = iCar == 0 ? a2 : a1;
00276             newConflict.time = time;
00277             newConflict.car = iCar == 0 ? car1 : car2;
00278
00279             // If already in constraints, skip
00280             if (node.constraints.hasConstraint(newConflict)) {
00281                 continue;
00282             }
00283
00284             ConstraintController newConstraints = node.constraints.copy();

```

```

00285     newConstraints.addConstraint(newConflict);
00286
00287     TimedAStar aStar(cars[car].getStart(), cars[car].getEnd(), graph, &newConstraints, car);
00288     std::vector<AStar::node> newPath = aStar.findPath();
00289
00290     if (newPath.empty()) {
00291         continue;
00292     }
00293
00294     cars[car].assignPath(newPath);
00295     double carOldCost = node.costs[car];
00296     double carNewCost = cars[car].getPathTime();
00297
00298     CBSNode newNode;
00299     newNode.paths = paths;
00300     newNode.paths[car] = cars[car].getPath();
00301     newNode.constraints = newConstraints;
00302     newNode.costs = node.costs;
00303     newNode.costs[car] = carNewCost;
00304     newNode.cost = cost - carOldCost + carNewCost;
00305     newNode.depth = depth + 1;
00306     newNode.hasResolved = false;
00307
00308     newNode.cost = 0;
00309     for (int i = 0; i < numCars; i++) {
00310         newNode.cost += newNode.costs[i];
00311     }
00312
00313     // hasConflict(newNode.paths, &car1, &car2, &p1, &p2, &a1, &a2, &time);
00314     // newNode.cost /= time;
00315
00316     // newNode.cost = 1 / (double)time;
00317
00318     openSet.push(newNode, newNode.cost);
00319 }
00320 }
00321
00322 return startNode;
00323 }
00324
00325 bool Manager::hasConflict(std::vector<std::vector<sf::Vector2f> paths, int *car1, int *car2,
sf::Vector2f *p1,
00326                          sf::Vector2f *p2, double *a1, double *a2, int *time) {
00327     int maxPathLength = 0;
00328     int numCars = (int)paths.size();
00329     for (int i = 0; i < numCars; i++) {
00330         maxPathLength = std::max(maxPathLength, (int)paths[i].size());
00331     }
00332
00333     double width = graph.getWidth();
00334     double height = graph.getHeight();
00335     auto outOfBounds = [&](sf::Vector2f p) {
00336         return p.x + CAR_LENGTH < 0 || p.y + CAR_LENGTH < 0 || p.x - CAR_LENGTH > width || p.y -
CAR_LENGTH > height;
00337     };
00338
00339     for (int t = 0; t < maxPathLength; t += CBS_PRECISION_FACTOR) {
00340         for (int i = 0; i < numCars; i++) {
00341             if (t >= (int)paths[i].size() - 1 || outOfBounds(paths[i][t]))
00342                 continue;
00343             for (int j = i + 1; j < numCars; j++) {
00344                 if (t >= (int)paths[j].size() - 1 || outOfBounds(paths[j][t]))
00345                     continue;
00346
00347                 sf::Vector2f diff = paths[i][t] - paths[j][t];
00348                 if (std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1) {
00349                     *car1 = i;
00350                     *car2 = j;
00351                     *p1 = paths[i][t];
00352                     *p2 = paths[j][t];
00353                     *a1 = std::atan2(paths[i][t + 1].y - paths[i][t].y, paths[i][t + 1].x - paths[i][t].x);
00354                     *a2 = std::atan2(paths[j][t + 1].y - paths[j][t].y, paths[j][t + 1].x - paths[j][t].x);
00355                     *time = t;
00356                     return true;
00357                 }
00358             }
00359         }
00360     }
00361
00362     return false;
00363 }

```

4.49 renderer.cpp File Reference

Implementation of the [Renderer](#) class.

```

#include <algorithm>
#include <iostream>
#include <random>
#include <vector>
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRT.h>
#include <spdlog/spdlog.h>
#include "aStar.h"
#include "config.h"
#include "renderer.h"
#include "utils.h"

```

4.49.1 Detailed Description

Implementation of the [Renderer](#) class.

This file contains the implementation of the [Renderer](#) class.

Definition in file [renderer.cpp](#).

4.50 renderer.cpp

[Go to the documentation of this file.](#)

```

00001
00007 #include <algorithm>
00008 #include <iostream>
00009 #include <random>
00010 #include <vector>
00011
00012 #include <ompl/base/State.h>
00013 #include <ompl/base/StateSpace.h>
00014 #include <ompl/base/spaces/DubinsStateSpace.h>
00015 #include <ompl/geometric/SimpleSetup.h>
00016 #include <ompl/geometric/planners/rrt/RRT.h>
00017 #include <spdlog/spdlog.h>
00018
00019 #include "aStar.h"
00020 #include "config.h"
00021 #include "renderer.h"
00022 #include "utils.h"
00023
00024 namespace ob = ompl::base;
00025
00026 void Renderer::startRender(const CityMap &cityMap, const CityGraph &cityGraph, Manager &manager) {
00027     window.create(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT), "City Map");
00028
00029     // Set the view to the center of the city map, allowing some basic camera movement
00030     // Arrow to move the camera, + and - to zoom in and out
00031     double height = cityMap.getHeight();
00032     double width = cityMap.getWidth();
00033     sf::View view(sf::FloatRect(0, 0, width, height));
00034     // Reset view function
00035     auto resetView = [&]() {
00036         double screenRatio = window.getSize().x / (double)window.getSize().y;
00037         double cityRatio = width / height;
00038         view.setCenter(width / 2, height / 2);
00039         if (screenRatio > cityRatio) {
00040             view.setSize(height * screenRatio, height);
00041         } else {
00042             view.setSize(width, width / screenRatio);
00043         }
00044         window.setView(view);
00045     };
00046
00047     resetView();
00048     time = 0;
00049
00050     sf::Clock clockCars;
00051     bool speedUp = false;
00052     bool pause = true;
00053
00054     while (true) {
00055         sf::Event event;

```

```

00056     while (window.pollEvent(event)) {
00057         if (event.type == sf::Event::Closed) {
00058             window.close();
00059             return;
00060         }
00061
00062         if (event.type == sf::Event::MouseButtonPressed) {
00063             if (event.mouseButton.button == sf::Mouse::Left) {
00064                 sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
00065                 manager.toggleCarDebug(mousePos);
00066             }
00067         }
00068
00069         if (event.type == sf::Event::KeyPressed) {
00070             if (event.key.code == sf::Keyboard::Escape) {
00071                 window.close();
00072                 return;
00073             }
00074             if (event.key.code == sf::Keyboard::Up) {
00075                 view.move(0, -height * MOVE_SPEED);
00076             }
00077             if (event.key.code == sf::Keyboard::Down) {
00078                 view.move(0, height * MOVE_SPEED);
00079             }
00080             if (event.key.code == sf::Keyboard::Left) {
00081                 view.move(-width * MOVE_SPEED, 0);
00082             }
00083             if (event.key.code == sf::Keyboard::Right) {
00084                 view.move(width * MOVE_SPEED, 0);
00085             }
00086             if (event.key.code == sf::Keyboard::Equal) {
00087                 view.zoom(1.0f - ZOOM_SPEED);
00088             }
00089             if (event.key.code == sf::Keyboard::Dash) {
00090                 view.zoom(1.0f + ZOOM_SPEED);
00091             }
00092             if (event.key.code == sf::Keyboard::R) {
00093                 resetView();
00094                 spdlog::debug("View reset");
00095             }
00096             if (event.key.code == sf::Keyboard::D) {
00097                 debug = !debug;
00098                 spdlog::debug("Debug mode: {}", debug);
00099             }
00100             if (event.key.code == sf::Keyboard::S) {
00101                 speedUp = !speedUp;
00102             }
00103             if (event.key.code == sf::Keyboard::P) {
00104                 pause = !pause;
00105             }
00106         }
00107
00108         // If resizing the window, reset the view
00109         if (event.type == sf::Event::Resized) {
00110             resetView();
00111         }
00112     }
00113
00114     window.setView(view);
00115     window.clear(sf::Color(247, 246, 242));
00116     renderCityMap(cityMap);
00117     renderManager(manager);
00118     if (!pause) {
00119         if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
00120             (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
00121             time += SIM_STEP_TIME;
00122             manager.moveCars();
00123             clockCars.restart();
00124         }
00125     }
00126     if (debug) {
00127         renderCityGraph(cityGraph, view);
00128     }
00129     // Remove outside the border (draw blank)
00130     sf::RectangleShape rectangle(sf::Vector2f(width, height));
00131     rectangle.setFillColor(sf::Color(247, 246, 242));
00132
00133     float w = width;
00134     float h = height;
00135
00136     std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h},
00137 {-w, 0}};
00138     for (auto b : border) {
00139         rectangle.setPosition(b);
00140         window.draw(rectangle);
00141     }

```

```

00142     renderTime();
00143     window.display();
00144 }
00145 }
00146
00147 void Renderer::renderCityMap(const CityMap &cityMap) {
00148     // Draw buildings
00149     std::vector<sf::Color> randomBuildingColors = {
00150         sf::Color(233, 234, 232), sf::Color(238, 231, 210), sf::Color(230, 229, 226), sf::Color(236,
234, 230),
00151         sf::Color(230, 223, 216), sf::Color(230, 234, 236), sf::Color(210, 215, 222)};
00152
00153     std::vector<sf::Color> greenAreaColor = {sf::Color(184, 230, 144), sf::Color(213, 240, 193)};
00154
00155     sf::Color waterColor(139, 214, 245);
00156
00157     auto greenAreas = cityMap.getGreenAreas();
00158     for (int i = 0; i < (int)greenAreas.size(); i++) {
00159         const auto &greenArea = greenAreas[i];
00160         auto points = greenArea.points;
00161         sf::ConvexShape convex;
00162         convex.setPointCount(points.size());
00163         for (size_t i = 0; i < points.size(); i++) {
00164             convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00165         }
00166         convex.setFillColor(greenAreaColor[greenArea.type]);
00167
00168         window.draw(convex);
00169     }
00170
00171     auto waterAreas = cityMap.getWaterAreas();
00172     for (int i = 0; i < (int)waterAreas.size(); i++) {
00173         const auto &waterArea = waterAreas[i];
00174         auto points = waterArea.points;
00175         sf::ConvexShape convex;
00176         convex.setPointCount(points.size());
00177         for (size_t i = 0; i < points.size(); i++) {
00178             convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00179         }
00180         convex.setFillColor(waterColor);
00181
00182         window.draw(convex);
00183     }
00184
00185     auto buildings = cityMap.getBuildings();
00186     for (int i = 0; i < (int)buildings.size(); i++) {
00187         const auto &building = buildings[i];
00188         auto points = building.points;
00189         sf::ConvexShape convex;
00190         convex.setPointCount(points.size());
00191         for (size_t i = 0; i < points.size(); i++) {
00192             convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00193         }
00194         convex.setFillColor(randomBuildingColors[i % randomBuildingColors.size()]);
00195
00196         window.draw(convex);
00197     }
00198
00199     // Draw roads
00200     sf::Color roadColor(194, 201, 202);
00201     for (const auto &road : cityMap.getRoads()) {
00202         for (const auto &segment : road.segments) {
00203             sf::Vector2f basedP1(segment.p1.x, segment.p1.y);
00204             sf::Vector2f basedP2(segment.p2.x, segment.p2.y);
00205
00206             double angle = segment.angle;
00207
00208             sf::Vector2f widthVec(sin(angle), -cos(angle));
00209             widthVec *= (float)road.width / 2;
00210
00211             sf::Vector2f p1 = basedP1 + widthVec;
00212             sf::Vector2f p2 = basedP1 - widthVec;
00213             sf::Vector2f p3 = basedP2 - widthVec;
00214             sf::Vector2f p4 = basedP2 + widthVec;
00215
00216             sf::ConvexShape convex;
00217             convex.setPointCount(4);
00218             convex.setPoint(0, p1);
00219             convex.setPoint(1, p2);
00220             convex.setPoint(2, p3);
00221             convex.setPoint(3, p4);
00222
00223             convex.setFillColor(roadColor);
00224
00225             window.draw(convex);
00226
00227             // Draw a circle at the start end end of the road (for filling the gap)

```

```

00228     double radius = road.width / 2;
00229     sf::CircleShape circle(radius);
00230     circle.setFillColor(roadColor);
00231     circle.setPosition(basedP1.x - radius, basedP1.y - radius);
00232     window.draw(circle);
00233     circle.setPosition(basedP2.x - radius, basedP2.y - radius);
00234     window.draw(circle);
00235 }
00236 }
00237
00238 // Draw intersections
00239 if (debug) {
00240     for (const auto &intersection : cityMap.getIntersections()) {
00241         double radius = intersection.radius;
00242         sf::CircleShape circle(radius);
00243         circle.setFillColor(sf::Color(0, 255, 0, 50));
00244         circle.setPosition(intersection.center.x - radius, intersection.center.y - radius);
00245         window.draw(circle);
00246     }
00247 }
00248 }
00249
00250 void Renderer::renderCityGraph(const CityGraph &cityGraph, const sf::View &view) {
00251     std::unordered_set<CityGraph::point> graphPoints = cityGraph.getGraphPoints();
00252     std::unordered_map<CityGraph::point, std::vector<CityGraph::neighbor> neighbors =
cityGraph.getNeighbors();
00253
00254     // Draw a line between each point and its neighbors
00255     for (const auto &point : graphPoints) {
00256         for (const auto &neighbor : neighbors[point]) {
00257             if (!neighbor.isRightWay)
00258                 continue;
00259
00260             double radius = turningRadius(neighbor.maxSpeed);
00261             auto space = ob::DubinsStateSpace(radius);
00262             ob::RealVectorBounds bounds(2);
00263             space.setBounds(bounds);
00264
00265             // Draw only if one of the points is inside the view
00266             sf::Vector2f viewCenter = view.getCenter();
00267             sf::Vector2f viewSize = view.getSize();
00268             sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
00269             sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;
00270
00271             if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
00272                 continue;
00273             }
00274             if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
00275                 continue;
00276             }
00277
00278             ob::State *start = space.allocState();
00279             ob::State *end = space.allocState();
00280
00281             start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
00282             start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle);
00283
00284             end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x,
neighbor.point.position.y);
00285             end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle);
00286
00287             // Draw the Dubins curve
00288             double step = CELL_SIZE / 2.0f;
00289             double distance = space.distance(start, end);
00290             int numSteps = distance / step;
00291             sf::Vector2f lastPosition;
00292             sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);
00293
00294             for (int k = 0; k < numSteps; k++) {
00295                 if (k == 0) {
00296                     lastPosition = {point.position.x, point.position.y};
00297                     continue;
00298                 }
00299
00300                 ob::State *state = space.allocState();
00301                 space.interpolate(start, end, (double)k / (double)numSteps, state);
00302
00303                 double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
00304                 double y = state->as<ob::DubinsStateSpace::StateType>()->getY();
00305
00306                 double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y,
2));
00307                 double angle = atan2(y - lastPosition.y, x - lastPosition.x) * 180 / M_PI;
00308
00309                 // Draw an arrow between the points
00310                 drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor,
false);

```

```

00311
00312     lastPosition = {(float)x, (float)y};
00313 }
00314
00315     continue;
00316     // Write the speed of the point
00317     sf::Text text;
00318     sf::Font font;
00319     font.loadFromFile("assets/fonts/arial.ttf");
00320     text.setFont(font);
00321     text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
00322     text.setCharacterSize(24);
00323     text.setFillColor(sf::Color::Black);
00324     text.setOutlineColor(sf::Color::White);
00325     text.setOutlineThickness(1.0f);
00326     text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
00327     text.setScale(0.02f, 0.02f);
00328     text.setOrigin(text.getLocalBounds().width / 2.0f, text.getLocalBounds().height / 2.0f);
00329     window.draw(text);
00330 }
00331
00332     // Draw a dot at each points
00333     double size = 0.3;
00334     sf::CircleShape circle(size);
00335     circle.setFillColor(sf::Color(255, 0, 0, 70));
00336     circle.setPosition(point.position.x - size, point.position.y - size);
00337     window.draw(circle);
00338 }
00339 }
00340
00341 void Renderer::renderManager(Manager &manager) { manager.renderCars(window); }
00342
00343 void Renderer::renderTime() {
00344     // At the top right corner of the view (keep the same size even if the view is resized)
00345     sf::Text text;
00346     sf::Font font = loadFont();
00347     sf::Vector2f viewSize = window.getView().getSize();
00348     text.setFont(font);
00349     text.setCharacterSize(24);
00350     text.setFillColor(sf::Color::White);
00351     text.setPosition(window.getView().getCenter() + sf::Vector2f(viewSize.x / 2, -viewSize.y / 2) +
00352         sf::Vector2f(-viewSize.x * 0.01f, viewSize.y * 0.01f));
00353     text.setString(std::to_string((int)time) + " s");
00354     text.setOutlineColor(sf::Color::Black);
00355     text.setOutlineThickness(1.0f);
00356     text.scale(viewSize.x * 0.001f, viewSize.x * 0.001f);
00357     text.setOrigin(text.getLocalBounds().width, 0);
00358     window.draw(text);
00359 }

```

4.51 test.cpp File Reference

A file for testing the project.

```

#include "test.h"
#include "spdlog/spdlog.h"
#include "tinycl2.h"
#include <SFML/Graphics.hpp>

```

4.51.1 Detailed Description

A file for testing the project.

Definition in file [test.cpp](#).

4.52 test.cpp

[Go to the documentation of this file.](#)

```

00001
00002 #include "test.h"
00003 #include "spdlog/spdlog.h"
00004 #include "tinycl2.h"
00005 #include <SFML/Graphics.hpp>
00006
00007 void Test::runTests() {
00008     testSpdlog();
00009     testTinyXML2();
00010     testSFML();
00011 }

```



```

00015
00016 void Test::testSpdlog() {
00017     try {
00018         spdlog::debug("Testing spdlog...");
00019         spdlog::debug("spdlog is working as expected.");
00020     } catch (const std::exception &e) {
00021         throw std::runtime_error("spdlog is not working as expected.");
00022     }
00023 }
00024
00025 void Test::testTinyXML2() {
00026     try {
00027         spdlog::debug("Testing TinyXML2...");
00028         tinyxml2::XMLDocument xmlDoc;
00029         xmlDoc.Parse("<root></root>");
00030         if (xmlDoc.Error()) {
00031             spdlog::error("TinyXML2 is not working as expected.");
00032             throw std::runtime_error("TinyXML2 is not working as expected.");
00033         }
00034         spdlog::debug("TinyXML2 is working as expected.");
00035     } catch (const std::exception &e) {
00036         spdlog::error("TinyXML2 is not working as expected.");
00037         throw std::runtime_error("TinyXML2 is not working as expected.");
00038     }
00039 }
00040
00041 void Test::testSFML() {
00042     try {
00043         spdlog::debug("Testing SFML...");
00044         sf::RenderWindow window(sf::VideoMode(100, 100), "Test");
00045         if (!window.isOpen()) {
00046             spdlog::error("SFML is not working as expected.");
00047             throw std::runtime_error("SFML is not working as expected.");
00048         }
00049         window.close();
00050         spdlog::debug("SFML is working as expected.");
00051     } catch (const std::exception &e) {
00052         spdlog::error("SFML is not working as expected.");
00053         throw std::runtime_error("SFML is not working as expected.");
00054     }
00055 }

```

4.53 timedAStar.cpp File Reference

Timed A* algorithm implementation.

```

#include "aStar.h"
#include "config.h"
#include "dubins.h"
#include "utils.h"
#include <spdlog/spdlog.h>
#include <unordered_set>

```

4.53.1 Detailed Description

Timed A* algorithm implementation.

This file contains the implementation of the Timed A* algorithm. It is used to find the shortest path between two points in a graph with time constraints.

Definition in file [timedAStar.cpp](#).

4.54 timedAStar.cpp

[Go to the documentation of this file.](#)

```

00001
00008 #include "aStar.h"
00009 #include "config.h"
00010 #include "dubins.h"
00011 #include "utils.h"
00012
00013 #include <spdlog/spdlog.h>
00014 #include <unordered_set>
00015
00016 TimedAStar::TimedAStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph,
00017                       ConstraintController *conflicts, int carIndex) {
00018     this->start.point = start;
00019     this->start.speed = 0;

```

```

00020     this->end.point = end;
00021     this->end.speed = 0;
00022     this->graph = cityGraph;
00023     this->conflicts = conflicts;
00024     this->carIndex = carIndex;
00025 }
00026
00027 void TimedAStar::process() {
00028     path.clear();
00029
00030     std::unordered_map<AStar::node, AStar::node> cameFrom;
00031     std::unordered_map<AStar::node, double> gScore;
00032     std::unordered_map<AStar::node, double> fScore;
00033
00034     auto heuristic = [&](const AStar::node &a) {
00035         sf::Vector2f diff = end.point.position - a.point.position;
00036         double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00037         return distance / CAR_MAX_SPEED_MS;
00038     };
00039
00040     CityGraph::neighbor end_;
00041     end_.point = end.point;
00042     end_.maxSpeed = CAR_MAX_SPEED_MS;
00043     end_.turningRadius = CAR_MIN_TURNING_RADIUS;
00044     Dubins dubins(a.point, end_, CAR_MAX_SPEED_MS, CAR_MAX_SPEED_MS);
00045     return dubins.time();
00046 };
00047
00048 auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };
00049
00050 std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSet(compare);
00051 std::unordered_set<AStar::node> isInOpenSet;
00052
00053 openSet.push(start);
00054 gScore[start] = 0;
00055 fScore[start] = heuristic(start);
00056
00057 auto neighbors = graph.getNeighbors();
00058
00059 int nbIterations = 0;
00060 while (!openSet.empty() && nbIterations++ < 1e5) {
00061     AStar::node current = openSet.top();
00062     openSet.pop();
00063     isInOpenSet.erase(current);
00064
00065     if (current.point == end.point) {
00066         AStar::node currentCopy = current;
00067
00068         while (!(currentCopy == start)) {
00069             path.push_back(currentCopy);
00070             currentCopy = cameFrom[currentCopy];
00071         }
00072         path.push_back(currentCopy);
00073         std::reverse(path.begin(), path.end());
00074         processed = true;
00075         break;
00076     }
00077
00078     for (const auto &neighborGraphPoint : neighbors[current.point]) {
00079         if (current.speed > neighborGraphPoint.maxSpeed)
00080             continue;
00081
00082         if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
00083             continue;
00084
00085         std::vector<double> newSpeeds;
00086         newSpeeds.push_back(current.speed);
00087
00088         double distance = neighborGraphPoint.distance;
00089         double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
00090         double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);
00091
00092         auto push = [&](double nSpeed) {
00093             int numSpeedDiv = 5;
00094             for (int i = 1; i < numSpeedDiv + 1; i++) {
00095                 double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
00096                 if (s < SPEED_RESOLUTION)
00097                     continue;
00098                 newSpeeds.push_back(s);
00099             }
00100         };
00101
00102         if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
00103             push(neighborGraphPoint.maxSpeed);
00104             // newSpeeds.push_back(neighborGraphPoint.maxSpeed);
00105             // newSpeeds.push_back((current.speed + neighborGraphPoint.maxSpeed) / 2);
00106         } else if (nSpeedDec < neighborGraphPoint.maxSpeed) {
00107             push(nSpeedDec);
00108             // newSpeeds.push_back(nSpeedDec);
00109         }
00110     }
00111 }

```

```

00107         // newSpeeds.push_back((current.speed + nSpeedAcc) / 2);
00108     }
00109
00110     if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) {
00111         if (nSpeedDec < 0 && current.speed > 0) {
00112             push(0);
00113             // newSpeeds.push_back(0);
00114             // newSpeeds.push_back((current.speed + 0) / 2);
00115         } else if (nSpeedDec >= 0) {
00116             push(nSpeedDec);
00117             // newSpeeds.push_back(nSpeedDec);
00118             // newSpeeds.push_back((current.speed + nSpeedDec) / 2);
00119         }
00120     }
00121
00122     AStar::node neighbor;
00123     neighbor.point = neighborGraphPoint.point;
00124     neighbor.arcFrom = {current.point, neighborGraphPoint};
00125     if (distance == 0) {
00126         neighbor.speed = current.speed;
00127         if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
00128             cameFrom[neighbor] = current;
00129             gScore[neighbor] = gScore[current];
00130             fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00131
00132             if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00133                 openSet.push(neighbor);
00134
00135                 isInOpenSet.insert(neighbor);
00136             }
00137         }
00138         continue;
00139     }
00140
00141     for (const auto &newSpeed : newSpeeds) {
00142         if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
00143             continue;
00144
00145         if (newSpeed == current.speed && newSpeed == 0)
00146             continue;
00147
00148         neighbor.speed = newSpeed;
00149
00150         double duration = 2 * distance / (current.speed + newSpeed);
00151         double tentativeGScore = gScore[current] + duration;
00152
00153         double t = gScore[current];
00154
00155         if (conflicts != nullptr &&
00156             conflicts->checkConstraints(carIndex, current.speed, newSpeed, t, current.point,
neighborGraphPoint))
00157             continue;
00158
00159         if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
00160             cameFrom[neighbor] = current;
00161             gScore[neighbor] = tentativeGScore;
00162             fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00163
00164             if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00165                 openSet.push(neighbor);
00166                 isInOpenSet.insert(neighbor);
00167             }
00168         }
00169     }
00170 }
00171 }
00172 }

```

4.55 utils.cpp File Reference

Utility functions implementation.

```

#include <spdlog/spdlog.h>
#include "car.h"
#include "utils.h"

```

Functions

- sf::Font [loadFont](#) ()

Load a font.

- bool `carsCollided` (`Car` car1, `Car` car2, int time)
- bool `carConflict` (`sf::Vector2f` carPos, double carAngle, `sf::Vector2f` confPos, double confAngle)

Check if two cars have a conflict.

4.55.1 Detailed Description

Utility functions implementation.

Definition in file [utils.cpp](#).

4.55.2 Function Documentation

`carConflict()`

```
bool carConflict (  
    sf::Vector2f carPos,  
    double carAngle,  
    sf::Vector2f confPos,  
    double confAngle)
```

Check if two cars have a conflict.

Parameters

<code>carPos</code>	The position of the car
<code>carAngle</code>	The angle of the car
<code>confPos</code>	The position of the conflicting car
<code>confAngle</code>	The angle of the conflicting car

Returns

If the cars have a conflict

Definition at line 49 of file [utils.cpp](#).

`carsCollided()`

```
bool carsCollided (  
    Car car1,  
    Car car2,  
    int time)
```

@brief Check if two cars collided

Parameters

<code>car1</code>	The first car
<code>car2</code>	The second car

Definition at line 23 of file [utils.cpp](#).

`loadFont()`

```
sf::Font loadFont ()
```

Load a font.

Returns

The font

Definition at line 13 of file [utils.cpp](#).

4.56 utils.cpp

[Go to the documentation of this file.](#)

```

00001
00005 #include <spdlog/spdlog.h>
00006
00007 #include "car.h"
00008 #include "utils.h"
00009
00010 bool fontLoaded = false;
00011 sf::Font font;
00012
00013 sf::Font loadFont() {
00014     if (!fontLoaded) {
00015         if (!font.loadFromFile("assets/fonts/arial.ttf")) {
00016             spdlog::error("Failed to load font");
00017         }
00018         fontLoaded = true;
00019     }
00020     return font;
00021 }
00022
00023 bool carsCollided(Car car1, Car car2, int time) {
00024     std::vector<sf::Vector2f> path1 = car1.getPath();
00025     std::vector<sf::Vector2f> path2 = car2.getPath();
00026     sf::Vector2f diff = path1[time] - path2[time];
00027     return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00028
00029     sf::Vector2f pos1 = path1[time];
00030     sf::Vector2f pos2 = path2[time];
00031
00032     double angle1 = atan2(path1[time + 1].y - path1[time].y, path1[time + 1].x - path1[time].x);
00033     double angle2 = atan2(path2[time + 1].y - path2[time].y, path2[time + 1].x - path2[time].x);
00034
00035     sf::Vector2f p11 = pos1 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
sin(angle1));
00036     sf::Vector2f p12 = pos1 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
sin(angle1));
00037     sf::Vector2f p21 = pos2 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
sin(angle2));
00038     sf::Vector2f p22 = pos2 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
sin(angle2));
00039
00040     bool colides = false;
00041     colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00042     colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00043     colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00044     colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00045
00046     return colides;
00047 }
00048
00049 bool carConflict(sf::Vector2f carPos, double carAngle, sf::Vector2f confPos, double confAngle) {
00050     sf::Vector2f diff = carPos - confPos;
00051     return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00052
00053     sf::Vector2f p11 = carPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle), CAR_LENGTH / 2.0f *
sin(carAngle));
00054     sf::Vector2f p12 = carPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle), CAR_LENGTH / 2.0f *
sin(carAngle));
00055     sf::Vector2f p21 = confPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle), CAR_LENGTH / 2.0f *
sin(confAngle));
00056     sf::Vector2f p22 = confPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle), CAR_LENGTH / 2.0f *
sin(confAngle));
00057
00058     bool colides = false;
00059     colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00060     colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00061     colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00062     colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00063
00064     return colides;
00065 }

```

Index

- [_aStarConflict](#), 4
- [_aStarNode](#), 5
- [_cityGraphNeighbor](#), 5
- [_cityGraphPoint](#), 6
- [_cityMapBuilding](#), 6
- [_cityMapGreenArea](#), 6
- [_cityMapIntersection](#), 7
- [_cityMapRoad](#), 7
- [_cityMapSegment](#), 8
- [_cityMapWaterArea](#), 8
- [_data](#), 8
- [_managerCBSNode](#), 9
- [addConstraint](#)
 - [ConstraintController](#), 20
- [assignExistingPath](#)
 - [Car](#), 11
- [assignPath](#)
 - [Car](#), 11
- [assignStartEnd](#)
 - [Car](#), 11
- [AStar](#), 9
 - [AStar](#), 9
 - [findPath](#), 10
- [aStar.cpp](#), 53, 54
- [aStar.h](#), 35, 36
- [Car](#), 10
 - [assignExistingPath](#), 11
 - [assignPath](#), 11
 - [assignStartEnd](#), 11
 - [chooseRandomStartEndPath](#), 12
 - [getAStarPath](#), 12
 - [getAverageSpeed](#), 12
 - [getElapsedDistance](#), 12
 - [getElapsedTime](#), 12
 - [getEnd](#), 13
 - [getPath](#), 13
 - [getPathLength](#), 13
 - [getPathTime](#), 13
 - [getPosition](#), 13
 - [getRemainingDistance](#), 13
 - [getRemainingTime](#), 14
 - [getSpeed](#), 14
 - [getSpeedAt](#), 14
 - [getStart](#), 14
 - [render](#), 14
- [car.cpp](#), 55
- [car.h](#), 37
- [carConflict](#)
 - [utils.cpp](#), 88
 - [utils.h](#), 49
- [carsCollided](#)
 - [utils.cpp](#), 88
 - [utils.h](#), 49
- [checkConstraints](#)
 - [ConstraintController](#), 21
- [chooseRandomStartEndPath](#)
 - [Car](#), 12
- [CityGraph](#), 15
 - [createGraph](#), 15
 - [getGraphPoints](#), 15
 - [getHeight](#), 15
 - [getNeighbors](#), 16
 - [getRandomPoint](#), 16
 - [getWidth](#), 16
- [cityGraph.cpp](#), 57, 58
- [cityGraph.h](#), 38, 39
- [CityMap](#), 16
 - [getBuildings](#), 17
 - [getGreenAreas](#), 17
 - [getHeight](#), 17
 - [getIntersections](#), 17
 - [getMaxLatLon](#), 18
 - [getMinLatLon](#), 18
 - [getRoads](#), 18
 - [getWaterAreas](#), 18
 - [getWidth](#), 18
 - [isCityMapLoaded](#), 18
 - [loadFile](#), 19
- [cityMap.cpp](#), 62
- [cityMap.h](#), 40
- [config.h](#), 41
- [ConstraintController](#), 19
 - [addConstraint](#), 20
 - [checkConstraints](#), 21
 - [copy](#), 21
 - [hasConstraint](#), 21
- [constraintController.cpp](#), 66, 67
- [copy](#)
 - [ConstraintController](#), 21
- [createCarsAStar](#)
 - [Manager](#), 28
- [createCarsCBS](#)
 - [Manager](#), 28
- [createData](#)
 - [DataManager](#), 22
- [createGraph](#)
 - [CityGraph](#), 15
- [createSubCBS](#)
 - [Manager](#), 28
- [DataManager](#), 22
 - [createData](#), 22
 - [DataManager](#), 22
- [dataManager.cpp](#), 68, 69
- [dataManager.h](#), 41, 42
- [distance](#)
 - [Dubins](#), 24
 - [utils.h](#), 49
- [drawArrow](#)
 - [renderer.h](#), 47
- [Dubins](#), 23
 - [distance](#), 24

- Dubins, 23, 24
 - path, 24
 - point, 25
 - time, 25
- dubins.cpp, 70
- dubins.h, 42, 43
- DubinsPath, 25
 - DubinsPath, 26
- empty
 - PriorityQueue< T >, 31
- FileSelector, 26
- fileSelector.cpp, 71, 72
- fileSelector.h, 43, 44
- findPath
 - AStar, 10
 - TimedAStar, 35
- getAStarPath
 - Car, 12
- getAverageSpeed
 - Car, 12
- getBuildings
 - CityMap, 17
- getCars
 - Manager, 28
- getElapsedDistance
 - Car, 12
- getElapsedTime
 - Car, 12
- getEnd
 - Car, 13
- getGraphPoints
 - CityGraph, 15
- getGreenAreas
 - CityMap, 17
- getHeight
 - CityGraph, 15
 - CityMap, 17
- getIntersections
 - CityMap, 17
- getMaxLatLon
 - CityMap, 18
- getMinLatLon
 - CityMap, 18
- getNeighbors
 - CityGraph, 16
- getNumCars
 - Manager, 28
- getPath
 - Car, 13
- getPathLength
 - Car, 13
- getPathTime
 - Car, 13
- getPosition
 - Car, 13
- getRandomPoint
 - CityGraph, 16
- getRemainingDistance
 - Car, 13
- getRemainingTime
 - Car, 14
- getRoads
 - CityMap, 18
- getSpeed
 - Car, 14
- getSpeedAt
 - Car, 14
- getStart
 - Car, 14
- getWaterAreas
 - CityMap, 18
- getWidth
 - CityGraph, 16
 - CityMap, 18
- hasConflict
 - Manager, 29
- hasConstraint
 - ConstraintController, 21
- index.py, 51
- isCityMapLoaded
 - CityMap, 18
- latLonToXY
 - utils.h, 50
- loadFile
 - CityMap, 19
- loadFont
 - utils.cpp, 88
 - utils.h, 50
- main.cpp, 73
- Manager, 26
 - createCarsAStar, 28
 - createCarsCBS, 28
 - createSubCBS, 28
 - getCars, 28
 - getNumCars, 28
 - hasConflict, 29
 - Manager, 27
 - processCBS, 29
 - renderCars, 29
 - toggleCarDebug, 30
- manager.cpp, 74
- manager.h, 44
- managerCBS.cpp, 75
- normalizeAngle
 - utils.h, 50
- path
 - Dubins, 24
- point
 - Dubins, 25
- pop

- PriorityQueue< T >, 31
- PriorityQueue
 - PriorityQueue< T >, 31
- PriorityQueue< T >, 30
 - empty, 31
 - pop, 31
 - PriorityQueue, 31
 - push, 31
- priorityQueue.h, 45
- processCBS
 - Manager, 29
- push
 - PriorityQueue< T >, 31
- render
 - Car, 14
- renderCars
 - Manager, 29
- renderCityGraph
 - Renderer, 32
- renderCityMap
 - Renderer, 32
- Renderer, 32
 - renderCityGraph, 32
 - renderCityMap, 32
 - renderManager, 32
- renderer.cpp, 79, 80
- renderer.h, 46, 47
 - drawArrow, 47
- renderManager
 - Renderer, 32
- Test, 33
- test.cpp, 84
- test.h, 48
- time
 - Dubins, 25
- TimedAStar, 33
 - findPath, 35
 - TimedAStar, 33
- timedAStar.cpp, 85
- toggleCarDebug
 - Manager, 30
- turningRadius
 - utils.h, 50
- turningRadiusToSpeed
 - utils.h, 51
- utils.cpp, 87, 89
 - carConflict, 88
 - carsCollided, 88
 - loadFont, 88
- utils.h, 48, 51
 - carConflict, 49
 - carsCollided, 49
 - distance, 49
 - latLonToXY, 50
 - loadFont, 50
 - normalizeAngle, 50
- turningRadius, 50
- turningRadiusToSpeed, 51