City CBS A∗

Generated by Doxygen 1.9.8

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 std Namespace Reference

Classes

- struct hash< _aStarConflict >
- struct hash< _aStarNode >
- struct hash< _cityGraphNeighbor >
- struct hash< _cityGraphPoint >
- struct hash< _managerOCBSConflict >
- struct hash< _managerOCBSConflictSituation >
- struct hash< std::pair< _cityGraphPoint, _cityGraphNeighbor > >

Chapter 6

Class Documentation

6.1 _aStarConflict Struct Reference

A conflict for the A∗ algorithm.

`#include <aStar.h>`

Public Member Functions

- bool operator== (const _aStarConflict &other) const

Public Attributes

- _cityGraphPoint point
  The point in the graph.
- int time
  The time of the conflict.
- int car
  The car that caused the conflict.

6.1.1 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.

6.1.2 Member Function Documentation

6.1.2.1 operator==()

```
bool _aStarConflict::operator== (
            const _aStarConflict & other ) const  [inline]
```

Definition at line 46 of file aStar.h.

```
00046                                                   {
00047     return point == other.point && time == other.time && car == other.car;
00048   }
```

6.1.3 Member Data Documentation

6.1.3.1 car

```
int _aStarConflict::car
```

The car that caused the conflict.

Definition at line 44 of file aStar.h.

6.1.3.2 point

```
_cityGraphPoint _aStarConflict::point
```

The point in the graph.

Definition at line 42 of file aStar.h.

6.1.3.3 time

```
int _aStarConflict::time
```

The time of the conflict.

Definition at line 43 of file aStar.h.

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

- aStar.h

6.2 _aStarNode Struct Reference

A node for the A∗ algorithm.

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

Public Member Functions

- bool operator== (const _aStarNode &other) const

Public Attributes

- _cityGraphPoint point

  The point in the graph.
- double speed

  The speed of the car.
- std::pair< _cityGraphPoint, _cityGraphNeighbor > arcFrom

  The arc from which the node was reached.

6.2.1 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.

6.2.2 Member Function Documentation

6.2.2.1 operator==()

```
bool _aStarNode::operator== (
            const _aStarNode & other ) const  [inline]
```

Definition at line 25 of file aStar.h.

```
00025                                                {
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   }
```

6.2.3 Member Data Documentation

6.2.3.1 arcFrom

```
std::pair<_cityGraphPoint, _cityGraphNeighbor> _aStarNode::arcFrom
```

The arc from which the node was reached.

Definition at line 23 of file aStar.h.

6.2.3.2 point

```
_cityGraphPoint _aStarNode::point
```

The point in the graph.

Definition at line 21 of file aStar.h.

6.2.3.3 speed

```
double _aStarNode::speed
```

The speed of the car.

Definition at line 22 of file aStar.h.

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

- aStar.h

6.3 _cityGraphNeighbor Struct Reference

A neighbor of a point in the city graph.

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

Public Member Functions

- bool operator== (const _cityGraphNeighbor &other) const

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.
- bool isRightWay
  If it is the right way.

6.3.1 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 44 of file cityGraph.h.

6.3.2 Member Function Documentation

6.3.2.1 operator==()

```
bool _cityGraphNeighbor::operator== (
            const _cityGraphNeighbor & other ) const  [inline]
```

Definition at line 50 of file cityGraph.h.

```
00050                                                       {
00051     return point == other.point && maxSpeed == other.maxSpeed && turningRadius == other.turningRadius
      &&
00052            isRightWay == other.isRightWay;
00053   }
```

6.3.3 Member Data Documentation

6.3.3.1 isRightWay

```
bool _cityGraphNeighbor::isRightWay
```

If it is the right way.

Definition at line 48 of file cityGraph.h.

6.3.3.2 maxSpeed

```
double _cityGraphNeighbor::maxSpeed
```

The maximum speed to reach the neighbor point.

Definition at line 46 of file cityGraph.h.

6.3.3.3 point

```
_cityGraphPoint _cityGraphNeighbor::point
```

The neighbor point.

Definition at line 45 of file cityGraph.h.

6.3.3.4 turningRadius

```
double _cityGraphNeighbor::turningRadius
```

The turning radius to reach the neighbor point.

Definition at line 47 of file cityGraph.h.

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

- cityGraph.h

6.4 _cityGraphPoint Struct Reference

A point in the city graph.

`#include <cityGraph.h>`

Public Member Functions

- bool operator== (const _cityGraphPoint &other) const

Public Attributes

- sf::Vector2f position

  The position of the point.
- sf::Angle angle

  The angle of the point.

6.4.1 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 21 of file cityGraph.h.

6.4.2 Member Function Documentation

6.4.2.1 operator==()

```
bool _cityGraphPoint::operator== (
            const _cityGraphPoint & other ) const  [inline]
```

Definition at line 25 of file cityGraph.h.

```
00025                                                {
00026     int x = std::round(position.x / CELL_SIZE);
00027     int y = std::round(position.y / CELL_SIZE);
00028     int a = std::round(angle.asRadians() / ANGLE_RESOLUTION);
00029     int oX = std::round(other.position.x / CELL_SIZE);
00030     int oY = std::round(other.position.y / CELL_SIZE);
00031     int oA = std::round(other.angle.asRadians() / ANGLE_RESOLUTION);
00032
00033     return x == oX && y == oY && a == oA;
00034   }
```

6.4.3 Member Data Documentation

6.4.3.1 angle

`sf::Angle _cityGraphPoint::angle`

The angle of the point.

Definition at line 23 of file cityGraph.h.

6.4.3.2 position

```
sf::Vector2f _cityGraphPoint::position
```

The position of the point.

Definition at line 22 of file cityGraph.h.

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

- cityGraph.h

6.5 _cityMapBuilding Struct Reference

A building in the city map.

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

Public Attributes

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

6.5.1 Detailed Description

A building in the city map.

Definition at line 44 of file cityMap.h.

6.5.2 Member Data Documentation

6.5.2.1 points

```
std::vector<sf::Vector2f> _cityMapBuilding::points
```

The points of the building.

Definition at line 45 of file cityMap.h.

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

- cityMap.h

6.6 _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.

6.6.1 Detailed Description

A green area in the city map.

Definition at line 52 of file cityMap.h.

6.6.2 Member Data Documentation

6.6.2.1 points

```
std::vector<sf::Vector2f> _cityMapGreenArea::points
```

The points of the green area.

Definition at line 53 of file cityMap.h.

6.6.2.2 type

```
int _cityMapGreenArea::type
```

The type of the green area.

Definition at line 54 of file cityMap.h.

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

- cityMap.h

6.7 _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.

6.7.1 Detailed Description

An intersection in the city map.

Definition at line 69 of file cityMap.h.

6.7.2 Member Data Documentation

6.7.2.1 center

```
sf::Vector2f _cityMapIntersection::center
```

The center of the intersection.

Definition at line 71 of file cityMap.h.

6.7.2.2 id

```
int _cityMapIntersection::id
```

The id of the intersection.

Definition at line 70 of file cityMap.h.

6.7.2.3 radius

```
double _cityMapIntersection::radius
```

The radius of the intersection.

Definition at line 72 of file cityMap.h.

6.7.2.4 roadSegmentIds

```
std::vector<std::pair<int, int> > _cityMapIntersection::roadSegmentIds
```

The ids of the road segments (roadId, segmentId). The segments are the same for both directions of the road.

Definition at line 73 of file cityMap.h.

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

- cityMap.h

6.8 _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.

6.8.1 Detailed Description

A road in the city map.

Definition at line 33 of file cityMap.h.

6.8.2 Member Data Documentation

6.8.2.1 id

```
int _cityMapRoad::id
```

The id of the road.

Definition at line 34 of file cityMap.h.

6.8.2.2 numLanes

```
int _cityMapRoad::numLanes
```

The number of lanes of the road.

Definition at line 37 of file cityMap.h.

6.8.2.3 segments

```
std::vector<_cityMapSegment> _cityMapRoad::segments
```

The segments of the road.

Definition at line 35 of file cityMap.h.

6.8.2.4 width

```
double _cityMapRoad::width
```

The width of the road.

Definition at line 36 of file cityMap.h.

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

- cityMap.h

6.9 _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.
- sf::Angle angle
  The angle of the segment.

6.9.1 Detailed Description

A segment in the city map.

Definition at line 21 of file cityMap.h.

6.9.2 Member Data Documentation

6.9.2.1 angle

```
sf::Angle _cityMapSegment::angle
```

The angle of the segment.

Definition at line 26 of file cityMap.h.

6.9.2.2 p1

`sf::Vector2f _cityMapSegment::p1`

The first point of the segment.

Definition at line 22 of file cityMap.h.

6.9.2.3 p1_offset

`sf::Vector2f _cityMapSegment::p1_offset`

The offset of the first point, used for the intersection.

Definition at line 24 of file cityMap.h.

6.9.2.4 p2

`sf::Vector2f _cityMapSegment::p2`

The second point of the segment.

Definition at line 23 of file cityMap.h.

6.9.2.5 p2_offset

`sf::Vector2f _cityMapSegment::p2_offset`

The offset of the second point, used for the intersection.

Definition at line 25 of file cityMap.h.

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

- cityMap.h

6.10 _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.

6.10.1 Detailed Description

A water area in the city map.

Definition at line 61 of file cityMap.h.

6.10.2 Member Data Documentation

6.10.2.1 points

```
std::vector<sf::Vector2f> _cityMapWaterArea::points
```

The points of the water area.

Definition at line 62 of file cityMap.h.

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

- cityMap.h

6.11 _data Struct Reference

Data structure.

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

Public Attributes

- double numCars
- double carDensity
- std::vector< double > carAvgSpeed

6.11.1 Detailed Description

Data structure.

This struct represents the data structure.

Definition at line 18 of file dataManager.h.

6.11.2 Member Data Documentation

6.11.2.1 carAvgSpeed

```
std::vector<double> _data::carAvgSpeed
```

Definition at line 21 of file dataManager.h.

6.11.2.2 carDensity

```
double _data::carDensity
```

Definition at line 20 of file dataManager.h.

6.11.2.3 numCars

```
double _data::numCars
```

Definition at line 19 of file dataManager.h.

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

- dataManager.h

6.12 _managerOCBSConflict Struct Reference

```
#include <manager_ocbs.h>
```

Public Member Functions

- bool operator== (const _managerOCBSConflict &other) const

Public Attributes

- int car
- int withCar
- double time
- sf::Vector2f position

6.12.1 Detailed Description

Definition at line 28 of file manager_ocbs.h.

6.12.2 Member Function Documentation

6.12.2.1 operator==()

```
bool _managerOCBSConflict::operator== (
            const _managerOCBSConflict & other ) const  [inline]
```

Definition at line 34 of file manager_ocbs.h.

```
00034                                                       {
00035     return car == other.car && withCar == other.withCar && time == other.time;
00036   }
```

6.12.3 Member Data Documentation

6.12.3.1 car

```
int _managerOCBSConflict::car
```

Definition at line 29 of file manager_ocbs.h.

6.12.3.2 position

```
sf::Vector2f _managerOCBSConflict::position
```

Definition at line 32 of file manager_ocbs.h.

6.12.3.3 time

```
double _managerOCBSConflict::time
```

Definition at line 31 of file manager_ocbs.h.

6.12.3.4 withCar

```
int _managerOCBSConflict::withCar
```

Definition at line 30 of file manager_ocbs.h.

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

- manager_ocbs.h

6.13 _managerOCBSConflictSituation Struct Reference

```
#include <manager_ocbs.h>
```

Public Member Functions

- bool operator== (const _managerOCBSConflictSituation &other) const

Public Attributes

- int car
- sf::Vector2f at
- double time

6.13.1 Detailed Description

Definition at line 12 of file manager_ocbs.h.

6.13.2 Member Function Documentation

6.13.2.1 operator==()

```
bool _managerOCBSConflictSituation::operator== (
            const _managerOCBSConflictSituation & other ) const  [inline]
```

Definition at line 17 of file manager_ocbs.h.

```
00017                                                                    {
00018     int t = std::round(time / OCBS_CONFLICT_RANGE);
00019     int oT = std::round(other.time / OCBS_CONFLICT_RANGE);
00020     int x = std::round(at.x / CELL_SIZE);
00021     int oX = std::round(other.at.x / CELL_SIZE);
00022     int y = std::round(at.y / CELL_SIZE);
00023     int oY = std::round(other.at.y / CELL_SIZE);
00024     return car == other.car && t == oT && x == oX && y == oY;
00025   }
```

6.13.3 Member Data Documentation

6.13.3.1 at

```
sf::Vector2f _managerOCBSConflictSituation::at
```

Definition at line 14 of file manager_ocbs.h.

6.13.3.2 car

```
int _managerOCBSConflictSituation::car
```

Definition at line 13 of file manager_ocbs.h.

6.13.3.3 time

```
double _managerOCBSConflictSituation::time
```

Definition at line 15 of file manager_ocbs.h.

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

- manager_ocbs.h

6.14 _managerOCBSNode Struct Reference

```
#include <manager_ocbs.h>
```

Public Member Functions

- bool operator< (const _managerOCBSNode &other) const

Public Attributes

- std::vector< std::vector< sf::Vector2f > > paths
  The paths 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.

6.14.1 Detailed Description

Definition at line 56 of file manager_ocbs.h.

6.14.2 Member Function Documentation

6.14.2.1 operator<()

```
bool _managerOCBSNode::operator< (
            const _managerOCBSNode & other ) const  [inline]
```

Definition at line 65 of file manager_ocbs.h.

```
00065                                                    {
00066     return cost > other.cost || (cost == other.cost && depth > other.depth);
00067   }
```

6.14.3 Member Data Documentation

6.14.3.1 cost

```
double _managerOCBSNode::cost
```

The total cost.

Definition at line 59 of file manager_ocbs.h.

6.14.3.2 costs

```
std::vector<double> _managerOCBSNode::costs
```

The individual path costs.

Definition at line 58 of file manager_ocbs.h.

6.14.3.3 depth

```
int _managerOCBSNode::depth
```

The depth in the CBS tree.

Definition at line 60 of file manager_ocbs.h.

6.14.3.4 hasResolved

```
bool _managerOCBSNode::hasResolved
```

If the node has resolved conflicts.

Definition at line 61 of file manager_ocbs.h.

6.14.3.5 paths

```
std::vector<std::vector<sf::Vector2f> > _managerOCBSNode::paths
```

The paths for all agents.

Definition at line 57 of file manager_ocbs.h.

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

- manager_ocbs.h

6.15 AStar Class Reference

A∗ algorithm.

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

Public Types

- using node = _aStarNode
- using conflict = _aStarConflict

Public Member Functions

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

6.15.1 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.

6.15.2 Member Typedef Documentation

6.15.2.1 conflict

```
using AStar::conflict = _aStarConflict
```

Definition at line 77 of file aStar.h.

6.15.2.2 node

```
using AStar::node = _aStarNode
```

Definition at line 76 of file aStar.h.

6.15.3 Constructor & Destructor Documentation

6.15.3.1 AStar()

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

Constructor.

Parameters

start	The start point
end	The end point
cityGraph	The graph

Definition at line 16 of file aStar.cpp.

```
00016                                                                    {
00017     this->start.point = start;
00018     this->start.speed = 0;
00019     this->end.point = end;
00020     this->end.speed = 0;
00021     this->graph = cityGraph;
00022 }
```

6.15.4 Member Function Documentation

6.15.4.1 findPath()

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

Find the path.

Returns

The path

Definition at line 91 of file aStar.h.

```
00091                                {
00092     if (!processed)
00093       process();
00094     return path;
00095   }
```

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

- aStar.h
- aStar.cpp

6.16 Car Class Reference

A car in the city.

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

Public Member Functions

- Car ()
  Constructor.
- void assignStartEnd (_cityGraphPoint start, _cityGraphPoint 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, CityGraph &graph)
  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.
- _cityGraphPoint getStart ()
  Get the start point.
- _cityGraphPoint 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.

6.16.1 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 22 of file car.h.

6.16.2 Constructor & Destructor Documentation

6.16.2.1 Car()

```
Car::Car ( )
```

Constructor.

Definition at line 14 of file car.cpp.

```
00014          {
00015   std::vector<sf::Color> colors = {sf::Color(50, 120, 190), sf::Color(183, 132, 144), sf::Color(105,
      101, 89),
00016                                    sf::Color(182, 18, 34),  sf::Color(24, 25, 24),    sf::Color(17,
      86, 122)};
00017   color = colors[rand() % colors.size()];
00018 }
```

6.16.3 Member Function Documentation

6.16.3.1 assignExistingPath()

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

Assign an existing path to the car.

Parameters

path	The path

Definition at line 116 of file car.cpp.

```
00116                                                   {
00117   this->path = path;
00118   currentPoint = 0;
00119 }
```

6.16.3.2 assignPath()

```
void Car::assignPath (
            std::vector< AStar::node > path,
            CityGraph & graph )
```

Assign a path to the car.

Parameters

path	The path

Definition at line 85 of file car.cpp.

```
00085                                                       {
00086   this->path.clear();
00087   this->aStarPath = path;
00088   currentPoint = 0;
00089
00090   double index = 0;
00091   double t = 0;
00092   double prevTime = 0;
00093
00094   for (int i = 1; i < (int)path.size(); i++) {
00095     AStar::node prevNode = path[i - 1];
00096     AStar::node node = path[i];
00097
00098     CityGraph::point start = node.arcFrom.first;
00099     CityGraph::neighbor end = node.arcFrom.second;
00100
00101     DubinsInterpolator *interpolator = graph.getInterpolator(start, end);
00102
00103     double duration = interpolator->getDuration(prevNode.speed, node.speed);
00104
00105     while (t < prevTime + duration) {
00106       double tt = t - prevTime;
00107       CityGraph::point p = interpolator->get(tt, prevNode.speed, node.speed);
00108
00109       this->path.push_back(p.position);
00110       t += SIM_STEP_TIME;
00111     }
00112     prevTime += duration;
00113   }
00114 }
```

6.16.3.3 assignStartEnd()

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

Assign the start and end points.

Parameters

start	The start point
end	The end point

Definition at line 34 of file car.h.

```
00034                                                                    {
00035     this->start = start;
00036     this->end = end;
00037   }
```

6.16.3.4 chooseRandomStartEndPath()

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

Choose a random start and end point in the graph.

Parameters

graph	The graph
cityMap	The city map

Definition at line 171 of file car.cpp.

```
00171                                                                      {
00172   CityGraph::point start;
00173   CityGraph::point end;
00174
00175   double minDistance = std::max(graph.getWidth(), graph.getHeight()) / 2.0;
00176   std::vector<AStar::node> path;
00177
00178   do {
00179     path.clear();
00180     start = graph.getRandomPoint();
00181     end = graph.getRandomPoint();
00182
00183     if (std::sqrt(std::pow(start.position.x - end.position.x, 2) + std::pow(start.position.y -
      end.position.y, 2)) <
00184         minDistance)
00185       continue;
00186
00187     AStar aStar(start, end, graph);
00188     path = aStar.findPath();
00189
00190     if (!path.empty() && (int)path.size() >= 3) {
00191       AStar aStar(start, end, graph);
00192       path.clear();
00193       path = aStar.findPath();
00194     }
00195   } while (path.empty() || (int)path.size() < 3);
00196
00197   this->assignStartEnd(start, end);
00198   this->assignPath(path, graph);
00199 }
```

6.16.3.5 getAStarPath()

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

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

Returns

The path

Definition at line 153 of file car.h.

```
00153 { return aStarPath; }
```

6.16.3.6 getAverageSpeed()

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

Get the average speed of the car.

Parameters

graph	The graph

Returns

The average speed

Definition at line 201 of file car.cpp.

```
00201                                           {
00202   double dist = 0;
00203   double time = 0;
00204   auto outOfBounds = [&](sf::Vector2f p) {
00205     return p.x < 0 || p.y < 0 || p.x > graph.getWidth() || p.y > graph.getWidth();
00206   };
00207
00208   for (int i = 0; i < (int)path.size() - 1; i++) {
00209     if (outOfBounds(path[i]) || outOfBounds(path[i + 1]))
00210       continue;
00211
00212     sf::Vector2f diff = path[i + 1] - path[i];
00213     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00214     time += SIM_STEP_TIME;
00215   }
00216
00217   if (time == 0)
00218     return 0;
00219
00220   return dist / time;
00221 }
```

6.16.3.7 getElapsedDistance()

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

Get the elapsed distance since the start of the car.

Returns

The elapsed distance

Definition at line 151 of file car.cpp.

```
00151                                 {
00152   double dist = 0;
00153   for (int i = 0; i < currentPoint; i++) {
00154     sf::Vector2f diff = path[i + 1] - path[i];
00155     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00156   }
00157
00158   return dist;
00159 }
```

6.16.3.8 getElapsedTime()

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

Get the elapsed time since the start of the car.

Returns

The elapsed time

Definition at line 138 of file car.cpp.

```
00138 { return (double)currentPoint * SIM_STEP_TIME; }
```

6.16.3.9 getEnd()

```
_cityGraphPoint Car::getEnd ( )  [inline]
```

Get the end point.

Returns

The end point

Definition at line 79 of file car.h.

```
00079 { return end; }
```

6.16.3.10 getPath()

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

Get the path of the car.

Returns

The path

Definition at line 147 of file car.h.

```
00147 { return path; }
```

6.16.3.11 getPathLength()

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

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

Returns

The distance

Definition at line 161 of file car.cpp.

```
00161                           {
00162   double dist = 0;
00163   for (int i = 0; i < (int)path.size() - 1; i++) {
00164     sf::Vector2f diff = path[i + 1] - path[i];
00165     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00166   }
00167
00168   return dist;
00169 }
```

6.16.3.12 getPathTime()

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

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

Returns

The time

Definition at line 139 of file car.cpp.

```
00139 { return (double)path.size() * SIM_STEP_TIME; }
```

6.16.3.13 getPosition()

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

Get the position of the car.

Returns

The position

Definition at line 141 of file car.h.

```
00141 { return path[currentPoint]; }
```

6.16.3.14 getRemainingDistance()

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

Get the remaining distance to reach the end point.

Returns

The remaining distance

Definition at line 141 of file car.cpp.

```
00141                                   {
00142   double dist = 0;
00143   for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00144     sf::Vector2f diff = path[i + 1] - path[i];
00145     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00146   }
00147
00148   return dist;
00149 }
```

6.16.3.15 getRemainingTime()

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

Get the remaining time to reach the end point.

Returns

The remaining time

Definition at line 137 of file car.cpp.

```
00137 { return (double)(path.size() - currentPoint) * SIM_STEP_TIME; }
```

6.16.3.16 getSpeed()

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

Get the current point in the path.

Returns

The current point in the path

Definition at line 121 of file car.cpp.

```
00121                       {
00122   if (currentPoint >= (int)path.size() - 1)
00123     return 0;
00124
00125   sf::Vector2f diff = path[currentPoint + 1] - path[currentPoint];
00126   return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00127 }
```

6.16.3.17 getSpeedAt()

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

Get the speed at a certain index in the path.

Parameters

index	The index

Returns

The speed at the index

Definition at line 129 of file car.cpp.

```
00129                                   {
00130   if (index >= (int)path.size() - 1)
00131     return 0;
00132
00133   sf::Vector2f diff = path[index + 1] - path[index];
00134   return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00135 }
```

6.16.3.18 getStart()

```
_cityGraphPoint Car::getStart ( )  [inline]
```

Get the start point.

Returns

The start point

Definition at line 73 of file car.h.

```
00073 { return start; }
```

6.16.3.19 move()

```
void Car::move ( )
```

Move the car, move to the next point in the path.

Definition at line 20 of file car.cpp.

```
00020                {
00021   if (currentPoint >= (int)path.size())
00022     return;
00023
00024   currentPoint++;
00025 }
```

6.16.3.20 render()

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

Render the car.

Parameters

window	The window

Definition at line 27 of file car.cpp.

```
00027                                       {
00028   if (1 + currentPoint >= (int)path.size())
00029     return;
00030
00031   sf::Vector2f point = path[currentPoint];
00032   sf::Vector2f nextPoint = path[currentPoint + 1];
00033   sf::Vector2f diff = nextPoint - point;
00034   double length = sqrt(diff.x * diff.x + diff.y * diff.y);
00035   int fact = 1;
00036
00037   while (point == nextPoint && currentPoint + fact < (int)path.size()) {
00038     fact++;
00039     nextPoint = path[currentPoint + fact];
00040     diff = nextPoint - point;
00041     length = sqrt(diff.x * diff.x + diff.y * diff.y);
00042   }
00043
00044   sf::RectangleShape shape(sf::Vector2f(CAR_LENGTH, CAR_WIDTH));
00045   shape.setOrigin({CAR_LENGTH / 2.0f, CAR_WIDTH / 2.0f});
00046   shape.setPosition(point);
00047   shape.setRotation(sf::radians(atan2(nextPoint.y - point.y, nextPoint.x - point.x)));
00048   if (debug)
00049     shape.setFillColor(sf::Color(255, 0, 0));
00050   else
00051     shape.setFillColor(color);
00052   window.draw(shape);
00053
00054   if (!debug)
00055     return;
00056
00057   // Render speed, elapsed time, remaining time, and distance
00058   double speed = 3.6f * length / (fact * SIM_STEP_TIME);
00059   int iSpeed = speed;
00060   int dSpeed = (double)(speed - iSpeed) * 100.0;
00061   sf::Font font = loadFont();
00062   sf::Text text(font);
00063   text.setCharacterSize(24);
00064   text.setFillColor(sf::Color::White);
00065   text.setPosition(getPosition());
00066   text.setString(std::to_string(iSpeed) + "." + std::to_string(dSpeed) + " km/h" + "\n" +
00067                  std::to_string((int)getElapsedTime()) + "s / " +
      std::to_string((int)getRemainingTime()) + "s" + "\n" +
00068                  std::to_string((int)getElapsedDistance()) + "m / " +
      std::to_string((int)getRemainingDistance()) +
00069                  "m");
00070   text.setOutlineColor(sf::Color::Black);
```

```
00071   text.setOutlineThickness(1.0f);
00072   text.scale({0.1f, 0.1f});
00073   text.setOrigin({text.getLocalBounds().position.x / 2.0f, text.getLocalBounds().position.y / 2.0f});
00074   window.draw(text);
00075
00076   // Render path
00077   for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00078     sf::Vertex line[] = {{path[i]}, {path[i + 1]}};
00079     line[0].color = sf::Color(255, 255, 255);
00080     line[1].color = sf::Color(255, 255, 255);
00081     window.draw(line, 2, sf::PrimitiveType::Lines);
00082   }
00083 }
```

6.16.3.21 toggleDebug()

```
void Car::toggleDebug ( )  [inline]
```

Toggle the debug mode. In debug mode, the path of the car is rendered and the car is rendered in red.

Definition at line 159 of file car.h.

```
00159 { debug = !debug; }
```

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

- car.h
- car.cpp

6.17 CityGraph Class Reference

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

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

Public Types

- using point = _cityGraphPoint
- using neighbor = _cityGraphNeighbor

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.
- DubinsInterpolator * getInterpolator (const point &point1, const neighbor &point2)

  Get the interpolator for a Dubins path between two points.

6.17.1 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 85 of file cityGraph.h.

6.17.2 Member Typedef Documentation

6.17.2.1 neighbor

```
using CityGraph::neighbor = _cityGraphNeighbor
```

Definition at line 88 of file cityGraph.h.

6.17.2.2 point

```
using CityGraph::point = _cityGraphPoint
```

Definition at line 87 of file cityGraph.h.

6.17.3 Member Function Documentation

6.17.3.1 createGraph()

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

Create a city graph.

This constructor creates a city graph from a city map.

Parameters

cityMap	The city map

Definition at line 20 of file cityGraph.cpp.

```
00020                                                   {
00021   auto roads = cityMap.getRoads();
00022   auto intersections = cityMap.getIntersections();
00023
00024   this->height = cityMap.getHeight();
00025   this->width = cityMap.getWidth();
00026
00027   // Graph's points are evenly distributed along a road segment
00028   for (const auto &road : roads) {
00029     if (road.segments.empty()) {
00030       continue;
00031     }
00032
00033     int numSeg = 0;
00034     for (const auto &segment : road.segments) {
00035       if (numSeg > 0) { // Link to the previous one
```

```
00036         for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
00037           double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width /
      road.numLanes;
00038           offset += road.width / (2 * road.numLanes);
00039
00040           point point1;
00041           point1.angle = road.segments[numSeg - 1].angle;
00042           point1.position = sf::Vector2f(
00043               road.segments[numSeg - 1].p2_offset.x + offset * sin(road.segments[numSeg -
      1].angle.asRadians()),
00044               road.segments[numSeg - 1].p2_offset.y + offset * -cos(road.segments[numSeg -
      1].angle.asRadians()));
00045
00046           point point2;
00047           point2.angle = road.segments[numSeg].angle;
00048           point2.position =
00049               sf::Vector2f(road.segments[numSeg].p1_offset.x + offset *
      sin(road.segments[numSeg].angle.asRadians()),
00050                            road.segments[numSeg].p1_offset.y + offset *
      -cos(road.segments[numSeg].angle.asRadians()));
00051
00052           linkPoints(point1, point2, 2, true);
00053         }
00054       }
00055       numSeg++;
00056
00057       double segmentLength =
00058           sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y -
      segment.p1_offset.y, 2));
00059       double pointDistance = 15;
00060       int numPoints = segmentLength / pointDistance;
00061       double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
00062       double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
00063       double dx_a = sin(segment.angle.asRadians());
00064       double dy_a = -cos(segment.angle.asRadians());
00065
00066       if (dx_a < 0) {
00067         dx_a = -dx_a;
00068         dy_a = -dy_a;
00069       }
00070
00071       for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
00072         double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
00073         offset += road.width / (2 * road.numLanes);
00074
00075         if (numPoints == 0) {
00076           point point1;
00077           point1.angle = segment.angle;
00078           point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y +
      offset * dy_a);
00079
00080           point point2;
00081           point2.angle = segment.angle;
00082           point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y +
      offset * dy_a);
00083
00084           linkPoints(point1, point2, 2, true);
00085           continue;
00086         }
00087
00088         for (int i = 0; i <= numPoints; i++) {
00089           point point1;
00090           point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
00091                                          segment.p1_offset.y + i * dy_s + offset * dy_a);
00092           point1.angle = segment.angle;
00093
00094           if (i > 0) {
00095             for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
00096               double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width /
      road.numLanes;
00097               offset2 += road.width / (2 * road.numLanes);
00098
00099               point point2;
00100               point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
00101                                              segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
00102               point2.angle = segment.angle;
00103
00104               int direction = 2;
00105               double a = atan2(dy_a, dx_a);
00106               if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
00107                 if (dy_s >= 0) {
00108                   direction = offset > 0 ? 0 : 1;
00109                 } else {
00110                   direction = offset > 0 ? 1 : 0;
00111                 }
00112                 linkPoints(point1, point2, direction, offset == offset2);
00113               } else {
```

```
00114                 if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
00115                   linkPoints(point1, point2, 2, true);
00116                 }
00117               }
00118             }
00119           }
00120         }
00121       }
00122     }
00123   }
00124
00125   // Connect the intersections
00126   for (const auto &intersection : intersections) {
00127     for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
00128       for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
00129         const auto &road1 = roads[roadSegmentId1.first];
00130         const auto &road2 = roads[roadSegmentId2.first];
00131         const auto &segment1 = road1.segments[roadSegmentId1.second];
00132         const auto &segment2 = road2.segments[roadSegmentId2.second];
00133
00134         // Find the point of the segment2 closest to the intersection
00135         point point1;
00136         point1.angle = segment1.angle;
00137         point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2,
      intersection.center))
00138                               ? segment1.p1_offset
00139                               : segment1.p2_offset;
00140
00141         point point2;
00142         point2.angle = segment2.angle;
00143         point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2,
      intersection.center))
00144                               ? segment2.p1_offset
00145                               : segment2.p2_offset;
00146
00147         for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
00148           double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width /
      road1.numLanes;
00149           offset1 += road1.width / (2 * road1.numLanes);
00150
00151           for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
00152             double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width /
      road2.numLanes;
00153             offset2 += road2.width / (2 * road2.numLanes);
00154
00155             point point1_offset;
00156             point1_offset.angle = segment1.angle;
00157             point1_offset.position = sf::Vector2f(point1.position.x + offset1 *
      sin(segment1.angle.asRadians()),
00158                                                   point1.position.y + offset1 *
      -cos(segment1.angle.asRadians()));
00159
00160             point point2_offset;
00161             point2_offset.angle = segment2.angle;
00162             point2_offset.position = sf::Vector2f(point2.position.x + offset2 *
      sin(segment2.angle.asRadians()),
00163                                                   point2.position.y + offset2 *
      -cos(segment2.angle.asRadians()));
00164
00165             linkPoints(point1_offset, point2_offset, 2, true);
00166           }
00167         }
00168       }
00169     }
00170   }
00171
00172   spdlog::info("Graph created with {} points", graphPoints.size());
00173
00174   // Remove all the neighbors that need to turn too much
00175   for (auto &point : graphPoints) {
00176     std::vector<neighbor> newNeighbors;
00177     double distance;
00178     for (auto &neighbor : neighbors[point]) {
00179       double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
00180       bool can = canLink(point, neighbor.point, speed, &distance);
00181
00182       if (!can)
00183         continue;
00184
00185       while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
00186         speed += 0.1;
00187         if (speed >= CAR_MAX_SPEED_MS) {
00188           speed = CAR_MAX_SPEED_MS;
00189           break;
00190         }
00191       }
00192
```

```
00193       if (can) {
00194         neighbor.maxSpeed = speed - 0.1;
00195         neighbor.turningRadius = turningRadius(speed);
00196         newNeighbors.push_back(neighbor);
00197       }
00198     }
00199
00200     neighbors[point].clear();
00201     for (const auto &neighbor : newNeighbors) {
00202       neighbors[point].push_back(neighbor);
00203     }
00204   }
00205
00206   // Interpolate all the curves
00207   spdlog::info("Interpolating curves ...");
00208
00209   interpolators.clear();
00210
00211   for (auto &point : graphPoints) {
00212     for (const auto &neighbor : neighbors[point]) {
00213       std::pair<_cityGraphPoint, _cityGraphNeighbor> key = {point, neighbor};
00214       if (interpolators.find(key) == interpolators.end()) {
00215         interpolators[key] = new DubinsInterpolator();
00216         interpolators[key]->init(point, neighbor.point, neighbor.turningRadius);
00217       }
00218     }
00219   }
00220
00221   spdlog::info("Curves interpolated");
00222 }
```

6.17.3.2 getGraphPoints()

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

Get graph points.

Returns

Graph points

Definition at line 109 of file cityGraph.h.

```
00109 { return graphPoints; }
```

6.17.3.3 getHeight()

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

Get the height of the city graph.

Returns

The height of the city graph

Definition at line 121 of file cityGraph.h.

```
00121 { return height; }
```

6.17.3.4 getInterpolator()

```
DubinsInterpolator * CityGraph::getInterpolator (
            const point & point1,
            const neighbor & point2 )  [inline]
```

Get the interpolator for a Dubins path between two points.

Parameters

point1	The first point
point2	The second point

Returns

The DubinsInterpolator for the path between the two points

Definition at line 135 of file cityGraph.h.

```
00135                                                                                                {
00136     std::pair<point, neighbor> key = {point1, point2};
00137     if (interpolators.find(key) != interpolators.end()) {
00138       return interpolators[key];
00139     }
00140     return nullptr;
00141   }
```

6.17.3.5 getNeighbors()

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

Get neighbors map.

Returns

Neighbors map

Definition at line 103 of file cityGraph.h.

```
00103 { return neighbors; }
```

6.17.3.6 getRandomPoint()

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

Get random point.

Returns

Random point

Definition at line 285 of file cityGraph.cpp.

```
00285                                          {
00286   std::vector<point> graphPointsOut;
00287   for (const auto &point : graphPoints) {
00288     if (point.position.x + CAR_LENGTH < 0 || point.position.x - CAR_LENGTH > width ||
00289         point.position.y + CAR_LENGTH < 0 || point.position.y - CAR_LENGTH > height)
00290       graphPointsOut.push_back(point);
00291   }
00292
00293   auto it = graphPointsOut.begin();
00294   std::random_device rd;
00295   std::mt19937 gen(rd());
00296   std::uniform_int_distribution<> dis(0, graphPointsOut.size() - 1);
00297
00298   std::advance(it, dis(gen));
00299
00300   return *it;
00301 }
```

6.17.3.7 getWidth()

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

Get the width of the city graph.

Returns

The width of the city graph

Definition at line 127 of file cityGraph.h.

```
00127 { return width; }
```

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

- cityGraph.h
- cityGraph.cpp

6.18 CityMap Class Reference

A city map.

`#include <cityMap.h>`

Public Types

- using segment = _cityMapSegment
- using road = _cityMapRoad
- using building = _cityMapBuilding
- using greenArea = _cityMapGreenArea
- using waterArea = _cityMapWaterArea
- using intersection = _cityMapIntersection

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.

6.18.1 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 84 of file cityMap.h.

6.18.2 Member Typedef Documentation

6.18.2.1 building

```
using CityMap::building = _cityMapBuilding
```

Definition at line 88 of file cityMap.h.

6.18.2.2 greenArea

```
using CityMap::greenArea = _cityMapGreenArea
```

Definition at line 89 of file cityMap.h.

6.18.2.3 intersection

```
using CityMap::intersection = _cityMapIntersection
```

Definition at line 91 of file cityMap.h.

6.18.2.4 road

```
using CityMap::road = _cityMapRoad
```

Definition at line 87 of file cityMap.h.

6.18.2.5 segment

```
using CityMap::segment = _cityMapSegment
```

Definition at line 86 of file cityMap.h.

6.18.2.6 waterArea

```
using CityMap::waterArea = _cityMapWaterArea
```

Definition at line 90 of file cityMap.h.

6.18.3 Constructor & Destructor Documentation

6.18.3.1 CityMap()

```
CityMap::CityMap ( )
```

Constructor.

Definition at line 12 of file cityMap.cpp.

```
00012                 {
00013   roads.clear();
00014   intersections.clear();
00015   minLatLon.x = minLatLon.y = maxLatLon.x = maxLatLon.y = 0;
00016 }
```

6.18.4 Member Function Documentation

6.18.4.1 getBuildings()

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

Get the buildings.

Returns

The buildings

Definition at line 126 of file cityMap.h.

```
00126 { return buildings; }
```

6.18.4.2 getGreenAreas()

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

Get the green areas.

Returns

The green areas

Definition at line 132 of file cityMap.h.

```
00132 { return greenAreas; }
```

6.18.4.3 getHeight()

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

Get the height of the city map.

Returns

The height of the city map

Definition at line 162 of file cityMap.h.

```
00162 { return height; }
```

6.18.4.4 getIntersections()

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

Get the intersections.

Returns

The intersections

Definition at line 120 of file cityMap.h.

```
00120 { return intersections; }
```

6.18.4.5 getMaxLatLon()

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

Get the maximum latitude and longitude.

Returns

The maximum latitude and longitude

Definition at line 150 of file cityMap.h.

```
00150 { return maxLatLon; }
```

6.18.4.6 getMinLatLon()

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

Get the minimum latitude and longitude.

Returns

The minimum latitude and longitude

Definition at line 144 of file cityMap.h.

```
00144 { return minLatLon; }
```

6.18.4.7 getRoads()

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

Get the roads.

Returns

The roads

Definition at line 114 of file cityMap.h.

```
00114 { return roads; }
```

6.18.4.8 getWaterAreas()

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

Get the water areas.

Returns

The water areas

Definition at line 138 of file cityMap.h.

```
00138 { return waterAreas; }
```

6.18.4.9 getWidth()

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

Get the width of the city map.

Returns

The width of the city map

Definition at line 156 of file cityMap.h.

```
00156 { return width; }
```

6.18.4.10 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 108 of file cityMap.h.

```
00108 { return isLoaded; }
```

6.18.4.11 loadFile()

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

Load a city map from a file.

Parameters

filename	The filename

Definition at line 18 of file cityMap.cpp.

```
00018                                                {
00019   spdlog::info("Loading file: {}", filename);
00020
00021   tinyxml2::XMLDocument doc;
00022   // Load the XML file
00023   if (doc.LoadFile(filename.c_str()) != tinyxml2::XML_SUCCESS) {
00024     spdlog::error("Failed to load file: {}", filename);
00025     return;
00026   }
00027
00028   // Extract the bounds of the map
00029   tinyxml2::XMLElement *bounds = doc.FirstChildElement("osm")->FirstChildElement("bounds");
00030   if (!bounds) {
00031     spdlog::error("Failed to extract bounds from file: {}", filename);
00032     return;
00033   }
00034
00035   minLatLon.x = bounds->FloatAttribute("minlon");
00036   minLatLon.y = bounds->FloatAttribute("minlat");
00037   maxLatLon.x = bounds->FloatAttribute("maxlon");
00038   maxLatLon.y = bounds->FloatAttribute("maxlat");
00039
00040   // Define the width and height of the map
00041   width = latLonToXY(minLatLon.y, minLatLon.x).x - latLonToXY(maxLatLon.y, maxLatLon.x).x;
00042   height = latLonToXY(minLatLon.y, minLatLon.x).y - latLonToXY(maxLatLon.y, maxLatLon.x).y;
00043   width = std::abs(width);
00044   height = std::abs(height);
00045
00046   std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
00047   spdlog::info("Loading roads and buildings ...");
00048
00049   // List of highway types to exclude
00050   std::set<std::string> excludedHighways = {"footway", "path",  "pedestrian", "cycleway",
00051                                             "steps",   "track", "bridleway",  "service"};
00052
00053   // List of highway types to include
00054   std::set<std::string> includedHighways = {
00055       "motorway",      "trunk",         "primary",    "secondary",    "tertiary",
      "unclassified", "residential",
00056       "living_street", "motorway_link", "trunk_link", "primary_link", "secondary_link",
      "tertiary_link"};
00057
00058   // Extract the roads
00059   tinyxml2::XMLElement *way = doc.FirstChildElement("osm")->FirstChildElement("way");
00060   int roadId = 0;
00061   while (way) {
00062     road r;
00063     building b;
00064     greenArea g;
00065     waterArea w;
00066     r.width = DEFAULT_ROAD_WIDTH;
00067     r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00068     r.id = roadId;
00069
00070     tinyxml2::XMLElement *nd = way->FirstChildElement("nd");
00071     while (nd) {
00072       tinyxml2::XMLElement *node = doc.FirstChildElement("osm")->FirstChildElement("node");
00073       while (node) {
00074         if (node->IntAttribute("id") == nd->IntAttribute("ref")) {
00075           sf::Vector2f p;
00076           p.x = node->FloatAttribute("lon");
00077           p.y = node->FloatAttribute("lat");
00078
00079           if (r.segments.size() > 0) {
00080             segment s;
00081             s.p1 = r.segments.back().p2;
00082             s.p2 = p;
00083             r.segments.push_back(s);
00084           } else {
00085             segment s;
00086             s.p1 = p;
00087             s.p2 = p;
00088             r.segments.push_back(s);
00089           }
00090
00091           b.points.push_back(p);
00092           g.points.push_back(p);
00093           w.points.push_back(p);
00094           break;
00095         }
00096         node = node->NextSiblingElement("node");
00097       }
00098       nd = nd->NextSiblingElement("nd");
00099     }
00100
00101     // Remove the first segment (it has the same p1 and p2)
```

```
00102     r.segments.erase(r.segments.begin());
00103
00104     std::string highwayType;
00105     bool isHighway = false;
00106     bool isBuilding = false;
00107     bool isUnderground = false;
00108     bool isGreenArea = false;
00109     bool isWaterArea = false;
00110     bool widthSet = false;
00111     bool lanesSet = false;
00112     tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
00113     while (tag) {
00114       if (strcmp(tag->Attribute("k"), "width") == 0) {
00115         r.width = tag->FloatAttribute("v");
00116         widthSet = true;
00117       } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
00118         r.numLanes = tag->IntAttribute("v");
00119         lanesSet = true;
00120       } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
00121         highwayType = tag->Attribute("v");
00122         isHighway = true;
00123       } else if (strcmp(tag->Attribute("k"), "building") == 0) {
00124         isBuilding = true;
00125       } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
00126         int layerValue = tag->IntAttribute("v");
00127         if (layerValue < 0) {
00128           isUnderground = true;
00129         }
00130       } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
00131         if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
00132             strcmp(tag->Attribute("v"), "meadow") == 0) {
00133           isGreenArea = true;
00134           g.type = 0;
00135         }
00136       } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
00137         if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
00138           isGreenArea = true;
00139           g.type = 1;
00140         }
00141       } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
00142                  (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream")
    == 0 ||
00143                   strcmp(tag->Attribute("v"), "canal") == 0)) {
00144         isWaterArea = true;
00145       } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
00146                  (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland")
    == 0)) {
00147         isWaterArea = true;
00148       } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
00149                  (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0
    ||
00150                   strcmp(tag->Attribute("v"), "river") == 0)) {
00151         isWaterArea = true;
00152       }
00153       tag = tag->NextSiblingElement("tag");
00154     }
00155     if (!widthSet && !lanesSet) {
00156       r.width = DEFAULT_ROAD_WIDTH;
00157       r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00158     } else if (!widthSet) {
00159       r.width = r.numLanes * DEFAULT_LANE_WIDTH;
00160     } else if (!lanesSet) {
00161       r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00162     }
00163     r.width = std::max(r.width, MIN_ROAD_WIDTH);
00164     r.numLanes = std::max(r.numLanes, 1);
00165
00166     if (isUnderground) {
00167       way = way->NextSiblingElement("way");
00168       continue;
00169     }
00170     if (isBuilding) {
00171       buildings.push_back(b);
00172       way = way->NextSiblingElement("way");
00173       continue;
00174     }
00175     if (isGreenArea) {
00176       greenAreas.push_back(g);
00177       way = way->NextSiblingElement("way");
00178       continue;
00179     }
00180     if (isWaterArea) {
00181       waterAreas.push_back(w);
00182       way = way->NextSiblingElement("way");
00183       continue;
00184     }
00185     if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
```

```
00186       way = way->NextSiblingElement("way");
00187       continue;
00188     }
00189     if (includedHighways.find(highwayType) != includedHighways.end()) {
00190       roads.push_back(r);
00191       roadId++;
00192     }
00193
00194     way = way->NextSiblingElement("way");
00195   }
00196
00197   // Convert lat/lon to meters (using the upper-left corner as origin)
00198   sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
00199   sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
00200   for (auto &r : roads) {
00201     for (auto &s : r.segments) {
00202       s.p1 = latLonToXY(s.p1.y, s.p1.x);
00203       s.p2 = latLonToXY(s.p2.y, s.p2.x);
00204
00205       s.p1.x -= minXY.x;
00206       s.p1.y -= minXY.y;
00207       s.p2.x -= minXY.x;
00208       s.p2.y -= minXY.y;
00209
00210       // Symetri to the x-axis
00211       s.p1.y = maxXY.y - minXY.y - s.p1.y;
00212       s.p2.y = maxXY.y - minXY.y - s.p2.y;
00213
00214       s.p1_offset = s.p1;
00215       s.p2_offset = s.p2;
00216
00217       s.angle = sf::radians(std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x));
00218     }
00219   }
00220   for (auto &b : buildings) {
00221     for (auto &p : b.points) {
00222       p = latLonToXY(p.y, p.x);
00223
00224       p.x -= minXY.x;
00225       p.y -= minXY.y;
00226
00227       // Symetri to the x-axis
00228       p.y = maxXY.y - minXY.y - p.y;
00229     }
00230   }
00231   for (auto &g : greenAreas) {
00232     for (auto &p : g.points) {
00233       p = latLonToXY(p.y, p.x);
00234
00235       p.x -= minXY.x;
00236       p.y -= minXY.y;
00237
00238       // Symetri to the x-axis
00239       p.y = maxXY.y - minXY.y - p.y;
00240     }
00241   }
00242   for (auto &w : waterAreas) {
00243     for (auto &p : w.points) {
00244       p = latLonToXY(p.y, p.x);
00245
00246       p.x -= minXY.x;
00247       p.y -= minXY.y;
00248
00249       // Symetri to the x-axis
00250       p.y = maxXY.y - minXY.y - p.y;
00251     }
00252   }
00253
00254   std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
00255   spdlog::info("Roads and buildings loaded ({} ms)",
00256                std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());
00257
00258   spdlog::info("Loading intersections ...");
00259
00260   // Intersections are at any roads' points if they are near another one
00261   // First add the intersections for each node point
00262   // Then merge the intersections that are close to each other
00263   intersections.clear();
00264   int intersectionId = 0;
00265
00266   // Add the intersections for each road segment
00267   spdlog::debug("Adding intersections ...");
00268   for (auto r : roads) {
00269     for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
00270       segment s = r.segments[s_id];
00271       std::vector<sf::Vector2f> points = {s.p1, s.p2};
00272       for (auto p : points) {
```

```
00273         intersection i = {intersectionId++, p, r.width / 2, {}};
00274         i.roadSegmentIds.push_back({r.id, s_id});
00275         intersections.push_back(i);
00276       }
00277     }
00278   }
00279   spdlog::debug("Intersections added");
00280
00281   // Merge the intersections that are close to each other
00282   spdlog::debug("Merging intersections ...");
00283   for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
00284     for (int i = 0; i < (int)intersections.size(); i++) {
00285       for (int j = i + 1; j < (int)intersections.size(); j++) {
00286         bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();
00287
00288         if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
00289           is_i = intersections[i].id < intersections[j].id;
00290         }
00291
00292         double minSpace = intersections[i].radius + intersections[j].radius;
00293         minSpace /= distCoef;
00294
00295         if (distance(intersections[i].center, intersections[j].center) < minSpace) {
00296           // Merge the intersections to i or j (depending on is_i)
00297           int index_from = is_i ? j : i;
00298           int index_to = is_i ? i : j;
00299
00300           for (auto &r : intersections[index_from].roadSegmentIds) {
00301             intersections[index_to].roadSegmentIds.push_back(r);
00302           }
00303
00304           intersections.erase(intersections.begin() + index_from);
00305           i -= 1;
00306           break;
00307         }
00308       }
00309     }
00310   }
00311   spdlog::debug("Intersections merged");
00312
00313   // Make the road point to be outside the intersection
00314   spdlog::debug("Adding offsets to the roads ...");
00315   for (auto &i : intersections) {
00316     for (auto &roadInfo : i.roadSegmentIds) {
00317       double dx =
00318           roads[roadInfo.first].segments[roadInfo.second].p2.x -
      roads[roadInfo.first].segments[roadInfo.second].p1.x;
00319       double dy =
00320           roads[roadInfo.first].segments[roadInfo.second].p2.y -
      roads[roadInfo.first].segments[roadInfo.second].p1.y;
00321       double dd = distance({0, 0}, {(float)dx, (float)dy});
00322       dx /= dd;
00323       dy /= dd;
00324
00325       double radius = i.radius;
00326
00327       if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
00328           distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
00329         roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
00330         roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
00331       } else {
00332         dx = -dx;
00333         dy = -dy;
00334         roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
00335         roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
00336       }
00337     }
00338   }
00339   spdlog::debug("Offsets added");
00340
00341   // Remove the intersections that link the same road
00342   spdlog::debug("Removing intersections that link the same road ...");
00343   for (int i = 0; i < (int)intersections.size(); i++) {
00344     if (intersections[i].roadSegmentIds.size() != 2)
00345       continue;
00346
00347     if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
00348       intersections.erase(intersections.begin() + i);
00349       i -= 1;
00350     }
00351   }
00352   spdlog::debug("Intersections removed");
00353
00354   // Log all the intersections and roads
00355   for (auto r : roads) {
00356     spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes,
      r.segments.size());
```

```
00357   }
00358   for (auto i : intersections) {
00359     spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id,
      i.center.x, i.center.y,
00360                   i.radius, i.roadSegmentIds.size());
00361   }
00362
00363   std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
00364   spdlog::info("Intersections loaded ({} ms)",
00365                std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());
00366
00367   spdlog::info("Number of roads: {}", roads.size());
00368   spdlog::info("Number of buildings: {}", buildings.size());
00369   spdlog::info("Number of intersections: {}", intersections.size());
00370
00371   spdlog::info("Width: {} m", width);
00372   spdlog::info("Height: {} m", height);
00373
00374   isLoaded = true;
00375 }
```

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

- cityMap.h
- cityMap.cpp

6.19 DataManager Class Reference

Data manager.

`#include <dataManager.h>`

Public Types

- using data = _data

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.

6.19.1 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.

6.19.2 Member Typedef Documentation

6.19.2.1 data

```
using DataManager::data = _data
```

Definition at line 32 of file dataManager.h.

6.19.3 Constructor & Destructor Documentation

6.19.3.1 DataManager()

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

Constructor.

Parameters

filename	The filename

Definition at line 18 of file dataManager.cpp.

```
00018                                       {
00019   // Create /data folder if it doesn't exist
00020   if (!std::filesystem::exists("data")) {
00021     spdlog::debug("Creating data folder");
00022     std::filesystem::create_directory("data");
00023   }
00024 }
```

6.19.4 Member Function Documentation

6.19.4.1 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

numData	The number of data
numCarsMin	The minimum number of cars
numCarsMax	The maximum number of cars
mapName	The map name

Definition at line 26 of file dataManager.cpp.

```
00026                                                                                                {
00027
00028   spdlog::error("Deprecated: Need to be updated to use the new manager system");
00029
00030   return;
00031
00032   // // If numData is less than 1, default to a very high number (as in your original code).
00033   // numData = numData < 1 ? INT_MAX : numData;
00034   //
00035   // // Remove file extension from mapName to construct the output filename.
00036   // std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
00037   // std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
00038   //                        (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";
00039   //
00040   // // Load the city map.
00041   // CityMap cityMap;
00042   // cityMap.loadFile("assets/map/" + mapName);
00043   //
00044   // // Create the city graph.
00045   // CityGraph cityGraph;
00046   // cityGraph.createGraph(cityMap);
00047   //
00048   // // Open the output file in append mode.
00049   // std::ofstream file;
00050   // file.open(filename, std::ios::app);
00051   // if (!file.is_open()) {
00052   //   spdlog::error("Failed to open file {}", filename);
00053   //   return;
00054   // }
00055   //
00056   // std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
00057   // std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);
00058   //
00059   // for (int i = 0; i < numData; i += 1) {
00060   //   int numCars = dist(rng);
00061   //
00062   //   Manager manager(cityGraph, cityMap, false);
00063   //   auto resData = manager.createCarsCBS(numCars);
00064   //   if (!resData.first) {
00065   //     spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
00066   //     i--;
00067   //     continue;
00068   //   }
00069   //
00070   //   data validResData = resData.second;
00071   //
00072   //   file « validResData.numCars « ";" « validResData.carDensity;
00073   //   for (auto speed : validResData.carAvgSpeed) {
00074   //     file « ";" « speed;
00075   //   }
00076   //   file « std::endl;
00077   //
00078   //   if (numData == INT_MAX) {
00079   //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars,
00080   //     validResData.carDensity);
00081   //   } else {
00082   //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData,
      validResData.numCars,
00083   //                  validResData.carDensity);
00084   //   }
00085   // }
00086   //
00087   // file.close();
00088 }
```

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

- dataManager.h
- dataManager.cpp

6.20 DubinsInterpolator Class Reference

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

Public Member Functions

- void init (_cityGraphPoint start, _cityGraphPoint end, double radius)

  Initialize the Dubins path with start and end points and a radius.
- _cityGraphPoint get (double time, double startSpeed, double endSpeed)

  Get the position at a certain time.
- double getDuration (double startSpeed, double endSpeed)

  Get the duration of the Dubins path based on the start and end speeds.
- double getDistance ()

  Get the distance between the start and end points depending on the dubins path.

6.20.1 Detailed Description

Definition at line 15 of file dubins.h.

6.20.2 Member Function Documentation

6.20.2.1 get()

```
CityGraph::point DubinsInterpolator::get (
            double time,
            double startSpeed,
            double endSpeed )
```

Get the position at a certain time.

Parameters

time	The time
startSpeed	The speed at the start point
endSpeed	The speed at the end point

Returns

The position at the time

Definition at line 70 of file interpolator.cpp.

```
00070                                                                                {
00071     double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);
00072     auto xFun = [&](double t) { return (0.5 * acc * std::pow(t, 2) + startSpeed * t) / distance; };
00073
00074     int index = std::round((numInterpolatedPoints - 1) * xFun(time));
00075     index = std::clamp(index, 0, numInterpolatedPoints - 1);
00076
00077     return interpolatedCurve[index];
00078 }
```

6.20.2.2 getDistance()

```
double DubinsInterpolator::getDistance ( )  [inline]
```

Get the distance between the start and end points depending on the dubins path.

Returns

The distance

Definition at line 46 of file dubins.h.

```
00046 { return distance; }
```

6.20.2.3 getDuration()

```
double DubinsInterpolator::getDuration (
            double startSpeed,
            double endSpeed )  [inline]
```

Get the duration of the Dubins path based on the start and end speeds.

Parameters

startSpeed	The speed at the start point
endSpeed	The speed at the end point

Returns

The duration of the Dubins path

Definition at line 40 of file dubins.h.

```
00040 { return 2 * distance / (startSpeed + endSpeed); }
```

6.20.2.4 init()

```
void DubinsInterpolator::init (
            _cityGraphPoint start,
            _cityGraphPoint end,
            double radius )
```

Initialize the Dubins path with start and end points and a radius.

Parameters

start	The start point
end	The end point
radius	The turning radius

Definition at line 10 of file interpolator.cpp.

```
00010                                                                                                {
00011     startPoint = start_;
00012     endPoint = end_;
00013     radius = radius_;
00014
00015     ob::DubinsStateSpace space = ob::DubinsStateSpace(radius, true);
00016
00017     ob::State *start = space.allocState();
00018     ob::State *end = space.allocState();
00019
00020     start->as<ob::DubinsStateSpace::StateType>()->setXY(startPoint.position.x, startPoint.position.y);
00021     start->as<ob::DubinsStateSpace::StateType>()->setYaw(startPoint.angle.asRadians());
```

```
00022
00023   end->as<ob::DubinsStateSpace::StateType>()->setXY(endPoint.position.x, endPoint.position.y);
00024   end->as<ob::DubinsStateSpace::StateType>()->setYaw(endPoint.angle.asRadians());
00025
00026   distance = space.distance(start, end);
00027
00028   sf::Vector2 diff = startPoint.position - endPoint.position;
00029   double absDist = std::sqrt(std::pow(diff.x, 2) + std::pow(diff.y, 2));
00030   if (distance > absDist + 2 * M_PI * radius) {
00031     spdlog::warn("Distance is way too big in DubinsInterpolator");
00032     distance = absDist;
00033   }
00034   if (distance + 0.1 < absDist) { // 0.1 is a small tolerance
00035     spdlog::warn("Distance is way too small in DubinsInterpolator");
00036     distance = absDist;
00037   }
00038
00039   double dx = DUBINS_INTERPOLATION_STEP / distance;
00040   interpolatedCurve.clear();
00041   interpolatedCurve.push_back(startPoint);
00042
00043   for (double x = dx; x < 1; x += dx) {
00044     if (x == 1)
00045       continue;
00046
00047     ob::State *state = space.allocState();
00048     space.interpolate(start, end, x, state);
00049     double x_ = state->as<ob::DubinsStateSpace::StateType>()->getX();
00050     double y_ = state->as<ob::DubinsStateSpace::StateType>()->getY();
00051     double yaw_ = state->as<ob::DubinsStateSpace::StateType>()->getYaw();
00052
00053     CityGraph::point point;
00054     point.position = {(float)x_, (float)y_};
00055     point.angle = sf::radians(yaw_);
00056
00057     interpolatedCurve.push_back(point);
00058
00059     space.freeState(state);
00060   }
00061
00062   interpolatedCurve.push_back(endPoint);
00063
00064   numInterpolatedPoints = interpolatedCurve.size();
00065
00066   space.freeState(start);
00067   space.freeState(end);
00068 }
```

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

- dubins.h
- interpolator.cpp

6.21 FileSelector Class Reference

A file selector.

`#include <fileSelector.h>`

Public Member Functions

- FileSelector (const std::string &path)
- ~FileSelector ()
- std::string selectFile ()

6.21.1 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 20 of file fileSelector.h.

6.21.2 Constructor & Destructor Documentation

6.21.2.1 FileSelector()

```
FileSelector::FileSelector (
            const std::string & path )  [inline]
```

Definition at line 33 of file fileSelector.h.

```
00033 : folderPath(path), selectedIndex(0) { loadFiles(); }
```

6.21.2.2 ~FileSelector()

```
FileSelector::~FileSelector ( )  [inline]
```

Definition at line 34 of file fileSelector.h.

```
00034 { std::cout « "\033[?25h"; }
```

6.21.3 Member Function Documentation

6.21.3.1 selectFile()

```
std::string FileSelector::selectFile ( )
```

Definition at line 64 of file fileSelector.cpp.

```
00064                                    {
00065   std::cout « "\033[?25l";
00066   if (files.empty()) {
00067     spdlog::error("No .osm files found in the folder: {}", folderPath);
00068     return "";
00069   }
00070
00071   displayFiles();
00072
00073   while (true) {
00074     char key = getKeyPress();
00075     if (key == 27) {
00076       if (getKeyPress() == '[') {
00077         switch (getKeyPress()) {
00078         case 'A':
00079           moveCursorUp();
00080           break;
00081         case 'B':
00082           moveCursorDown();
00083           break;
00084         }
00085       }
00086     } else if (key == '\n') {
00087       std::cout « "\033[" « selectedIndex + 1 « "A\033[2K\r" « std::flush;
00088       std::cout « "\033[?25h";
00089       spdlog::info("Selected file: {}", files[selectedIndex]);
00090       return files[selectedIndex];
00091     }
00092   }
00093 }
```

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

- fileSelector.h
- fileSelector.cpp

6.22 std::hash< _aStarConflict > Struct Reference

`#include <aStar.h>`

Public Member Functions

- std::size_t operator() (const _aStarConflict &conflict) const

6.22.1 Detailed Description

Definition at line 60 of file aStar.h.

6.22.2 Member Function Documentation

6.22.2.1 operator()()

```
std::size_t std::hash< _aStarConflict >::operator() (
            const _aStarConflict & conflict ) const  [inline]
```

Definition at line 61 of file aStar.h.

```
00061                                                       {
00062     return std::hash<_cityGraphPoint>()(conflict.point) ^ std::hash<int>()(conflict.time) ^
00063            std::hash<int>()(conflict.car);
00064   }
```

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

- aStar.h

6.23 std::hash< _aStarNode > Struct Reference

`#include <aStar.h>`

Public Member Functions

- std::size_t operator() (const _aStarNode &point) const

6.23.1 Detailed Description

Definition at line 52 of file aStar.h.

6.23.2 Member Function Documentation

6.23.2.1 operator()()

```
std::size_t std::hash< _aStarNode >::operator() (
            const _aStarNode & point ) const  [inline]
```

Definition at line 53 of file aStar.h.

```
00053                                                   {
00054     double s = std::round(point.speed / SPEED_RESOLUTION);
00055
00056     return std::hash<_cityGraphPoint>()(point.point) ^ std::hash<double>()(s) ^
00057            std::hash<_cityGraphPoint>()(point.arcFrom.first) ^
      std::hash<CityGraph::neighbor>()(point.arcFrom.second);
00058   }
```

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

- aStar.h

6.24 std::hash< _cityGraphNeighbor > Struct Reference

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

Public Member Functions

- std::size_t operator() (const _cityGraphNeighbor &neighbor) const

6.24.1 Detailed Description

Definition at line 66 of file cityGraph.h.

6.24.2 Member Function Documentation

6.24.2.1 operator()()

```
std::size_t std::hash< _cityGraphNeighbor >::operator() (
            const _cityGraphNeighbor & neighbor ) const  [inline]
```

Definition at line 67 of file cityGraph.h.

```
00067                                                           {
00068     return std::hash<_cityGraphPoint>()(neighbor.point) ^ std::hash<double>()(neighbor.maxSpeed) ^
00069            std::hash<double>()(neighbor.turningRadius) ^ std::hash<bool>()(neighbor.isRightWay);
00070   }
```

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

- cityGraph.h

6.25 std::hash< _cityGraphPoint > Struct Reference

`#include <cityGraph.h>`

Public Member Functions

- std::size_t operator() (const _cityGraphPoint &point) const

6.25.1 Detailed Description

Definition at line 57 of file cityGraph.h.

6.25.2 Member Function Documentation

6.25.2.1 operator()()

```
std::size_t std::hash< _cityGraphPoint >::operator() (
            const _cityGraphPoint & point ) const  [inline]
```

Definition at line 58 of file cityGraph.h.

```
00058                                                 {
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(point.angle.asRadians() / ANGLE_RESOLUTION);
00062
00063     return std::hash<int>()(x) ^ std::hash<int>()(y) ^ std::hash<int>()(a);
00064   }
```

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

- cityGraph.h

6.26 std::hash< _managerOCBSConflict > Struct Reference

`#include <manager_ocbs.h>`

Public Member Functions

- std::size_t operator() (const _managerOCBSConflict &point) const

6.26.1 Detailed Description

Definition at line 48 of file manager_ocbs.h.

6.26.2 Member Function Documentation

6.26.2.1 operator()()

```
std::size_t std::hash< _managerOCBSConflict >::operator() (
            const _managerOCBSConflict & point ) const  [inline]
```

Definition at line 49 of file manager_ocbs.h.

```
00049                                                                {
00050     return std::hash<int>()(point.car) ^ std::hash<int>()(point.withCar) ^
      std::hash<double>()(point.time) ^
00051            std::hash<float>()(point.position.x) ^ std::hash<float>()(point.position.y);
00052   }
```

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

- manager_ocbs.h

6.27 std::hash< _managerOCBSConflictSituation > Struct Reference

```
#include <manager_ocbs.h>
```

Public Member Functions

- std::size_t operator() (const _managerOCBSConflictSituation &point) const

6.27.1 Detailed Description

Definition at line 40 of file manager_ocbs.h.

6.27.2 Member Function Documentation

6.27.2.1 operator()()

```
std::size_t std::hash< _managerOCBSConflictSituation >::operator() (
            const _managerOCBSConflictSituation & point ) const  [inline]
```

Definition at line 41 of file manager_ocbs.h.

```
00041                                                                        {
00042     int t = std::round(point.time / OCBS_CONFLICT_RANGE);
00043     int x = std::round(point.at.x / CAR_LENGTH);
00044     int y = std::round(point.at.y / CAR_LENGTH);
00045     return std::hash<int>()(point.car) ^ std::hash<int>()(t) ^ std::hash<int>()(x) ^
      std::hash<int>()(y);
00046   }
```

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

- manager_ocbs.h

6.28 std::hash< std::pair< _cityGraphPoint, _cityGraphNeighbor > > Struct Reference

`#include <cityGraph.h>`

Public Member Functions

- std::size_t operator() (const std::pair< _cityGraphPoint, _cityGraphNeighbor > &pair) const

6.28.1 Detailed Description

Definition at line 72 of file cityGraph.h.

6.28.2 Member Function Documentation

6.28.2.1 operator()()

```
std::size_t std::hash< std::pair< _cityGraphPoint, _cityGraphNeighbor > >::operator() (
            const std::pair< _cityGraphPoint, _cityGraphNeighbor > & pair ) const  [inline]
```

Definition at line 73 of file cityGraph.h.

```
00073                                                                                {
00074     return std::hash<_cityGraphPoint>()(pair.first) ^ std::hash<_cityGraphNeighbor>()(pair.second);
00075   }
```

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

- cityGraph.h

6.29 Manager Class Reference

A manager for the cars.

`#include <manager.h>`

Inherited by ManagerOCBS.

Public Member Functions

- Manager (const CityGraph &cityGraph, const CityMap &CityMap)
  Constructor.
- virtual void initializeAgents (int numAgents)
  Initialize agents and set up the system.
- virtual void planPaths ()=0
  Using the created agents, create a path for each agent using an algorithm.
- virtual void updateAgents ()
  Make a simulation step.
- virtual void userInput (sf::Event event, sf::RenderWindow &window)
  Process user input.
- virtual void renderAgents (sf::RenderWindow &window) final
  Render the agents based on their current position.
- virtual int getNumAgents ()
  Get the number of agents.
- virtual std::vector< Car > getCars ()
  Get the cars.

Protected Attributes

- int numCars
- std::vector< Car > cars
- CityGraph graph
- CityMap map

6.29.1 Detailed Description

A manager for the cars.

The manager class is used to manage the cars during any pathfinding algorithm. It is used to create abstract managers like a CBS one.

Definition at line 23 of file manager.h.

6.29.2 Constructor & Destructor Documentation

6.29.2.1 Manager()

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

Constructor.

Parameters

cityGraph	The city graph
CityMap	The city map

Definition at line 30 of file manager.h.

```
00030 : graph(cityGraph), map(CityMap) {}
```

6.29.3 Member Function Documentation

6.29.3.1 getCars()

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

Get the cars.

Returns

The cars

Definition at line 74 of file manager.h.

```
00074 { return cars; }
```

6.29.3.2 getNumAgents()

```
virtual int Manager::getNumAgents ( )  [inline], [virtual]
```

Get the number of agents.

Returns

The number of agents

Definition at line 68 of file manager.h.

```
00068 { return numCars; }
```

6.29.3.3 initializeAgents()

```
void Manager::initializeAgents (
            int numAgents )  [virtual]
```

Initialize agents and set up the system.

Parameters

numCars	The number of agents

Definition at line 9 of file index.cpp.

```
00009                                          {
00010   spdlog::info("Initializing {} agent...", numCars);
00011   this->numCars = numCars;
00012
00013   for (int i = 0; i < numCars; i++) {
00014     Car car;
00015     cars.push_back(car);
00016   }
00017
00018   // Create a path for each car (random start and end points)
00019   for (int i = 0; i < numCars; i++) {
00020     bool valid = false;
00021     cars[i].chooseRandomStartEndPath(graph, map);
00022   }
00023
00024   spdlog::info("Initialized {} agents", cars.size());
00025 }
```

6.29.3.4 planPaths()

```
virtual void Manager::planPaths ( )  [pure virtual]
```

Using the created agents, create a path for each agent using an algorithm.

This function is used to create a path for each agent using an algorithm. The algorithm is not specified in this class, but it is expected to be implemented in a derived class.

Implemented in ManagerOCBS.

6.29.3.5 renderAgents()

```
void Manager::renderAgents (
            sf::RenderWindow & window )  [final], [virtual]
```

Render the agents based on their current position.

Parameters

window	The window

Definition at line 33 of file index.cpp.

```
00033                                                  {
00034   for (Car &car : cars) {
00035     car.render(window);
00036   }
00037 }
```

6.29.3.6 updateAgents()

```
void Manager::updateAgents ( )  [virtual]
```

Make a simulation step.

Definition at line 27 of file index.cpp.

```
00027                              {
00028   for (Car &car : cars) {
00029     car.move();
00030   }
00031 }
```

6.29.3.7 userInput()

```
virtual void Manager::userInput (
            sf::Event event,
            sf::RenderWindow & window )  [inline], [virtual]
```

Process user input.

Parameters

event	The event
window	The window

Reimplemented in ManagerOCBS.

Definition at line 56 of file manager.h.

```
00056 {};
```

6.29.4 Member Data Documentation

6.29.4.1 cars

```
std::vector<Car> Manager::cars  [protected]
```

Definition at line 78 of file manager.h.

6.29.4.2 graph

```
CityGraph Manager::graph  [protected]
```

Definition at line 79 of file manager.h.

6.29.4.3 map

```
CityMap Manager::map  [protected]
```

Definition at line 80 of file manager.h.

6.29.4.4 numCars

```
int Manager::numCars  [protected]
```

Definition at line 77 of file manager.h.

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

- manager.h
- index.cpp

6.30 ManagerOCBS Class Reference

Manager for the CBS algorithm This class is responsible for managing the agents and their paths using the Conflict-Based Search (CBS) algorithm. It inherits from the Manager class and implements the pathfinding logic specific to the CBS algorithm. This class initializes paths for agents, handles user input, and plans paths using the CBS algorithm.

```
#include <manager_ocbs.h>
```

Inherits Manager.

Public Types

- using ConflictSituation = _managerOCBSConflictSituation
- using Conflict = _managerOCBSConflict
- using Node = _managerOCBSNode

Public Member Functions

- ManagerOCBS (const CityGraph &cityGraph, const CityMap &cityMap)

  Constructor.
- void initializePaths (Node *node)

  Initialize agents and set up the system.
- void userInput (sf::Event event, sf::RenderWindow &window) override

  Make a simulation step.
- void planPaths () override

  Using the created agents, create a path for each agent using an algorithm.

Public Member Functions inherited from Manager

- Manager (const CityGraph &cityGraph, const CityMap &CityMap)

  Constructor.
- virtual void initializeAgents (int numAgents)

  Initialize agents and set up the system.
- virtual void updateAgents ()

  Make a simulation step.
- virtual void renderAgents (sf::RenderWindow &window) final

  Render the agents based on their current position.
- virtual int getNumAgents ()

  Get the number of agents.
- virtual std::vector< Car > getCars ()

  Get the cars.

Additional Inherited Members

Protected Attributes inherited from Manager

- int numCars
- std::vector< Car > cars
- CityGraph graph
- CityMap map

6.30.1 Detailed Description

Manager for the CBS algorithm This class is responsible for managing the agents and their paths using the Conflict-Based Search (CBS) algorithm. It inherits from the Manager class and implements the pathfinding logic specific to the CBS algorithm. This class initializes paths for agents, handles user input, and plans paths using the CBS algorithm.

Definition at line 77 of file manager_ocbs.h.

6.30.2 Member Typedef Documentation

6.30.2.1 Conflict

```
using ManagerOCBS::Conflict = _managerOCBSConflict
```

Definition at line 80 of file manager_ocbs.h.

6.30.2.2 ConflictSituation

```
using ManagerOCBS::ConflictSituation = _managerOCBSConflictSituation
```

Definition at line 79 of file manager_ocbs.h.

6.30.2.3 Node

```
using ManagerOCBS::Node = _managerOCBSNode
```

Definition at line 81 of file manager_ocbs.h.

6.30.3 Constructor & Destructor Documentation

6.30.3.1 ManagerOCBS()

```
ManagerOCBS::ManagerOCBS (
            const CityGraph & cityGraph,
            const CityMap & cityMap )  [inline]
```

Constructor.

Parameters

cityGraph	The city graph
CityMap	The city map

Definition at line 88 of file manager_ocbs.h.

```
00088 : Manager(cityGraph, cityMap) {}
```

6.30.4 Member Function Documentation

6.30.4.1 initializePaths()

```
void ManagerOCBS::initializePaths (
            Node * node )
```

Initialize agents and set up the system.

Parameters

numCars	The number of agents

Definition at line 54 of file ocbs.cpp.

```
00054                                                {
00055   for (int i = 0; i < numCars; i++) {
00056     spdlog::debug("Finding path for car {}", i);
00057     pathfinding(node, i);
00058   }
00059 }
```

6.30.4.2 planPaths()

```
void ManagerOCBS::planPaths ( )  [override], [virtual]
```

Using the created agents, create a path for each agent using an algorithm.

This function is used to create a path for each agent using an algorithm. The algorithm is not specified in this class, but it is expected to be implemented in a derived class.

Implements Manager.

Definition at line 23 of file ocbs.cpp.

```
00023                                {
00024   openSet = std::priority_queue<Node>();
00025   starts.clear();
00026   starts.resize(numCars);
00027   ends.clear();
00028   ends.resize(numCars);
00029   baseCosts.clear();
00030   baseCosts.resize(numCars);
00031
00032   Node node;
00033   node.paths.resize(numCars);
00034   node.costs.resize(numCars);
00035   node.cost = 0;
00036   node.depth = 0;
00037   node.hasResolved = false;
00038   conflicts.clear();
00039
00040   for (int i = 0; i < numCars; i++) {
00041     node.paths[i] = cars[i].getPath();
00042     node.costs[i] = cars[i].getPathTime();
00043     node.cost += node.costs[i];
00044     baseCosts[i] = node.costs[i];
00045     starts[i] = cars[i].getStart();
00046     ends[i] = cars[i].getEnd();
00047   }
00048
00049   openSet.push(node);
00050   spdlog::info("Starting to find paths using CBS");
00051   findPaths();
00052 }
```

6.30.4.3 userInput()

```
void ManagerOCBS::userInput (
             sf::Event event,
             sf::RenderWindow & window )  [override], [virtual]
```

Make a simulation step.

Reimplemented from Manager.

Definition at line 6 of file ocbs.cpp.

```
00006                                                          {
00007   // If left mouse click over a car, toggle debug for that car
00008   if (event.is<sf::Event::MouseButtonPressed>() &&
00009       event.getIf<sf::Event::MouseButtonPressed>()->button == sf::Mouse::Button::Left) {
00010     sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
00011     for (int i = 0; i < numCars; i++) {
00012       sf::Vector2f diff = cars[i].getPosition() - mousePos;
00013       double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00014       if (len < 2 * CAR_LENGTH) {
00015         cars[i].toggleDebug();
00016         spdlog::debug("Toggling debug for car {}", i);
00017         return;
00018       }
00019     }
00020   }
00021 }
```

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

- manager_ocbs.h
- ocbs.cpp

6.31 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.

6.31.1 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.

6.31.2 Member Function Documentation

6.31.2.1 renderCityGraph()

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

Render the city graph.

Parameters

cityGraph	The city graph
view	The view

Definition at line 233 of file renderer.cpp.

```
00233                                                                    {
00234   std::unordered_set<CityGraph::point> graphPoints = cityGraph.getGraphPoints();
00235   std::unordered_map<CityGraph::point, std::vector<CityGraph::neighbor» neighbors =
      cityGraph.getNeighbors();
00236
00237   // Draw a line between each point and its neighbors
00238   for (const auto &point : graphPoints) {
```

```
00239     for (const auto &neighbor : neighbors[point]) {
00240       if (!neighbor.isRightWay)
00241         continue;
00242 
00243       double radius = turningRadius(neighbor.maxSpeed);
00244       auto space = ob::DubinsStateSpace(radius, true);
00245       ob::RealVectorBounds bounds(2);
00246       space.setBounds(bounds);
00247 
00248       // Draw only if one of the points is inside the view
00249       sf::Vector2f viewCenter = view.getCenter();
00250       sf::Vector2f viewSize = view.getSize();
00251       sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
00252       sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;
00253 
00254       if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
00255         continue;
00256       }
00257       if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
00258         continue;
00259       }
00260 
00261       ob::State *start = space.allocState();
00262       ob::State *end = space.allocState();
00263 
00264       start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
00265       start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle.asRadians());
00266 
00267       end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x,
    neighbor.point.position.y);
00268       end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle.asRadians());
00269 
00270       // Draw the Dubins curve
00271       double step = CELL_SIZE / 2.0f;
00272       double distance = space.distance(start, end);
00273       int numSteps = distance / step;
00274       sf::Vector2f lastPosition;
00275       sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);
00276 
00277       for (int k = 0; k < numSteps; k++) {
00278         if (k == 0) {
00279           lastPosition = {point.position.x, point.position.y};
00280           continue;
00281         }
00282 
00283         ob::State *state = space.allocState();
00284         space.interpolate(start, end, (double)k / (double)numSteps, state);
00285 
00286         double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
00287         double y = state->as<ob::DubinsStateSpace::StateType>()->getY();
00288 
00289         double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y,
    2));
00290         sf::Angle angle = sf::radians(atan2(y - lastPosition.y, x - lastPosition.x));
00291 
00292         // Draw an arrow between the points
00293         drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor,
    false);
00294 
00295         lastPosition = {(float)x, (float)y};
00296       }
00297 
00298       continue;
00299       // Write the speed of the point
00300       sf::Font font = loadFont();
00301       sf::Text text(font);
00302       text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
00303       text.setCharacterSize(24);
00304       text.setFillColor(sf::Color::Black);
00305       text.setOutlineColor(sf::Color::White);
00306       text.setOutlineThickness(1.0f);
00307       text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
00308       text.setScale({0.02f, 0.02f});
00309       text.setOrigin({text.getLocalBounds().size.x / 2.0f, text.getLocalBounds().size.y / 2.0f});
00310       window.draw(text);
00311     }
00312 
00313     // Draw a dot at each points
00314     double size = 0.3;
00315     sf::CircleShape circle(size);
00316     circle.setFillColor(sf::Color(255, 0, 0, 70));
00317     circle.setPosition({(float)(point.position.x - size), (float)(point.position.y - size)});
00318     window.draw(circle);
00319   }
00320 }
```

6.31.2.2 renderCityMap()

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

Render the city map.

Parameters

cityMap	The city map

Definition at line 130 of file renderer.cpp.

```
00130                                                {
00131   // Draw buildings
00132   std::vector<sf::Color> randomBuildingColors = {
00133       sf::Color(233, 234, 232), sf::Color(238, 231, 210), sf::Color(230, 229, 226), sf::Color(236,
      234, 230),
00134       sf::Color(230, 223, 216), sf::Color(230, 234, 236), sf::Color(210, 215, 222)};
00135
00136   std::vector<sf::Color> greenAreaColor = {sf::Color(184, 230, 144), sf::Color(213, 240, 193)};
00137
00138   sf::Color waterColor(139, 214, 245);
00139
00140   auto greenAreas = cityMap.getGreenAreas();
00141   for (int i = 0; i < (int)greenAreas.size(); i++) {
00142     const auto &greenArea = greenAreas[i];
00143     auto points = greenArea.points;
00144     sf::ConvexShape convex;
00145     convex.setPointCount(points.size());
00146     for (size_t i = 0; i < points.size(); i++) {
00147       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00148     }
00149     convex.setFillColor(greenAreaColor[greenArea.type]);
00150
00151     window.draw(convex);
00152   }
00153
00154   auto waterAreas = cityMap.getWaterAreas();
00155   for (int i = 0; i < (int)waterAreas.size(); i++) {
00156     const auto &waterArea = waterAreas[i];
00157     auto points = waterArea.points;
00158     sf::ConvexShape convex;
00159     convex.setPointCount(points.size());
00160     for (size_t i = 0; i < points.size(); i++) {
00161       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00162     }
00163     convex.setFillColor(waterColor);
00164
00165     window.draw(convex);
00166   }
00167
00168   auto buildings = cityMap.getBuildings();
00169   for (int i = 0; i < (int)buildings.size(); i++) {
00170     const auto &building = buildings[i];
00171     auto points = building.points;
00172     sf::ConvexShape convex;
00173     convex.setPointCount(points.size());
00174     for (size_t i = 0; i < points.size(); i++) {
00175       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00176     }
00177     convex.setFillColor(randomBuildingColors[i % randomBuildingColors.size()]);
00178
00179     window.draw(convex);
00180   }
00181
00182   // Draw roads
00183   sf::Color roadColor(194, 201, 202);
00184   for (const auto &road : cityMap.getRoads()) {
00185     for (const auto &segment : road.segments) {
00186       sf::Vector2f basedP1(segment.p1.x, segment.p1.y);
00187       sf::Vector2f basedP2(segment.p2.x, segment.p2.y);
00188
00189       sf::Angle angle = segment.angle;
00190
00191       sf::Vector2f widthVec({sin(angle.asRadians()), -cos(angle.asRadians())});
00192       widthVec *= (float)road.width / 2;
00193
00194       sf::Vector2f p1 = basedP1 + widthVec;
00195       sf::Vector2f p2 = basedP1 - widthVec;
```

```
00196       sf::Vector2f p3 = basedP2 - widthVec;
00197       sf::Vector2f p4 = basedP2 + widthVec;
00198
00199       sf::ConvexShape convex;
00200       convex.setPointCount(4);
00201       convex.setPoint(0, p1);
00202       convex.setPoint(1, p2);
00203       convex.setPoint(2, p3);
00204       convex.setPoint(3, p4);
00205
00206       convex.setFillColor(roadColor);
00207
00208       window.draw(convex);
00209
00210       // Draw a circle at the start end end of the road (for filling the gap)
00211       double radius = road.width / 2;
00212       sf::CircleShape circle(radius);
00213       circle.setFillColor(roadColor);
00214       circle.setPosition({(float)(basedP1.x - radius), (float)(basedP1.y - radius)});
00215       window.draw(circle);
00216       circle.setPosition({(float)(basedP2.x - radius), (float)(basedP2.y - radius)});
00217       window.draw(circle);
00218     }
00219   }
00220
00221   // Draw intersections
00222   if (debug) {
00223     for (const auto &intersection : cityMap.getIntersections()) {
00224       double radius = intersection.radius;
00225       sf::CircleShape circle(radius);
00226       circle.setFillColor(sf::Color(0, 255, 0, 50));
00227       circle.setPosition({(float)(intersection.center.x - radius), (float)(intersection.center.y -
      radius)});
00228       window.draw(circle);
00229     }
00230   }
00231 }
```

6.31.2.3 renderManager()

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

Render the cars.

Parameters

manager	The manager

Definition at line 322 of file renderer.cpp.

```
00322 { manager.renderAgents(window); }
```

6.31.2.4 renderTime()

```
void Renderer::renderTime ( )
```

Render the time.

Definition at line 324 of file renderer.cpp.

```
00324                                {
00325   // At the top right corner of the view (keep the same size even if the view is resized)
00326   sf::Font font = loadFont();
00327   sf::Text text(font);
00328   sf::Vector2f viewSize = window.getView().getSize();
00329   text.setCharacterSize(24);
00330   text.setFillColor(sf::Color::White);
00331   text.setPosition(window.getView().getCenter() + sf::Vector2f(viewSize.x / 2, -viewSize.y / 2) +
00332                    sf::Vector2f(-viewSize.x * 0.01f, viewSize.y * 0.01f));
00333   text.setString(std::to_string((int)time) + " s");
```

```
  text.setOutlineColor(sf::Color::Black);
  text.setOutlineThickness(1.0f);
  text.scale({viewSize.x * 0.001f, viewSize.x * 0.001f});
  text.setOrigin({text.getLocalBounds().size.x, 0});
  window.draw(text);
}
```

6.31.2.5 startRender()

```
void Renderer::startRender (
        const CityMap & cityMap,
        const CityGraph & cityGraph,
        Manager & manager )
```

Start the rendering.

Definition at line 20 of file renderer.cpp.

```
00020 {
00021   manager.planPaths();
00022 
00023   window.create(sf::VideoMode({SCREEN_WIDTH, SCREEN_HEIGHT}), "City Map");
00024 
00025   // Set the view to the center of the city map, allowing some basic camera movement
00026   // Arrow to move the camera, + and - to zoom in and out
00027   double height = cityMap.getHeight();
00028   double width = cityMap.getWidth();
00029   sf::View view(sf::FloatRect({0, 0}, {(float)width, (float)height}));
00030   // Reset view function
00031   auto resetView = [&]() {
00032     double screenRatio = window.getSize().x / (double)window.getSize().y;
00033     double cityRatio = width / height;
00034     view.setCenter({(float)width / 2, (float)height / 2});
00035     if (screenRatio > cityRatio) {
00036       view.setSize({(float)(height * screenRatio), (float)height});
00037     } else {
00038       view.setSize({(float)width, (float)(width / screenRatio)});
00039     }
00040     window.setView(view);
00041   };
00042 
00043   resetView();
00044   renderCityMap(cityMap);
00045   window.display();
00046   time = 0;
00047 
00048   sf::Clock clockCars;
00049   bool speedUp = false;
00050   bool pause = true;
00051 
00052   while (true) {
00053     while (const std::optional event = window.pollEvent()) {
00054       if (event->is<sf::Event::Closed>()) {
00055         window.close();
00056         return;
00057       }
00058 
00059       if (event->is<sf::Event::KeyPressed>() || event->is<sf::Event::MouseButtonPressed>()) {
00060         manager.userInput(event.value(), window);
00061       }
00062 
00063       if (const auto *resized = event->getIf<sf::Event::Resized>()) {
00064         resetView();
00065       }
00066 
00067       if (!event->is<sf::Event::KeyPressed>())
00068         continue;
00069 
00070       if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Escape) {
00071         window.close();
00072       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Up) {
00073         view.move({0, -(float)(height * MOVE_SPEED)});
00074       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Down) {
00075         view.move({0, +(float)(height * MOVE_SPEED)});
00076       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Left) {
00077         view.move({-(float)(width * MOVE_SPEED), 0});
00078       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Right) {
00079         view.move({+(float)(width * MOVE_SPEED), 0});
00080       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Equal) {
00081         view.zoom(1.0f - ZOOM_SPEED);
```

```
00082       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Subtract) {
00083         view.zoom(1.0f + ZOOM_SPEED);
00084       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::R) {
00085         resetView();
00086         spdlog::debug("View reset");
00087       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::D) {
00088         debug = !debug;
00089         spdlog::debug("Debug mode: {}", debug);
00090       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::S) {
00091         speedUp = !speedUp;
00092       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::P) {
00093         pause = !pause;
00094       }
00095     }
00096
00097     window.setView(view);
00098     window.clear(sf::Color(247, 246, 242));
00099     renderCityMap(cityMap);
00100     renderManager(manager);
00101     if (!pause) {
00102       if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
00103           (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
00104         time += SIM_STEP_TIME;
00105         manager.updateAgents();
00106         clockCars.restart();
00107       }
00108     }
00109     if (debug) {
00110       renderCityGraph(cityGraph, view);
00111     }
00112     // Remove outside the border (draw blank)
00113     sf::RectangleShape rectangle(sf::Vector2f(width, height));
00114     rectangle.setFillColor(sf::Color(247, 246, 242));
00115
00116     float w = width;
00117     float h = height;
00118
00119     std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h},
      {-w, 0}};
00120     for (auto b : border) {
00121       rectangle.setPosition(b);
00122       window.draw(rectangle);
00123     }
00124
00125     renderTime();
00126     window.display();
00127   }
00128 }
```

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

- renderer.h
- renderer.cpp

6.32 Test Class Reference

A class for testing the project.

`#include <test.h>`

Public Member Functions

- void runTests ()

  Run the tests.

6.32.1 Detailed Description

A class for testing the project.

This class is used to test the project.

Definition at line 13 of file test.h.

6.32.2 Member Function Documentation

6.32.2.1 runTests()

```
void Test::runTests ( )
```

Run the tests.

Definition at line 13 of file test.cpp.

```
00013                       {
00014   testSpdlog();
00015   testTinyXML2();
00016   testSFML();
00017 }
```

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

- test.h
- test.cpp

Chapter 7

File Documentation

7.1 aStar.h File Reference

A∗ algorithm.

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

Classes

- struct _aStarNode
  A node for the A∗ algorithm.
- struct _aStarConflict
  A conflict for the A∗ algorithm.
- struct std::hash< _aStarNode >
- struct std::hash< _aStarConflict >
- class AStar
  A∗ algorithm.

Namespaces

- namespace std

Typedefs

- typedef struct _aStarNode _aStarNode
- typedef struct _aStarConflict _aStarConflict

7.1.1 Detailed Description

A∗ algorithm.

This file contains the declaration of the AStar class. This class represents the A∗ algorithm, which is used to find the shortest path between two points in a graph.

Definition in file aStar.h.

7.1.2 Typedef Documentation

7.1.2.1 _aStarConflict

```
typedef struct _aStarConflict _aStarConflict
```

7.1.2.2 _aStarNode

```
typedef struct _aStarNode _aStarNode
```

7.2 aStar.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "cityGraph.h"
00011 #include "config.h"
00012
00020 typedef struct _aStarNode {
00021   _cityGraphPoint point;
00022   double speed;
00023   std::pair<_cityGraphPoint, _cityGraphNeighbor> arcFrom;
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   _cityGraphPoint point;
00043   int time;
00044   int car;
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<_cityGraphPoint>()(point.point) ^ std::hash<double>()(s) ^
00057            std::hash<_cityGraphPoint>()(point.arcFrom.first) ^
     std::hash<CityGraph::neighbor>()(point.arcFrom.second);
00058   }
00059 };
00060 template <> struct hash<_aStarConflict> {
00061   std::size_t operator()(const _aStarConflict &conflict) const {
00062     return std::hash<_cityGraphPoint>()(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 };
```

7.3 car.h File Reference

A car in the city.

```
#include "aStar.h"
#include "cityGraph.h"
#include "dubins.h"
#include <vector>
```

Classes

- class Car

  A car in the city.

Functions

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

  Check if two cars have a conflict.

7.3.1 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.

7.3.2 Function Documentation

7.3.2.1 carConflict()

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

Check if two cars have a conflict.

Parameters

carPos	The position of the car
carAngle	The angle of the car
confPos	The position of the conflicting car
confAngle	The angle of the conflicting car

Returns

If the cars have a conflict

Definition at line 47 of file utils.cpp.

```
00047                                                                                               {
00048     sf::Vector2f diff = carPos - confPos;
00049     return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00050 
00051     sf::Vector2f p11 = carPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle.asRadians()),
00052                                               CAR_LENGTH / 2.0f * sin(carAngle.asRadians()));
00053     sf::Vector2f p12 = carPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle.asRadians()),
00054                                               CAR_LENGTH / 2.0f * sin(carAngle.asRadians()));
00055     sf::Vector2f p21 = confPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00056                                                CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00057     sf::Vector2f p22 = confPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00058                                                CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00059 
00060     bool colides = false;
00061     colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00062     colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00063     colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00064     colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00065 
00066     return colides;
00067 }
```

7.3.2.2 carsCollided()

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

@bref Check if two cars collided

Parameters

car1	The first car
car2	The second car

Definition at line 21 of file utils.cpp.

```
00021                                                      {
00022     std::vector<sf::Vector2f> path1 = car1.getPath();
00023     std::vector<sf::Vector2f> path2 = car2.getPath();
00024     sf::Vector2f diff = path1[time] - path2[time];
00025     return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00026 
00027     sf::Vector2f pos1 = path1[time];
00028     sf::Vector2f pos2 = path2[time];
00029 
00030     double angle1 = atan2(path1[time + 1].y - path1[time].y, path1[time + 1].x - path1[time].x);
00031     double angle2 = atan2(path2[time + 1].y - path2[time].y, path2[time + 1].x - path2[time].x);
00032 
00033     sf::Vector2f p11 = pos1 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
      sin(angle1));
00034     sf::Vector2f p12 = pos1 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
      sin(angle1));
```

```
00035   sf::Vector2f p21 = pos2 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
      sin(angle2));
00036   sf::Vector2f p22 = pos2 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
      sin(angle2));
00037
00038   bool colides = false;
00039   colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00040   colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00041   colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00042   colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00043
00044   return colides;
00045 }
```

7.4 car.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "aStar.h"
00011 #include "cityGraph.h"
00012 #include "dubins.h"
00013 #include <vector>
00014
00022 class Car {
00023 public:
00027   Car();
00028
00034   void assignStartEnd(_cityGraphPoint start, _cityGraphPoint end) {
00035     this->start = start;
00036     this->end = end;
00037   }
00038
00044   void chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap);
00045
00050   void assignPath(std::vector<AStar::node> path, CityGraph &graph);
00051
00056   void assignExistingPath(std::vector<sf::Vector2f> path);
00057
00061   void move();
00062
00067   void render(sf::RenderWindow &window);
00068
00073   _cityGraphPoint getStart() { return start; }
00074
00079   _cityGraphPoint getEnd() { return end; }
00080
00085   double getSpeed();
00086
00092   double getSpeedAt(int index);
00093
00099   double getAverageSpeed(CityGraph &graph);
00100
00105   double getRemainingTime();
00106
00111   double getElapsedTime();
00112
00117   double getPathTime();
00118
00123   double getRemainingDistance();
00124
00129   double getElapsedDistance();
00130
00135   double getPathLength();
00136
00141   sf::Vector2f getPosition() { return path[currentPoint]; }
00142
00147   std::vector<sf::Vector2f> getPath() { return path; }
00148
00153   std::vector<AStar::node> getAStarPath() { return aStarPath; }
00154
00159   void toggleDebug() { debug = !debug; }
00160
00161 private:
00162   _cityGraphPoint start;
00163   _cityGraphPoint end;
00164   std::vector<sf::Vector2f> path;
00165   std::vector<AStar::node> aStarPath;
00166   int currentPoint = 0;
00167   bool debug = false;
```

```
00168   sf::Color color;
00169 };
00170
00176 bool carsCollided(Car car1, Car car2, int time);
00177
00186 bool carConflict(sf::Vector2f carPos, sf::Angle carAngle, sf::Vector2f confPos, sf::Angle confAngle);
```

7.5 cityGraph.h File Reference

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

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

Classes

- struct _cityGraphPoint

  A point in the city graph.
- struct _cityGraphNeighbor

  A neighbor of a point in the city graph.
- struct std::hash< _cityGraphPoint >
- struct std::hash< _cityGraphNeighbor >
- struct std::hash< std::pair< _cityGraphPoint, _cityGraphNeighbor > >
- class CityGraph

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

Namespaces

- namespace std

Typedefs

- typedef struct _cityGraphNeighbor _cityGraphNeighbor

7.5.1 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.

7.5.2 Typedef Documentation

7.5.2.1 _cityGraphNeighbor

```
typedef struct _cityGraphNeighbor _cityGraphNeighbor
```

7.6 cityGraph.h

Go to the documentation of this file.

```
00001
00007 #pragma once
00008
00009 #include "cityMap.h"
00010 #include "config.h"
00011 #include <unordered_set>
00012
00013 class DubinsInterpolator;
00014
00021 struct _cityGraphPoint {
00022   sf::Vector2f position;
00023   sf::Angle angle;
00025   bool operator==(const _cityGraphPoint &other) const {
00026     int x = std::round(position.x / CELL_SIZE);
00027     int y = std::round(position.y / CELL_SIZE);
00028     int a = std::round(angle.asRadians() / ANGLE_RESOLUTION);
00029     int oX = std::round(other.position.x / CELL_SIZE);
00030     int oY = std::round(other.position.y / CELL_SIZE);
00031     int oA = std::round(other.angle.asRadians() / ANGLE_RESOLUTION);
00032
00033     return x == oX && y == oY && a == oA;
00034   }
00035 };
00036
00044 typedef struct _cityGraphNeighbor {
00045   _cityGraphPoint point;
00046   double maxSpeed;
00047   double turningRadius;
00048   bool isRightWay;
00050   bool operator==(const _cityGraphNeighbor &other) const {
00051     return point == other.point && maxSpeed == other.maxSpeed && turningRadius == other.turningRadius
      &&
00052            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(point.angle.asRadians() / ANGLE_RESOLUTION);
00062
00063     return std::hash<int>()(x) ^ std::hash<int>()(y) ^ std::hash<int>()(a);
00064   }
00065 };
00066 template <> struct hash<_cityGraphNeighbor> {
00067   std::size_t operator()(const _cityGraphNeighbor &neighbor) const {
00068     return std::hash<_cityGraphPoint>()(neighbor.point) ^ std::hash<double>()(neighbor.maxSpeed) ^
00069            std::hash<double>()(neighbor.turningRadius) ^ std::hash<bool>()(neighbor.isRightWay);
00070   }
00071 };
00072 template <> struct hash<std::pair<_cityGraphPoint, _cityGraphNeighbor» {
00073   std::size_t operator()(const std::pair<_cityGraphPoint, _cityGraphNeighbor> &pair) const {
00074     return std::hash<_cityGraphPoint>()(pair.first) ^ std::hash<_cityGraphNeighbor>()(pair.second);
00075   }
00076 };
00077 } // namespace std
00078
00085 class CityGraph {
00086 public:
00087   using point = _cityGraphPoint;
00088   using neighbor = _cityGraphNeighbor;
00089
00097   void createGraph(const CityMap &cityMap);
00098
00103   std::unordered_map<point, std::vector<neighbor» getNeighbors() const { return neighbors; }
00104
00109   std::unordered_set<point> getGraphPoints() const { return graphPoints; }
00110
00115   point getRandomPoint() const;
00116
00121   double getHeight() const { return height; }
00122
00127   double getWidth() const { return width; }
00128
00135   DubinsInterpolator *getInterpolator(const point &point1, const neighbor &point2) {
00136     std::pair<point, neighbor> key = {point1, point2};
00137     if (interpolators.find(key) != interpolators.end()) {
00138       return interpolators[key];
00139     }
00140     return nullptr;
```

```
00141   }
00142
00143 private:
00144   std::unordered_map<point, std::vector<neighbor» neighbors;
00145   std::unordered_set<point> graphPoints;
00146
00147   std::unordered_map<std::pair<point, neighbor>, DubinsInterpolator *> interpolators;
00148
00149   void linkPoints(const point &point1, const point &point2, int direction,
00150                   bool subPoints); // direction: 0 -> point1 to point2, 1 -> point2 to point1, 2 ->
      both
00151   bool canLink(const point &point1, const point &point2, double speed, double *distance) const;
00152
00153   double width;
00154   double height;
00155 };
```

7.7 cityMap.h File Reference

City map class definition.

```
#include "config.h"
#include <SFML/Graphics.hpp>
#include <math.h>
#include <string>
#include <tinyxml2.h>
#include <vector>
```

Classes

- struct _cityMapSegment

  A segment in the city map.
- struct _cityMapRoad

  A road in the city map.
- struct _cityMapBuilding

  A building in the city map.
- struct _cityMapGreenArea

  A green area in the city map.
- struct _cityMapWaterArea

  A water area in the city map.
- struct _cityMapIntersection

  An intersection in the city map.
- class CityMap

  A city map.

7.7.1 Detailed Description

City map class definition.

This file contains the definition of the CityMap class, which represents a city map.

Definition in file cityMap.h.

7.8 cityMap.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "config.h"
00011 #include <SFML/Graphics.hpp>
00012 #include <math.h>
00013 #include <string>
00014 #include <tinyxml2.h>
00015 #include <vector>
00016
00021 typedef struct {
00022   sf::Vector2f p1;
00023   sf::Vector2f p2;
00024   sf::Vector2f p1_offset;
00025   sf::Vector2f p2_offset;
00026   sf::Angle angle;
00027 } _cityMapSegment;
00028
00033 typedef struct {
00034   int id;
00035   std::vector<_cityMapSegment> segments;
00036   double width;
00037   int numLanes;
00038 } _cityMapRoad;
00039
00044 typedef struct {
00045   std::vector<sf::Vector2f> points;
00046 } _cityMapBuilding;
00047
00052 typedef struct {
00053   std::vector<sf::Vector2f> points;
00054   int type;
00055 } _cityMapGreenArea;
00056
00061 typedef struct {
00062   std::vector<sf::Vector2f> points;
00063 } _cityMapWaterArea;
00064
00069 typedef struct {
00070   int id;
00071   sf::Vector2f center;
00072   double radius;
00073   std::vector<std::pair<int, int» roadSegmentIds;
00075 } _cityMapIntersection;
00076
00084 class CityMap {
00085 public:
00086   using segment = _cityMapSegment;
00087   using road = _cityMapRoad;
00088   using building = _cityMapBuilding;
00089   using greenArea = _cityMapGreenArea;
00090   using waterArea = _cityMapWaterArea;
00091   using intersection = _cityMapIntersection;
00092
00096   CityMap();
00097
00102   void loadFile(const std::string &filename);
00103
00108   bool isCityMapLoaded() const { return isLoaded; }
00109
00114   std::vector<road> getRoads() const { return roads; }
00115
00120   std::vector<intersection> getIntersections() const { return intersections; }
00121
00126   std::vector<building> getBuildings() const { return buildings; }
00127
00132   std::vector<greenArea> getGreenAreas() const { return greenAreas; }
00133
00138   std::vector<waterArea> getWaterAreas() const { return waterAreas; }
00139
00144   sf::Vector2f getMinLatLon() const { return minLatLon; }
00145
00150   sf::Vector2f getMaxLatLon() const { return maxLatLon; }
00151
00156   int getWidth() const { return width; }
00157
00162   int getHeight() const { return height; }
00163
00164 private:
00165   bool isLoaded = false;
00166
00167   std::vector<road> roads;
```

```
00168   std::vector<intersection> intersections;
00169   std::vector<building> buildings;
00170   std::vector<greenArea> greenAreas;
00171   std::vector<waterArea> waterAreas;
00172
00173   sf::Vector2f minLatLon;
00174   sf::Vector2f maxLatLon;
00175   double width;  // in meters
00176   double height; // in meters
00177 };
```

7.9 config.h File Reference

Configuration file.

```
#include <string>
```

Variables

- constexpr int ENVIRONMENT = 0
- constexpr int SCREEN_WIDTH = 2880
- constexpr int SCREEN_HEIGHT = 1864
- constexpr double LOG_CBS_REFRESHRATE = 0.3
- constexpr int EARTH_RADIUS = 6371000
- constexpr double DEFAULT_ROAD_WIDTH = 7.0
- constexpr double DEFAULT_LANE_WIDTH = 3.5
- constexpr double MIN_ROAD_WIDTH = 4.0
- constexpr bool ROAD_ENABLE_RIGHT_HAND_TRAFFIC = false
- constexpr double DUBINS_INTERPOLATION_STEP = 0.1
- constexpr double ZOOM_SPEED = 0.1
- constexpr double MOVE_SPEED = 0.01
- constexpr double SIM_STEP_TIME = 0.05
- constexpr int CBS_PRECISION_FACTOR = 1
- constexpr double CBS_MAX_SUB_TIME = 30
- constexpr double CBS_MAX_OPENSET_SIZE = 5
- constexpr double OCBS_CONFLICT_RANGE = SIM_STEP_TIME ∗ 5
- constexpr double CELL_SIZE = 0.1
- constexpr double SPEED_RESOLUTION = 0.3
- constexpr double ANGLE_RESOLUTION = 0.1
- constexpr double TIME_RESOLUTION = SIM_STEP_TIME
- constexpr double CAR_MIN_TURNING_RADIUS = 1.5
- constexpr double CAR_MAX_SPEED_KM = 30.0
- constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6
- constexpr double CAR_MAX_G_FORCE = 0.5
- constexpr double CAR_ACCELERATION = 1
- constexpr double CAR_DECELERATION = 1
- constexpr double CAR_LENGTH = 4.2
- constexpr double CAR_WIDTH = 1.6

7.9.1 Detailed Description

Configuration file.

Definition in file config.h.

7.9.2 Variable Documentation

7.9.2.1 ANGLE_RESOLUTION

```
constexpr double ANGLE_RESOLUTION = 0.1  [constexpr]
```

Definition at line 36 of file config.h.

7.9.2.2 CAR_ACCELERATION

```
constexpr double CAR_ACCELERATION = 1  [constexpr]
```

Definition at line 43 of file config.h.

7.9.2.3 CAR_DECELERATION

```
constexpr double CAR_DECELERATION = 1  [constexpr]
```

Definition at line 44 of file config.h.

7.9.2.4 CAR_LENGTH

```
constexpr double CAR_LENGTH = 4.2  [constexpr]
```

Definition at line 45 of file config.h.

7.9.2.5 CAR_MAX_G_FORCE

```
constexpr double CAR_MAX_G_FORCE = 0.5  [constexpr]
```

Definition at line 42 of file config.h.

7.9.2.6 CAR_MAX_SPEED_KM

```
constexpr double CAR_MAX_SPEED_KM = 30.0  [constexpr]
```

Definition at line 40 of file config.h.

7.9.2.7 CAR_MAX_SPEED_MS

```
constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6  [constexpr]
```

Definition at line 41 of file config.h.

7.9.2.8 CAR_MIN_TURNING_RADIUS

```
constexpr double CAR_MIN_TURNING_RADIUS = 1.5  [constexpr]
```

Definition at line 39 of file config.h.

7.9.2.9 CAR_WIDTH

```
constexpr double CAR_WIDTH = 1.6  [constexpr]
```

Definition at line 46 of file config.h.

7.9.2.10 CBS_MAX_OPENSET_SIZE

```
constexpr double CBS_MAX_OPENSET_SIZE = 5  [constexpr]
```

Definition at line 29 of file config.h.

7.9.2.11 CBS_MAX_SUB_TIME

```
constexpr double CBS_MAX_SUB_TIME = 30  [constexpr]
```

Definition at line 28 of file config.h.

7.9.2.12 CBS_PRECISION_FACTOR

```
constexpr int CBS_PRECISION_FACTOR = 1  [constexpr]
```

Definition at line 27 of file config.h.

7.9.2.13 CELL_SIZE

```
constexpr double CELL_SIZE = 0.1  [constexpr]
```

Definition at line 34 of file config.h.

7.9.2.14 DEFAULT_LANE_WIDTH

```
constexpr double DEFAULT_LANE_WIDTH = 3.5  [constexpr]
```

Definition at line 17 of file config.h.

7.9.2.15 DEFAULT_ROAD_WIDTH

```
constexpr double DEFAULT_ROAD_WIDTH = 7.0  [constexpr]
```

Definition at line 16 of file config.h.

7.9.2.16 DUBINS_INTERPOLATION_STEP

```
constexpr double DUBINS_INTERPOLATION_STEP = 0.1  [constexpr]
```

Definition at line 21 of file config.h.

7.9.2.17 EARTH_RADIUS

```
constexpr int EARTH_RADIUS = 6371000  [constexpr]
```

Definition at line 14 of file config.h.

7.9.2.18 ENVIRONMENT

```
constexpr int ENVIRONMENT = 0  [constexpr]
```

Definition at line 9 of file config.h.

7.9.2.19 LOG_CBS_REFRESHRATE

```
constexpr double LOG_CBS_REFRESHRATE = 0.3  [constexpr]
```

Definition at line 12 of file config.h.

7.9.2.20 MIN_ROAD_WIDTH

```
constexpr double MIN_ROAD_WIDTH = 4.0  [constexpr]
```

Definition at line 18 of file config.h.

7.9.2.21 MOVE_SPEED

```
constexpr double MOVE_SPEED = 0.01  [constexpr]
```

Definition at line 24 of file config.h.

7.9.2.22 OCBS_CONFLICT_RANGE

```
constexpr double OCBS_CONFLICT_RANGE = SIM_STEP_TIME * 5  [constexpr]
```

Definition at line 31 of file config.h.

7.9.2.23 ROAD_ENABLE_RIGHT_HAND_TRAFFIC

```
constexpr bool ROAD_ENABLE_RIGHT_HAND_TRAFFIC = false  [constexpr]
```

Definition at line 19 of file config.h.

7.9.2.24 SCREEN_HEIGHT

```
constexpr int SCREEN_HEIGHT = 1864  [constexpr]
```

Definition at line 11 of file config.h.

7.9.2.25 SCREEN_WIDTH

```
constexpr int SCREEN_WIDTH = 2880  [constexpr]
```

Definition at line 10 of file config.h.

7.9.2.26 SIM_STEP_TIME

```
constexpr double SIM_STEP_TIME = 0.05  [constexpr]
```

Definition at line 26 of file config.h.

7.9.2.27 SPEED_RESOLUTION

```
constexpr double SPEED_RESOLUTION = 0.3  [constexpr]
```

Definition at line 35 of file config.h.

7.9.2.28 TIME_RESOLUTION

```
constexpr double TIME_RESOLUTION = SIM_STEP_TIME  [constexpr]
```

Definition at line 37 of file config.h.

7.9.2.29 ZOOM_SPEED

```
constexpr double ZOOM_SPEED = 0.1  [constexpr]
```

Definition at line 23 of file config.h.

7.10 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_REFRESHRATE = 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 DUBINS_INTERPOLATION_STEP = 0.1; // in meters (useless to be more precise than the
      cell size)
00022
00023 constexpr double ZOOM_SPEED = 0.1;
00024 constexpr double MOVE_SPEED = 0.01;
00025
00026 constexpr double SIM_STEP_TIME = 0.05;  // in seconds
00027 constexpr int CBS_PRECISION_FACTOR = 1; // CBS_PRECISION_FACTOR * SIM_STEP_TIME must not be to high
00028 constexpr double CBS_MAX_SUB_TIME = 30; // in seconds
00029 constexpr double CBS_MAX_OPENSET_SIZE = 5;
00030
00031 constexpr double OCBS_CONFLICT_RANGE = SIM_STEP_TIME * 5;
00032
00033 // For hash functions (to reduce items that are really close to each other)
00034 constexpr double CELL_SIZE = 0.1;                 // in meters
00035 constexpr double SPEED_RESOLUTION = 0.3;          // in m/s
00036 constexpr double ANGLE_RESOLUTION = 0.1;          // in radians
00037 constexpr double TIME_RESOLUTION = SIM_STEP_TIME; // in seconds
00038
00039 constexpr double CAR_MIN_TURNING_RADIUS = 1.5;              // in meters
00040 constexpr double CAR_MAX_SPEED_KM = 30.0;                   // in km/h
00041 constexpr double CAR_MAX_SPEED_MS = CAR_MAX_SPEED_KM / 3.6; // in m/s
00042 constexpr double CAR_MAX_G_FORCE = 0.5;                     // in m/s^2
00043 constexpr double CAR_ACCELERATION = 1;                      // in m/s^2
00044 constexpr double CAR_DECELERATION = 1;                      // in m/s^2
00045 constexpr double CAR_LENGTH = 4.2;                          // in meters
00046 constexpr double CAR_WIDTH = 1.6;                           // in meters
```

7.11 dataManager.h File Reference

Data manager.

```
#include <string>
#include <vector>
```

Classes

- struct _data

  Data structure.
- class DataManager

  Data manager.

7.11.1 Detailed Description

Data manager.

This file contains the data manager class.

Definition in file dataManager.h.

7.12 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 };
```

7.13 dubins.h File Reference

Dubins path.

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

Classes

- class DubinsInterpolator

7.13.1 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.

7.14 dubins.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "cityGraph.h"
00011 #include <vector>
00012
00013 class AStar;
00014
00015 class DubinsInterpolator {
00016 public:
00023   void init(_cityGraphPoint start, _cityGraphPoint end, double radius);
00024
00032   _cityGraphPoint get(double time, double startSpeed, double endSpeed);
00033
00040   double getDuration(double startSpeed, double endSpeed) { return 2 * distance / (startSpeed +
      endSpeed); }
00041
00046   double getDistance() { return distance; }
00047
00048 private:
00049   _cityGraphPoint startPoint;
00050   _cityGraphPoint endPoint;
00051   double distance;
00052   double radius;
00053   int numInterpolatedPoints;
00054
00055   // Points spaced by DUBINS_INTERPOLATION_STEP. The first point and the last point are always the
      start and end points.
00056   std::vector<_cityGraphPoint> interpolatedCurve;
00057 };
```

7.15 fileSelector.h File Reference

File selector.

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

Classes

- class FileSelector

  A file selector.

7.15.1 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.

7.16 fileSelector.h

Go to the documentation of this file.

```
00001
00007 #pragma once
00008
00009 #include <iostream>
00010 #include <termios.h>
00011 #include <unistd.h>
00012 #include <vector>
00013
00020 class FileSelector {
00021 private:
00022   std::string folderPath;
00023   std::vector<std::string> files;
00024   int selectedIndex;
00026   void loadFiles();
00027   char getKeyPress();
00028   void moveCursorUp();
00029   void moveCursorDown();
00030   void displayFiles();
00032 public:
00033   FileSelector(const std::string &path) : folderPath(path), selectedIndex(0) { loadFiles(); }
00034   ~FileSelector() { std::cout « "\033[?25h"; }
00035
00036   std::string selectFile();
00037 };
```

7.17 manager.h File Reference

Manager for the cars.

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

Classes

- class Manager

  A manager for the cars.

7.17.1 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.

7.18 manager.h

Go to the documentation of this file.

```
00001
00008 #pragma once
00009
00010 #include "car.h"
00011 #include "cityGraph.h"
00012 #include <SFML/Graphics.hpp>
00013 #include <spdlog/spdlog.h>
00014 #include <vector>
00015
00023 class Manager {
00024 public:
00030   Manager(const CityGraph &cityGraph, const CityMap &CityMap) : graph(cityGraph), map(CityMap) {}
00031
00036   virtual void initializeAgents(int numAgents);
00037
00044   virtual void planPaths() = 0;
00045
00049   virtual void updateAgents();
00050
00056   virtual void userInput(sf::Event event, sf::RenderWindow &window) {};
00057
00062   virtual void renderAgents(sf::RenderWindow &window) final;
00063
00068   virtual int getNumAgents() { return numCars; }
00069
00074   virtual std::vector<Car> getCars() { return cars; }
00075
00076 protected:
00077   int numCars;
00078   std::vector<Car> cars;
00079   CityGraph graph;
00080   CityMap map;
00081 };
```

7.19 manager_ocbs.h File Reference

Manager for the CBS algorithm.

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

Classes

- struct _managerOCBSConflictSituation
- struct _managerOCBSConflict
- struct std::hash< _managerOCBSConflictSituation >
- struct std::hash< _managerOCBSConflict >
- struct _managerOCBSNode
- class ManagerOCBS

  Manager for the CBS algorithm This class is responsible for managing the agents and their paths using the Conflict-Based Search (CBS) algorithm. It inherits from the Manager class and implements the pathfinding logic specific to the CBS algorithm. This class initializes paths for agents, handles user input, and plans paths using the CBS algorithm.

Namespaces

- namespace std

Typedefs

- typedef struct _managerOCBSConflictSituation _managerOCBSConflictSituation
- typedef struct _managerOCBSConflict _managerOCBSConflict
- typedef struct _managerOCBSNode _managerOCBSNode

7.19.1 Detailed Description

Manager for the CBS algorithm.

Definition in file manager_ocbs.h.

7.19.2 Typedef Documentation

7.19.2.1 _managerOCBSConflict

```
typedef struct _managerOCBSConflict _managerOCBSConflict
```

7.19.2.2 _managerOCBSConflictSituation

```
typedef struct _managerOCBSConflictSituation _managerOCBSConflictSituation
```

7.19.2.3 _managerOCBSNode

```
typedef struct _managerOCBSNode _managerOCBSNode
```

7.20 manager_ocbs.h

Go to the documentation of this file.

```
00001
00005 #pragma once
00006
00007 #include "cityGraph.h"
00008 #include "manager.h"
00009 #include <SFML/Graphics.hpp>
00010 #include <vector>
00011
00012 typedef struct _managerOCBSConflictSituation {
00013   int car;
00014   sf::Vector2f at;
00015   double time;
00016
00017   bool operator==(const _managerOCBSConflictSituation &other) const {
00018     int t = std::round(time / OCBS_CONFLICT_RANGE);
00019     int oT = std::round(other.time / OCBS_CONFLICT_RANGE);
00020     int x = std::round(at.x / CELL_SIZE);
00021     int oX = std::round(other.at.x / CELL_SIZE);
00022     int y = std::round(at.y / CELL_SIZE);
00023     int oY = std::round(other.at.y / CELL_SIZE);
00024     return car == other.car && t == oT && x == oX && y == oY;
00025   }
00026 } _managerOCBSConflictSituation;
00027
00028 typedef struct _managerOCBSConflict {
00029   int car;
00030   int withCar;
00031   double time;
```

```
00032   sf::Vector2f position;
00033
00034   bool operator==(const _managerOCBSConflict &other) const {
00035     return car == other.car && withCar == other.withCar && time == other.time;
00036   }
00037 } _managerOCBSConflict;
00038
00039 namespace std {
00040 template <> struct hash<_managerOCBSConflictSituation> {
00041   std::size_t operator()(const _managerOCBSConflictSituation &point) const {
00042     int t = std::round(point.time / OCBS_CONFLICT_RANGE);
00043     int x = std::round(point.at.x / CAR_LENGTH);
00044     int y = std::round(point.at.y / CAR_LENGTH);
00045     return std::hash<int>()(point.car) ^ std::hash<int>()(t) ^ std::hash<int>()(x) ^
      std::hash<int>()(y);
00046   }
00047 };
00048 template <> struct hash<_managerOCBSConflict> {
00049   std::size_t operator()(const _managerOCBSConflict &point) const {
00050     return std::hash<int>()(point.car) ^ std::hash<int>()(point.withCar) ^
      std::hash<double>()(point.time) ^
00051            std::hash<float>()(point.position.x) ^ std::hash<float>()(point.position.y);
00052   }
00053 };
00054 } // namespace std
00055
00056 typedef struct _managerOCBSNode {
00057   std::vector<std::vector<sf::Vector2f» paths;
00058   std::vector<double> costs;
00059   double cost;
00060   int depth;
00061   bool hasResolved;
00062   // std::unordered_multimap<_managerOCBSConflictSituation, std::unordered_set<_managerOCBSConflict»
00063   //     conflicts; /**< \brief The conflicts for all agents */
00064
00065   bool operator<(const _managerOCBSNode &other) const {
00066     return cost > other.cost || (cost == other.cost && depth > other.depth);
00067   }
00068 } _managerOCBSNode;
00069
00077 class ManagerOCBS : public Manager {
00078 public:
00079   using ConflictSituation = _managerOCBSConflictSituation;
00080   using Conflict = _managerOCBSConflict;
00081   using Node = _managerOCBSNode;
00082
00088   ManagerOCBS(const CityGraph &cityGraph, const CityMap &cityMap) : Manager(cityGraph, cityMap) {}
00089
00094   void initializePaths(Node *node);
00095
00099   void userInput(sf::Event event, sf::RenderWindow &window) override;
00100
00107   void planPaths() override;
00108
00109 private:
00110   bool findConflict(int *car1, int *car2, int *time, Node *node);
00111   bool findPaths();
00112   void pathfinding(Node *node, int carIndex);
00113
00114   std::vector<_cityGraphPoint> starts;
00115   std::vector<_cityGraphPoint> ends;
00116   std::vector<double> baseCosts;
00117   std::priority_queue<_managerOCBSNode> openSet;
00118   std::unordered_map<_managerOCBSConflictSituation, std::unordered_set<_managerOCBSConflict> *>
00119       conflicts;
00120 };
```

7.21 renderer.h File Reference

A renderer for the city.

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

Classes

- class Renderer

  A renderer for the city.

Functions

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

  Draw an arrow.

7.21.1 Detailed Description

A renderer for the city.

Definition in file renderer.h.

7.21.2 Function Documentation

7.21.2.1 drawArrow()

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

Draw an arrow.

Parameters

window	The window
position	The position
rotation	The rotation
length	The length
thickness	The thickness
color	The color
outline	If the arrow should have an outline

Definition at line 67 of file renderer.h.

```
                                                                              {
    sf::ConvexShape arrow;

    arrow.setFillColor(color);
    arrow.setOrigin({-(float)length / 2, 0});
    arrow.setPosition(position);
    arrow.setRotation(rotation);

    arrow.setPointCount(7);
    arrow.setPoint(0, sf::Vector2f(0, 0));
```

```
00078   arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
00079   arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
00080   arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
00081   arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
00082   arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
00083   arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));
00084
00085   if (outline) {
00086     arrow.setOutlineThickness(thickness / 10);
00087     arrow.setOutlineColor(sf::Color::Black);
00088   }
00089
00090   window.draw(arrow);
00091 }
```

7.22 renderer.h

Go to the documentation of this file.

```
00001
00005 #pragma once
00006
00007 #include "cityMap.h"
00008 #include "manager.h"
00009 #include <SFML/Graphics.hpp>
00010
00011 class CityGraph;
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
00050 private:
00051   sf::RenderWindow window;
00052   double time;
00053
00054   bool debug = false;
00055 };
00056
00067 inline void drawArrow(sf::RenderWindow &window, sf::Vector2f position, sf::Angle rotation, double
      length,
00068                       double thickness, sf::Color color = sf::Color::Red, bool outline = false) {
00069   sf::ConvexShape arrow;
00070
00071   arrow.setFillColor(color);
00072   arrow.setOrigin({-(float)length / 2, 0});
00073   arrow.setPosition(position);
00074   arrow.setRotation(rotation);
00075
00076   arrow.setPointCount(7);
00077   arrow.setPoint(0, sf::Vector2f(0, 0));
00078   arrow.setPoint(1, sf::Vector2f(-2 * length / 5, thickness));
00079   arrow.setPoint(2, sf::Vector2f(-2 * length / 5, thickness / 2));
00080   arrow.setPoint(3, sf::Vector2f(-length, thickness / 2));
00081   arrow.setPoint(4, sf::Vector2f(-length, -thickness / 2));
00082   arrow.setPoint(5, sf::Vector2f(-2 * length / 5, -thickness / 2));
00083   arrow.setPoint(6, sf::Vector2f(-2 * length / 5, -thickness));
00084
00085   if (outline) {
00086     arrow.setOutlineThickness(thickness / 10);
00087     arrow.setOutlineColor(sf::Color::Black);
00088   }
00089
00090   window.draw(arrow);
00091 }
```

7.23 test.h File Reference

A header file for the Test class.

Classes

- class Test

  A class for testing the project.

7.23.1 Detailed Description

A header file for the Test class.

Definition in file test.h.

7.24 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 };
```

7.25 utils.h File Reference

Utility functions.

```
#include "car.h"
#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 turningRadius (double speed)

  Get the turning radius from the speed.
- double turningRadiusToSpeed (double radius)

  Get the speed from the turning radius.
- sf::Font loadFont ()

  Load a font.

7.25.1 Detailed Description

Utility functions.

Definition in file utils.h.

7.25.2 Function Documentation

7.25.2.1 distance()

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

Get the distance between two points.

Parameters

p1	The first point
p2	The second point

Definition at line 29 of file utils.h.

```
00029                                                                {
00030   return std::sqrt(std::pow(p2.x - p1.x, 2) + std::pow(p2.y - p1.y, 2));
00031 }
```

7.25.2.2 latLonToXY()

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

Convert latitude and longitude to x and y.

Parameters

lat	The latitude
lon	The longitude

Returns

The x and y

Definition at line 17 of file utils.h.

```
00017                                                                {
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 }
```

7.25.2.3 loadFont()

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

Load a font.

Returns

The font

Definition at line 11 of file utils.cpp.

```
00011                    {
00012   if (!fontLoaded) {
00013     if (!font.openFromFile("assets/fonts/arial.ttf")) {
00014       spdlog::error("Failed to load font");
00015     }
00016     fontLoaded = true;
00017   }
00018   return font;
00019 }
```

7.25.2.4 turningRadius()

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

Get the turning radius from the speed.

Parameters

speed	The speed

Returns

The turning radius

Definition at line 38 of file utils.h.

```
00038 { return speed * speed / CAR_MAX_G_FORCE; }
```

7.25.2.5 turningRadiusToSpeed()

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

Get the speed from the turning radius.

Parameters

radius	The turning radius

Returns

The speed

Definition at line 45 of file utils.h.

```
00045 { return std::sqrt(radius * CAR_MAX_G_FORCE); }
```

7.26 utils.h

Go to the documentation of this file.

```
00001
00005 #pragma once
00006
00007 #include "car.h"
00008 #include "config.h"
00009 #include <SFML/Graphics.hpp>
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
00038 inline double turningRadius(double speed) { return speed * speed / CAR_MAX_G_FORCE; }
00039
00045 inline double turningRadiusToSpeed(double radius) { return std::sqrt(radius * CAR_MAX_G_FORCE); }
00046
00051 sf::Font loadFont();
```

7.27 aStar.cpp File Reference

A∗ algorithm implementation.

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

7.27.1 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.

7.28 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 <spdlog/spdlog.h>
00014 #include <unordered_set>
00015
00016 AStar::AStar(CityGraph::point start, CityGraph::point end, const CityGraph &cityGraph) {
00017   this->start.point = start;
00018   this->start.speed = 0;
00019   this->end.point = end;
00020   this->end.speed = 0;
00021   this->graph = cityGraph;
00022 }
00023
00024 void AStar::process() {
00025   std::unordered_map<AStar::node, AStar::node> cameFrom;
00026   std::unordered_map<AStar::node, double> gScore;
00027   std::unordered_map<AStar::node, double> fScore;
00028
00029   auto heuristic = [&](const AStar::node &a) {
00030     sf::Vector2f diff = end.point.position - a.point.position;
00031     double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00032     return distance / CAR_MAX_SPEED_MS;
00033   };
00034   auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };
00035
00036   std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSetAstar(compare);
00037   std::unordered_set<AStar::node> isInOpenSet;
00038
00039   openSetAstar.push(start);
00040   gScore[start] = 0;
00041   fScore[start] = heuristic(start);
00042
00043   auto neighbors = graph.getNeighbors();
00044
00045   int nbIterations = 0;
00046   while (!openSetAstar.empty() && nbIterations++ < 1e5) {
00047     AStar::node current = openSetAstar.top();
00048     openSetAstar.pop();
00049     isInOpenSet.erase(current);
00050
00051     if (current.point == end.point) {
00052       AStar::node currentCopy = current;
00053       path.clear();
00054
00055       while (!(currentCopy == start)) {
00056         path.push_back(currentCopy);
00057         currentCopy = cameFrom[currentCopy];
00058       }
00059
00060       path.push_back(currentCopy);
00061       std::reverse(path.begin(), path.end());
00062       processed = true;
00063     }
00064
00065     for (const auto &neighborGraphPoint : neighbors[current.point]) {
00066       if (current.speed > neighborGraphPoint.maxSpeed)
00067         continue;
00068
00069       if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
00070         continue;
00071
00072       std::vector<double> newSpeeds;
00073       newSpeeds.push_back(current.speed);
00074
00075       double distance = graph.getInterpolator(current.point, neighborGraphPoint)->getDistance();
00076       double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
00077       double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);
00078
00079       auto push = [&](double nSpeed) {
00080         int numSpeedDiv = 5;
00081         for (int i = 1; i < numSpeedDiv + 1; i++) {
00082           double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
00083           if (s < SPEED_RESOLUTION)
00084             continue;
00085           newSpeeds.push_back(s);
00086         }
00087       };
00088
```

```
00089       if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
00090         push(neighborGraphPoint.maxSpeed);
00091       } else if (nSpeedAcc < neighborGraphPoint.maxSpeed) {
00092         push(nSpeedAcc);
00093       }
00094
00095       if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) { // check if nSpeedDec is finite and
      not NaN
00096         if (nSpeedDec < 0 && current.speed > 0) {
00097           push(0);
00098         } else if (nSpeedDec >= 0) {
00099           push(nSpeedDec);
00100         }
00101       }
00102
00103       AStar::node neighbor;
00104       neighbor.point = neighborGraphPoint.point;
00105       neighbor.arcFrom = {current.point, neighborGraphPoint};
00106       if (distance == 0) {
00107         neighbor.speed = current.speed;
00108         if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
00109           cameFrom[neighbor] = current;
00110           gScore[neighbor] = gScore[current];
00111           fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00112
00113           if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00114             openSetAstar.push(neighbor);
00115
00116             isInOpenSet.insert(neighbor);
00117           }
00118         }
00119         continue;
00120       }
00121
00122       for (const auto &newSpeed : newSpeeds) {
00123         if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
00124           continue;
00125
00126         if (newSpeed == current.speed && newSpeed == 0)
00127           continue;
00128
00129         neighbor.speed = newSpeed;
00130
00131         double duration = 2 * distance / (current.speed + newSpeed);
00132         double tentativeGScore = gScore[current] + duration;
00133
00134         double t = gScore[current];
00135
00136         if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
00137           cameFrom[neighbor] = current;
00138           gScore[neighbor] = tentativeGScore;
00139           fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00140
00141           if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00142             openSetAstar.push(neighbor);
00143             isInOpenSet.insert(neighbor);
00144           }
00145         }
00146       }
00147     }
00148   }
00149 }
```

7.29 car.cpp File Reference

Car class implementation.

```
#include "car.h"
#include "config.h"
#include "utils.h"
#include <SFML/Graphics/Text.hpp>
#include <SFML/System/Angle.hpp>
#include <spdlog/spdlog.h>
```

7.29.1 Detailed Description

Car class implementation.

This file contains the implementation of the Car class.

Definition in file car.cpp.

7.30 car.cpp

Go to the documentation of this file.

```
00001
00007 #include "car.h"
00008 #include "config.h"
00009 #include "utils.h"
00010 #include <SFML/Graphics/Text.hpp>
00011 #include <SFML/System/Angle.hpp>
00012 #include <spdlog/spdlog.h>
00013
00014 Car::Car() {
00015   std::vector<sf::Color> colors = {sf::Color(50, 120, 190), sf::Color(183, 132, 144), sf::Color(105,
      101, 89),
00016                                    sf::Color(182, 18, 34),  sf::Color(24, 25, 24),    sf::Color(17,
      86, 122)};
00017   color = colors[rand() % colors.size()];
00018 }
00019
00020 void Car::move() {
00021   if (currentPoint >= (int)path.size())
00022     return;
00023
00024   currentPoint++;
00025 }
00026
00027 void Car::render(sf::RenderWindow &window) {
00028   if (1 + currentPoint >= (int)path.size())
00029     return;
00030
00031   sf::Vector2f point = path[currentPoint];
00032   sf::Vector2f nextPoint = path[currentPoint + 1];
00033   sf::Vector2f diff = nextPoint - point;
00034   double length = sqrt(diff.x * diff.x + diff.y * diff.y);
00035   int fact = 1;
00036
00037   while (point == nextPoint && currentPoint + fact < (int)path.size()) {
00038     fact++;
00039     nextPoint = path[currentPoint + fact];
00040     diff = nextPoint - point;
00041     length = sqrt(diff.x * diff.x + diff.y * diff.y);
00042   }
00043
00044   sf::RectangleShape shape(sf::Vector2f(CAR_LENGTH, CAR_WIDTH));
00045   shape.setOrigin({CAR_LENGTH / 2.0f, CAR_WIDTH / 2.0f});
00046   shape.setPosition(point);
00047   shape.setRotation(sf::radians(atan2(nextPoint.y - point.y, nextPoint.x - point.x)));
00048   if (debug)
00049     shape.setFillColor(sf::Color(255, 0, 0));
00050   else
00051     shape.setFillColor(color);
00052   window.draw(shape);
00053
00054   if (!debug)
00055     return;
00056
00057   // Render speed, elapsed time, remaining time, and distance
00058   double speed = 3.6f * length / (fact * SIM_STEP_TIME);
00059   int iSpeed = speed;
00060   int dSpeed = (double)(speed - iSpeed) * 100.0;
00061   sf::Font font = loadFont();
00062   sf::Text text(font);
00063   text.setCharacterSize(24);
00064   text.setFillColor(sf::Color::White);
00065   text.setPosition(getPosition());
00066   text.setString(std::to_string(iSpeed) + "." + std::to_string(dSpeed) + " km/h" + "\n" +
00067                  std::to_string((int)getElapsedTime()) + "s / " +
      std::to_string((int)getRemainingTime()) + "s" + "\n" +
```

```
00068                 std::to_string((int)getElapsedDistance()) + "m / " +
      std::to_string((int)getRemainingDistance()) +
00069                 "m");
00070   text.setOutlineColor(sf::Color::Black);
00071   text.setOutlineThickness(1.0f);
00072   text.scale({0.1f, 0.1f});
00073   text.setOrigin({text.getLocalBounds().position.x / 2.0f, text.getLocalBounds().position.y / 2.0f});
00074   window.draw(text);
00075
00076   // Render path
00077   for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00078     sf::Vertex line[] = {{path[i]}, {path[i + 1]}};
00079     line[0].color = sf::Color(255, 255, 255);
00080     line[1].color = sf::Color(255, 255, 255);
00081     window.draw(line, 2, sf::PrimitiveType::Lines);
00082   }
00083 }
00084
00085 void Car::assignPath(std::vector<AStar::node> path, CityGraph &graph) {
00086   this->path.clear();
00087   this->aStarPath = path;
00088   currentPoint = 0;
00089
00090   double index = 0;
00091   double t = 0;
00092   double prevTime = 0;
00093
00094   for (int i = 1; i < (int)path.size(); i++) {
00095     AStar::node prevNode = path[i - 1];
00096     AStar::node node = path[i];
00097
00098     CityGraph::point start = node.arcFrom.first;
00099     CityGraph::neighbor end = node.arcFrom.second;
00100
00101     DubinsInterpolator *interpolator = graph.getInterpolator(start, end);
00102
00103     double duration = interpolator->getDuration(prevNode.speed, node.speed);
00104
00105     while (t < prevTime + duration) {
00106       double tt = t - prevTime;
00107       CityGraph::point p = interpolator->get(tt, prevNode.speed, node.speed);
00108
00109       this->path.push_back(p.position);
00110       t += SIM_STEP_TIME;
00111     }
00112     prevTime += duration;
00113   }
00114 }
00115
00116 void Car::assignExistingPath(std::vector<sf::Vector2f> path) {
00117   this->path = path;
00118   currentPoint = 0;
00119 }
00120
00121 double Car::getSpeed() {
00122   if (currentPoint >= (int)path.size() - 1)
00123     return 0;
00124
00125   sf::Vector2f diff = path[currentPoint + 1] - path[currentPoint];
00126   return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00127 }
00128
00129 double Car::getSpeedAt(int index) {
00130   if (index >= (int)path.size() - 1)
00131     return 0;
00132
00133   sf::Vector2f diff = path[index + 1] - path[index];
00134   return sqrt(diff.x * diff.x + diff.y * diff.y) / SIM_STEP_TIME;
00135 }
00136
00137 double Car::getRemainingTime() { return (double)(path.size() - currentPoint) * SIM_STEP_TIME; }
00138 double Car::getElapsedTime() { return (double)currentPoint * SIM_STEP_TIME; }
00139 double Car::getPathTime() { return (double)path.size() * SIM_STEP_TIME; }
00140
00141 double Car::getRemainingDistance() {
00142   double dist = 0;
00143   for (int i = currentPoint; i < (int)path.size() - 1; i++) {
00144     sf::Vector2f diff = path[i + 1] - path[i];
00145     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00146   }
00147
00148   return dist;
00149 }
00150
00151 double Car::getElapsedDistance() {
00152   double dist = 0;
00153   for (int i = 0; i < currentPoint; i++) {
```

```
00154     sf::Vector2f diff = path[i + 1] - path[i];
00155     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00156   }
00157
00158   return dist;
00159 }
00160
00161 double Car::getPathLength() {
00162   double dist = 0;
00163   for (int i = 0; i < (int)path.size() - 1; i++) {
00164     sf::Vector2f diff = path[i + 1] - path[i];
00165     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00166   }
00167
00168   return dist;
00169 }
00170
00171 void Car::chooseRandomStartEndPath(CityGraph &graph, CityMap &cityMap) {
00172   CityGraph::point start;
00173   CityGraph::point end;
00174
00175   double minDistance = std::max(graph.getWidth(), graph.getHeight()) / 2.0;
00176   std::vector<AStar::node> path;
00177
00178   do {
00179     path.clear();
00180     start = graph.getRandomPoint();
00181     end = graph.getRandomPoint();
00182
00183     if (std::sqrt(std::pow(start.position.x - end.position.x, 2) + std::pow(start.position.y -
      end.position.y, 2)) <
00184         minDistance)
00185       continue;
00186
00187     AStar aStar(start, end, graph);
00188     path = aStar.findPath();
00189
00190     if (!path.empty() && (int)path.size() >= 3) {
00191       AStar aStar(start, end, graph);
00192       path.clear();
00193       path = aStar.findPath();
00194     }
00195   } while (path.empty() || (int)path.size() < 3);
00196
00197   this->assignStartEnd(start, end);
00198   this->assignPath(path, graph);
00199 }
00200
00201 double Car::getAverageSpeed(CityGraph &graph) {
00202   double dist = 0;
00203   double time = 0;
00204   auto outOfBounds = [&](sf::Vector2f p) {
00205     return p.x < 0 || p.y < 0 || p.x > graph.getWidth() || p.y > graph.getWidth();
00206   };
00207
00208   for (int i = 0; i < (int)path.size() - 1; i++) {
00209     if (outOfBounds(path[i]) || outOfBounds(path[i + 1]))
00210       continue;
00211
00212     sf::Vector2f diff = path[i + 1] - path[i];
00213     dist += sqrt(diff.x * diff.x + diff.y * diff.y);
00214     time += SIM_STEP_TIME;
00215   }
00216
00217   if (time == 0)
00218     return 0;
00219
00220   return dist / time;
00221 }
```

7.31 cityGraph.cpp File Reference

City graph implementation.

```
#include "cityGraph.h"
#include "utils.h"
#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>
```

7.31.1 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.

7.32 cityGraph.cpp

Go to the documentation of this file.

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

```
00054       }
00055       numSeg++;
00056
00057       double segmentLength =
00058           sqrt(pow(segment.p2_offset.x - segment.p1_offset.x, 2) + pow(segment.p2_offset.y -
      segment.p1_offset.y, 2));
00059       double pointDistance = 15;
00060       int numPoints = segmentLength / pointDistance;
00061       double dx_s = (segment.p2_offset.x - segment.p1_offset.x) / numPoints;
00062       double dy_s = (segment.p2_offset.y - segment.p1_offset.y) / numPoints;
00063       double dx_a = sin(segment.angle.asRadians());
00064       double dy_a = -cos(segment.angle.asRadians());
00065
00066       if (dx_a < 0) {
00067         dx_a = -dx_a;
00068         dy_a = -dy_a;
00069       }
00070
00071       for (int i_lane = 0; i_lane < road.numLanes; i_lane++) {
00072         double offset = ((double)i_lane - (double)road.numLanes / 2.0f) * road.width / road.numLanes;
00073         offset += road.width / (2 * road.numLanes);
00074
00075         if (numPoints == 0) {
00076           point point1;
00077           point1.angle = segment.angle;
00078           point1.position = sf::Vector2f(segment.p1_offset.x + offset * dx_a, segment.p1_offset.y +
      offset * dy_a);
00079
00080           point point2;
00081           point2.angle = segment.angle;
00082           point2.position = sf::Vector2f(segment.p2_offset.x + offset * dx_a, segment.p2_offset.y +
      offset * dy_a);
00083
00084           linkPoints(point1, point2, 2, true);
00085           continue;
00086         }
00087
00088         for (int i = 0; i <= numPoints; i++) {
00089           point point1;
00090           point1.position = sf::Vector2f(segment.p1_offset.x + i * dx_s + offset * dx_a,
00091                                          segment.p1_offset.y + i * dy_s + offset * dy_a);
00092           point1.angle = segment.angle;
00093
00094           if (i > 0) {
00095             for (int i2_lane = 0; i2_lane < road.numLanes; i2_lane++) {
00096               double offset2 = ((double)i2_lane - (double)road.numLanes / 2.0f) * road.width /
      road.numLanes;
00097               offset2 += road.width / (2 * road.numLanes);
00098
00099               point point2;
00100               point2.position = sf::Vector2f(segment.p1_offset.x + (i - 1) * dx_s + offset2 * dx_a,
00101                                              segment.p1_offset.y + (i - 1) * dy_s + offset2 * dy_a);
00102               point2.angle = segment.angle;
00103
00104               int direction = 2;
00105               double a = atan2(dy_a, dx_a);
00106               if (offset == offset2 || (offset >= 0 && offset2 >= 0)) {
00107                 if (dy_s >= 0) {
00108                   direction = offset > 0 ? 0 : 1;
00109                 } else {
00110                   direction = offset > 0 ? 1 : 0;
00111                 }
00112                 linkPoints(point1, point2, direction, offset == offset2);
00113               } else {
00114                 if (!ROAD_ENABLE_RIGHT_HAND_TRAFFIC) {
00115                   linkPoints(point1, point2, 2, true);
00116                 }
00117               }
00118             }
00119           }
00120         }
00121       }
00122     }
00123   }
00124
00125   // Connect the intersections
00126   for (const auto &intersection : intersections) {
00127     for (const auto &roadSegmentId1 : intersection.roadSegmentIds) {
00128       for (const auto &roadSegmentId2 : intersection.roadSegmentIds) {
00129         const auto &road1 = roads[roadSegmentId1.first];
00130         const auto &road2 = roads[roadSegmentId2.first];
00131         const auto &segment1 = road1.segments[roadSegmentId1.second];
00132         const auto &segment2 = road2.segments[roadSegmentId2.second];
00133
00134         // Find the point of the segment2 closest to the intersection
00135         point point1;
00136         point1.angle = segment1.angle;
```

```
00137         point1.position = (distance(segment1.p1, intersection.center) < distance(segment1.p2,
      intersection.center))
00138                               ? segment1.p1_offset
00139                               : segment1.p2_offset;
00140
00141         point point2;
00142         point2.angle = segment2.angle;
00143         point2.position = (distance(segment2.p1, intersection.center) < distance(segment2.p2,
      intersection.center))
00144                               ? segment2.p1_offset
00145                               : segment2.p2_offset;
00146
00147         for (int iL_1 = 0; iL_1 < road1.numLanes; iL_1++) {
00148           double offset1 = ((double)iL_1 - (double)road1.numLanes / 2.0f) * road1.width /
      road1.numLanes;
00149           offset1 += road1.width / (2 * road1.numLanes);
00150
00151           for (int iL_2 = 0; iL_2 < road2.numLanes; iL_2++) {
00152             double offset2 = ((double)iL_2 - (double)road2.numLanes / 2.0f) * road2.width /
      road2.numLanes;
00153             offset2 += road2.width / (2 * road2.numLanes);
00154
00155             point point1_offset;
00156             point1_offset.angle = segment1.angle;
00157             point1_offset.position = sf::Vector2f(point1.position.x + offset1 *
      sin(segment1.angle.asRadians()),
00158                                                   point1.position.y + offset1 *
      -cos(segment1.angle.asRadians()));
00159
00160             point point2_offset;
00161             point2_offset.angle = segment2.angle;
00162             point2_offset.position = sf::Vector2f(point2.position.x + offset2 *
      sin(segment2.angle.asRadians()),
00163                                                   point2.position.y + offset2 *
      -cos(segment2.angle.asRadians()));
00164
00165             linkPoints(point1_offset, point2_offset, 2, true);
00166           }
00167         }
00168       }
00169     }
00170   }
00171
00172   spdlog::info("Graph created with {} points", graphPoints.size());
00173
00174   // Remove all the neighbors that need to turn too much
00175   for (auto &point : graphPoints) {
00176     std::vector<neighbor> newNeighbors;
00177     double distance;
00178     for (auto &neighbor : neighbors[point]) {
00179       double speed = turningRadiusToSpeed(CAR_MIN_TURNING_RADIUS);
00180       bool can = canLink(point, neighbor.point, speed, &distance);
00181
00182       if (!can)
00183         continue;
00184
00185       while (canLink(point, neighbor.point, speed + 0.1, &distance)) {
00186         speed += 0.1;
00187         if (speed >= CAR_MAX_SPEED_MS) {
00188           speed = CAR_MAX_SPEED_MS;
00189           break;
00190         }
00191       }
00192
00193       if (can) {
00194         neighbor.maxSpeed = speed - 0.1;
00195         neighbor.turningRadius = turningRadius(speed);
00196         newNeighbors.push_back(neighbor);
00197       }
00198     }
00199
00200     neighbors[point].clear();
00201     for (const auto &neighbor : newNeighbors) {
00202       neighbors[point].push_back(neighbor);
00203     }
00204   }
00205
00206   // Interpolate all the curves
00207   spdlog::info("Interpolating curves ...");
00208
00209   interpolators.clear();
00210
00211   for (auto &point : graphPoints) {
00212     for (const auto &neighbor : neighbors[point]) {
00213       std::pair<_cityGraphPoint, _cityGraphNeighbor> key = {point, neighbor};
00214       if (interpolators.find(key) == interpolators.end()) {
00215         interpolators[key] = new DubinsInterpolator();
```

```
00216         interpolators[key]->init(point, neighbor.point, neighbor.turningRadius);
00217       }
00218     }
00219   }
00220
00221   spdlog::info("Curves interpolated");
00222 }
00223
00224 void CityGraph::linkPoints(const point &p, const point &n, int direction, bool subPoints) {
00225   std::vector<sf::Angle> anglesPoint = {p.angle, p.angle + sf::radians(M_PI)};
00226   std::vector<sf::Angle> anglesNeighbor = {n.angle, n.angle + sf::radians(M_PI)};
00227
00228   point copyPoint = p;
00229   point copyNeighbor = n;
00230
00231   bool isRiP = direction == 2 || direction == 0;
00232   bool isRiN = direction == 2 || direction == 1;
00233   bool isStraight = direction != 2;
00234   isStraight &= (anglesPoint[0] == anglesNeighbor[0] || anglesPoint[0] == anglesNeighbor[1] ||
00235                  anglesPoint[1] == anglesNeighbor[0] || anglesPoint[1] == anglesNeighbor[1]);
00236   isStraight &= subPoints;
00237
00238   if (!isStraight) {
00239     for (const auto &anglePoint : anglesPoint) {
00240       for (const auto &angleNeighbor : anglesNeighbor) {
00241         copyPoint.angle = anglePoint;
00242         copyNeighbor.angle = angleNeighbor;
00243
00244         neighbors[copyPoint].push_back({copyNeighbor, 0, 0, isRiP}); // This fields will be updated
    later
00245         neighbors[copyNeighbor].push_back({copyPoint, 0, 0, isRiN});
00246
00247         graphPoints.insert(copyPoint);
00248         graphPoints.insert(copyNeighbor);
00249       }
00250     }
00251     return;
00252   }
00253
00254   // Link adding points in the middle
00255   double pointDistance = 3;
00256   double distance = std::sqrt(std::pow(n.position.x - p.position.x, 2) + std::pow(n.position.y -
    p.position.y, 2));
00257   int numPoints = distance / pointDistance;
00258   double dx = (n.position.x - p.position.x) / numPoints;
00259   double dy = (n.position.y - p.position.y) / numPoints;
00260
00261   for (const auto &anglePoint : anglesPoint) {
00262     for (const auto &angleNeighbor : anglesNeighbor) {
00263       point previousPoint = p;
00264       previousPoint.angle = anglePoint;
00265
00266       for (int i = 1; i <= numPoints; i++) {
00267         point newPoint;
00268         newPoint.position = sf::Vector2f(p.position.x + i * dx, p.position.y + i * dy);
00269         newPoint.angle = anglePoint;
00270
00271         neighbors[previousPoint].push_back({newPoint, 0, 0, isRiP}); // This fields will be updated
    later
00272         neighbors[newPoint].push_back({previousPoint, 0, 0, isRiN});
00273
00274         previousPoint = newPoint;
00275
00276         graphPoints.insert(newPoint);
00277       }
00278
00279       // Add the last point
00280       neighbors[previousPoint].push_back({n, 0, 0, isRiP}); // This fields will be updated later
00281     }
00282   }
00283 }
00284
00285 CityGraph::point CityGraph::getRandomPoint() const {
00286   std::vector<point> graphPointsOut;
00287   for (const auto &point : graphPoints) {
00288     if (point.position.x + CAR_LENGTH < 0 || point.position.x - CAR_LENGTH > width ||
00289         point.position.y + CAR_LENGTH < 0 || point.position.y - CAR_LENGTH > height)
00290       graphPointsOut.push_back(point);
00291   }
00292
00293   auto it = graphPointsOut.begin();
00294   std::random_device rd;
00295   std::mt19937 gen(rd());
00296   std::uniform_int_distribution<> dis(0, graphPointsOut.size() - 1);
00297
00298   std::advance(it, dis(gen));
00299
```

```
00300   return *it;
00301 }
00302
00303 bool CityGraph::canLink(const point &point1, const point &point2, double speed, double *distance)
      const {
00304   double radius = turningRadius(speed);
00305
00306   ob::DubinsStateSpace space(radius, true);
00307
00308   ob::State *start = space.allocState();
00309   ob::State *end = space.allocState();
00310
00311   start->as<ob::DubinsStateSpace::StateType>()->setXY(point1.position.x, point1.position.y);
00312   start->as<ob::DubinsStateSpace::StateType>()->setYaw(point1.angle.asRadians());
00313
00314   end->as<ob::DubinsStateSpace::StateType>()->setXY(point2.position.x, point2.position.y);
00315   end->as<ob::DubinsStateSpace::StateType>()->setYaw(point2.angle.asRadians());
00316
00317   double total = 0;
00318
00319   // Extract the path
00320   ob::DubinsStateSpace::DubinsPath path = space.dubins(start, end);
00321   for (unsigned int i = 0; i < 3; ++i) // Dubins path has up to 3 segments
00322   {
00323     auto type = (*path.type_)[i];
00324     if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_LEFT) {
00325       total += std::abs(path.length_[i]);
00326     } else if (type == ob::DubinsStateSpace::DubinsPathSegmentType::DUBINS_RIGHT) {
00327       total += std::abs(path.length_[i]);
00328     }
00329   }
00330
00331   *distance = space.distance(start, end);
00332   return total < M_PI * 0.75f;
00333 }
```

7.33 cityMap.cpp File Reference

CityMap class implementation.

```
#include "cityMap.h"
#include "utils.h"
#include <set>
#include <spdlog/spdlog.h>
```

7.33.1 Detailed Description

CityMap class implementation.

This file contains the implementation of the CityMap class.

Definition in file cityMap.cpp.

7.34 cityMap.cpp

Go to the documentation of this file.

```
00001
00007 #include "cityMap.h"
00008 #include "utils.h"
00009 #include <set>
00010 #include <spdlog/spdlog.h>
00011
00012 CityMap::CityMap() {
00013   roads.clear();
```

```
00014   intersections.clear();
00015   minLatLon.x = minLatLon.y = maxLatLon.x = maxLatLon.y = 0;
00016 }
00017
00018 void CityMap::loadFile(const std::string &filename) {
00019   spdlog::info("Loading file: {}", filename);
00020
00021   tinyxml2::XMLDocument doc;
00022   // Load the XML file
00023   if (doc.LoadFile(filename.c_str()) != tinyxml2::XML_SUCCESS) {
00024     spdlog::error("Failed to load file: {}", filename);
00025     return;
00026   }
00027
00028   // Extract the bounds of the map
00029   tinyxml2::XMLElement *bounds = doc.FirstChildElement("osm")->FirstChildElement("bounds");
00030   if (!bounds) {
00031     spdlog::error("Failed to extract bounds from file: {}", filename);
00032     return;
00033   }
00034
00035   minLatLon.x = bounds->FloatAttribute("minlon");
00036   minLatLon.y = bounds->FloatAttribute("minlat");
00037   maxLatLon.x = bounds->FloatAttribute("maxlon");
00038   maxLatLon.y = bounds->FloatAttribute("maxlat");
00039
00040   // Define the width and height of the map
00041   width = latLonToXY(minLatLon.y, minLatLon.x).x - latLonToXY(maxLatLon.y, maxLatLon.x).x;
00042   height = latLonToXY(minLatLon.y, minLatLon.x).y - latLonToXY(maxLatLon.y, maxLatLon.x).y;
00043   width = std::abs(width);
00044   height = std::abs(height);
00045
00046   std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
00047   spdlog::info("Loading roads and buildings ...");
00048
00049   // List of highway types to exclude
00050   std::set<std::string> excludedHighways = {"footway", "path",  "pedestrian", "cycleway",
00051                                             "steps",   "track", "bridleway",  "service"};
00052
00053   // List of highway types to include
00054   std::set<std::string> includedHighways = {
00055       "motorway",      "trunk",         "primary",    "secondary",    "tertiary",
      "unclassified", "residential",
00056       "living_street", "motorway_link", "trunk_link", "primary_link", "secondary_link",
      "tertiary_link"};
00057
00058   // Extract the roads
00059   tinyxml2::XMLElement *way = doc.FirstChildElement("osm")->FirstChildElement("way");
00060   int roadId = 0;
00061   while (way) {
00062     road r;
00063     building b;
00064     greenArea g;
00065     waterArea w;
00066     r.width = DEFAULT_ROAD_WIDTH;
00067     r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00068     r.id = roadId;
00069
00070     tinyxml2::XMLElement *nd = way->FirstChildElement("nd");
00071     while (nd) {
00072       tinyxml2::XMLElement *node = doc.FirstChildElement("osm")->FirstChildElement("node");
00073       while (node) {
00074         if (node->IntAttribute("id") == nd->IntAttribute("ref")) {
00075           sf::Vector2f p;
00076           p.x = node->FloatAttribute("lon");
00077           p.y = node->FloatAttribute("lat");
00078
00079           if (r.segments.size() > 0) {
00080             segment s;
00081             s.p1 = r.segments.back().p2;
00082             s.p2 = p;
00083             r.segments.push_back(s);
00084           } else {
00085             segment s;
00086             s.p1 = p;
00087             s.p2 = p;
00088             r.segments.push_back(s);
00089           }
00090
00091           b.points.push_back(p);
00092           g.points.push_back(p);
00093           w.points.push_back(p);
00094           break;
00095         }
00096         node = node->NextSiblingElement("node");
00097       }
00098       nd = nd->NextSiblingElement("nd");
```

```
00099     }
00100
00101     // Remove the first segment (it has the same p1 and p2)
00102     r.segments.erase(r.segments.begin());
00103
00104     std::string highwayType;
00105     bool isHighway = false;
00106     bool isBuilding = false;
00107     bool isUnderground = false;
00108     bool isGreenArea = false;
00109     bool isWaterArea = false;
00110     bool widthSet = false;
00111     bool lanesSet = false;
00112     tinyxml2::XMLElement *tag = way->FirstChildElement("tag");
00113     while (tag) {
00114       if (strcmp(tag->Attribute("k"), "width") == 0) {
00115         r.width = tag->FloatAttribute("v");
00116         widthSet = true;
00117       } else if (strcmp(tag->Attribute("k"), "lanes") == 0) {
00118         r.numLanes = tag->IntAttribute("v");
00119         lanesSet = true;
00120       } else if (strcmp(tag->Attribute("k"), "highway") == 0) {
00121         highwayType = tag->Attribute("v");
00122         isHighway = true;
00123       } else if (strcmp(tag->Attribute("k"), "building") == 0) {
00124         isBuilding = true;
00125       } else if (strcmp(tag->Attribute("k"), "layer") == 0) {
00126         int layerValue = tag->IntAttribute("v");
00127         if (layerValue < 0) {
00128           isUnderground = true;
00129         }
00130       } else if (strcmp(tag->Attribute("k"), "landuse") == 0) {
00131         if (strcmp(tag->Attribute("v"), "forest") == 0 || strcmp(tag->Attribute("v"), "grass") == 0 ||
00132             strcmp(tag->Attribute("v"), "meadow") == 0) {
00133           isGreenArea = true;
00134           g.type = 0;
00135         }
00136       } else if (strcmp(tag->Attribute("k"), "leisure") == 0) {
00137         if (strcmp(tag->Attribute("v"), "park") == 0 || strcmp(tag->Attribute("v"), "garden") == 0) {
00138           isGreenArea = true;
00139           g.type = 1;
00140         }
00141       } else if (strcmp(tag->Attribute("k"), "waterway") == 0 &&
00142                  (strcmp(tag->Attribute("v"), "river") == 0 || strcmp(tag->Attribute("v"), "stream")
      == 0 ||
00143                   strcmp(tag->Attribute("v"), "canal") == 0)) {
00144         isWaterArea = true;
00145       } else if (strcmp(tag->Attribute("k"), "natural") == 0 &&
00146                  (strcmp(tag->Attribute("v"), "water") == 0 || strcmp(tag->Attribute("v"), "wetland")
      == 0)) {
00147         isWaterArea = true;
00148       } else if (strcmp(tag->Attribute("k"), "water") == 0 &&
00149                  (strcmp(tag->Attribute("v"), "lake") == 0 || strcmp(tag->Attribute("v"), "pond") == 0
      ||
00150                   strcmp(tag->Attribute("v"), "river") == 0)) {
00151         isWaterArea = true;
00152       }
00153       tag = tag->NextSiblingElement("tag");
00154     }
00155     if (!widthSet && !lanesSet) {
00156       r.width = DEFAULT_ROAD_WIDTH;
00157       r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00158     } else if (!widthSet) {
00159       r.width = r.numLanes * DEFAULT_LANE_WIDTH;
00160     } else if (!lanesSet) {
00161       r.numLanes = r.width / DEFAULT_LANE_WIDTH;
00162     }
00163     r.width = std::max(r.width, MIN_ROAD_WIDTH);
00164     r.numLanes = std::max(r.numLanes, 1);
00165
00166     if (isUnderground) {
00167       way = way->NextSiblingElement("way");
00168       continue;
00169     }
00170     if (isBuilding) {
00171       buildings.push_back(b);
00172       way = way->NextSiblingElement("way");
00173       continue;
00174     }
00175     if (isGreenArea) {
00176       greenAreas.push_back(g);
00177       way = way->NextSiblingElement("way");
00178       continue;
00179     }
00180     if (isWaterArea) {
00181       waterAreas.push_back(w);
00182       way = way->NextSiblingElement("way");
```

```
00183       continue;
00184     }
00185     if (!isHighway || excludedHighways.find(highwayType) != excludedHighways.end()) {
00186       way = way->NextSiblingElement("way");
00187       continue;
00188     }
00189     if (includedHighways.find(highwayType) != includedHighways.end()) {
00190       roads.push_back(r);
00191       roadId++;
00192     }
00193
00194     way = way->NextSiblingElement("way");
00195   }
00196
00197   // Convert lat/lon to meters (using the upper-left corner as origin)
00198   sf::Vector2f minXY = latLonToXY(minLatLon.y, minLatLon.x);
00199   sf::Vector2f maxXY = latLonToXY(maxLatLon.y, maxLatLon.x);
00200   for (auto &r : roads) {
00201     for (auto &s : r.segments) {
00202       s.p1 = latLonToXY(s.p1.y, s.p1.x);
00203       s.p2 = latLonToXY(s.p2.y, s.p2.x);
00204
00205       s.p1.x -= minXY.x;
00206       s.p1.y -= minXY.y;
00207       s.p2.x -= minXY.x;
00208       s.p2.y -= minXY.y;
00209
00210       // Symetri to the x-axis
00211       s.p1.y = maxXY.y - minXY.y - s.p1.y;
00212       s.p2.y = maxXY.y - minXY.y - s.p2.y;
00213
00214       s.p1_offset = s.p1;
00215       s.p2_offset = s.p2;
00216
00217       s.angle = sf::radians(std::atan2(s.p2.y - s.p1.y, s.p2.x - s.p1.x));
00218     }
00219   }
00220   for (auto &b : buildings) {
00221     for (auto &p : b.points) {
00222       p = latLonToXY(p.y, p.x);
00223
00224       p.x -= minXY.x;
00225       p.y -= minXY.y;
00226
00227       // Symetri to the x-axis
00228       p.y = maxXY.y - minXY.y - p.y;
00229     }
00230   }
00231   for (auto &g : greenAreas) {
00232     for (auto &p : g.points) {
00233       p = latLonToXY(p.y, p.x);
00234
00235       p.x -= minXY.x;
00236       p.y -= minXY.y;
00237
00238       // Symetri to the x-axis
00239       p.y = maxXY.y - minXY.y - p.y;
00240     }
00241   }
00242   for (auto &w : waterAreas) {
00243     for (auto &p : w.points) {
00244       p = latLonToXY(p.y, p.x);
00245
00246       p.x -= minXY.x;
00247       p.y -= minXY.y;
00248
00249       // Symetri to the x-axis
00250       p.y = maxXY.y - minXY.y - p.y;
00251     }
00252   }
00253
00254   std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
00255   spdlog::info("Roads and buildings loaded ({} ms)",
00256                std::chrono::duration_cast<std::chrono::milliseconds>(end - begin).count());
00257
00258   spdlog::info("Loading intersections ...");
00259
00260   // Intersections are at any roads' points if they are near another one
00261   // First add the intersections for each node point
00262   // Then merge the intersections that are close to each other
00263   intersections.clear();
00264   int intersectionId = 0;
00265
00266   // Add the intersections for each road segment
00267   spdlog::debug("Adding intersections ...");
00268   for (auto r : roads) {
00269     for (int s_id = 0; s_id < (int)r.segments.size(); s_id++) {
```

```
00270       segment s = r.segments[s_id];
00271       std::vector<sf::Vector2f> points = {s.p1, s.p2};
00272       for (auto p : points) {
00273         intersection i = {intersectionId++, p, r.width / 2, {}};
00274         i.roadSegmentIds.push_back({r.id, s_id});
00275         intersections.push_back(i);
00276       }
00277     }
00278   }
00279   spdlog::debug("Intersections added");
00280
00281   // Merge the intersections that are close to each other
00282   spdlog::debug("Merging intersections ...");
00283   for (int distCoef = 5; distCoef > 0; distCoef -= 1) {
00284     for (int i = 0; i < (int)intersections.size(); i++) {
00285       for (int j = i + 1; j < (int)intersections.size(); j++) {
00286         bool is_i = intersections[i].roadSegmentIds.size() > intersections[j].roadSegmentIds.size();
00287
00288         if (intersections[i].roadSegmentIds.size() == intersections[j].roadSegmentIds.size()) {
00289           is_i = intersections[i].id < intersections[j].id;
00290         }
00291
00292         double minSpace = intersections[i].radius + intersections[j].radius;
00293         minSpace /= distCoef;
00294
00295         if (distance(intersections[i].center, intersections[j].center) < minSpace) {
00296           // Merge the intersections to i or j (depending on is_i)
00297           int index_from = is_i ? j : i;
00298           int index_to = is_i ? i : j;
00299
00300           for (auto &r : intersections[index_from].roadSegmentIds) {
00301             intersections[index_to].roadSegmentIds.push_back(r);
00302           }
00303
00304           intersections.erase(intersections.begin() + index_from);
00305           i -= 1;
00306           break;
00307         }
00308       }
00309     }
00310   }
00311   spdlog::debug("Intersections merged");
00312
00313   // Make the road point to be outside the intersection
00314   spdlog::debug("Adding offsets to the roads ...");
00315   for (auto &i : intersections) {
00316     for (auto &roadInfo : i.roadSegmentIds) {
00317       double dx =
00318           roads[roadInfo.first].segments[roadInfo.second].p2.x -
      roads[roadInfo.first].segments[roadInfo.second].p1.x;
00319       double dy =
00320           roads[roadInfo.first].segments[roadInfo.second].p2.y -
      roads[roadInfo.first].segments[roadInfo.second].p1.y;
00321       double dd = distance({0, 0}, {(float)dx, (float)dy});
00322       dx /= dd;
00323       dy /= dd;
00324
00325       double radius = i.radius;
00326
00327       if (distance(roads[roadInfo.first].segments[roadInfo.second].p1, i.center) <
00328           distance(roads[roadInfo.first].segments[roadInfo.second].p2, i.center)) {
00329         roads[roadInfo.first].segments[roadInfo.second].p1_offset.x = i.center.x + dx * radius;
00330         roads[roadInfo.first].segments[roadInfo.second].p1_offset.y = i.center.y + dy * radius;
00331       } else {
00332         dx = -dx;
00333         dy = -dy;
00334         roads[roadInfo.first].segments[roadInfo.second].p2_offset.x = i.center.x + dx * radius;
00335         roads[roadInfo.first].segments[roadInfo.second].p2_offset.y = i.center.y + dy * radius;
00336       }
00337     }
00338   }
00339   spdlog::debug("Offsets added");
00340
00341   // Remove the intersections that link the same road
00342   spdlog::debug("Removing intersections that link the same road ...");
00343   for (int i = 0; i < (int)intersections.size(); i++) {
00344     if (intersections[i].roadSegmentIds.size() != 2)
00345       continue;
00346
00347     if (intersections[i].roadSegmentIds[0].first == intersections[i].roadSegmentIds[1].first) {
00348       intersections.erase(intersections.begin() + i);
00349       i -= 1;
00350     }
00351   }
00352   spdlog::debug("Intersections removed");
00353
00354   // Log all the intersections and roads
```

```
00355   for (auto r : roads) {
00356     spdlog::debug("Road: id={}, width={}, numLanes={}, segments={}", r.id, r.width, r.numLanes,
      r.segments.size());
00357   }
00358   for (auto i : intersections) {
00359     spdlog::debug("Intersection: id={}, center=({}, {}), radius={}, roadSegmentIds={}", i.id,
      i.center.x, i.center.y,
00360                   i.radius, i.roadSegmentIds.size());
00361   }
00362
00363   std::chrono::steady_clock::time_point end2 = std::chrono::steady_clock::now();
00364   spdlog::info("Intersections loaded ({} ms)",
00365                std::chrono::duration_cast<std::chrono::milliseconds>(end2 - end).count());
00366
00367   spdlog::info("Number of roads: {}", roads.size());
00368   spdlog::info("Number of buildings: {}", buildings.size());
00369   spdlog::info("Number of intersections: {}", intersections.size());
00370
00371   spdlog::info("Width: {} m", width);
00372   spdlog::info("Height: {} m", height);
00373
00374   isLoaded = true;
00375 }
```

7.35 dataManager.cpp File Reference

Data manager.

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

7.35.1 Detailed Description

Data manager.

This file contains the implementation of the DataManager class.

Definition in file dataManager.cpp.

7.36 dataManager.cpp

Go to the documentation of this file.

```
00001
00007 #include "dataManager.h"
00008 #include "cityGraph.h"
00009 #include "cityMap.h"
00010 #include "config.h"
00011 #include "manager.h"
00012 #include <filesystem>
00013 #include <fstream>
00014 #include <iostream>
00015 #include <random>
00016 #include <spdlog/spdlog.h>
```

```
00017
00018 DataManager::DataManager(std::string filename) {
00019   // Create /data folder if it doesn't exist
00020   if (!std::filesystem::exists("data")) {
00021     spdlog::debug("Creating data folder");
00022     std::filesystem::create_directory("data");
00023   }
00024 }
00025
00026 void DataManager::createData(int numData, int numCarsMin, int numCarsMax, std::string mapName) {
00027
00028   spdlog::error("Deprecated: Need to be updated to use the new manager system");
00029
00030   return;
00031
00032   // // If numData is less than 1, default to a very high number (as in your original code).
00033   // numData = numData < 1 ? INT_MAX : numData;
00034   //
00035   // // Remove file extension from mapName to construct the output filename.
00036   // std::string mapNameNoExt = mapName.substr(0, mapName.find_last_of("."));
00037   // std::string filename = "data/" + mapNameNoExt + "_" + std::to_string((int)CBS_MAX_SUB_TIME) +
00038   //                        (ROAD_ENABLE_RIGHT_HAND_TRAFFIC ? "_RHT" : "") + "_data.csv";
00039   //
00040   // // Load the city map.
00041   // CityMap cityMap;
00042   // cityMap.loadFile("assets/map/" + mapName);
00043   //
00044   // // Create the city graph.
00045   // CityGraph cityGraph;
00046   // cityGraph.createGraph(cityMap);
00047   //
00048   // // Open the output file in append mode.
00049   // std::ofstream file;
00050   // file.open(filename, std::ios::app);
00051   // if (!file.is_open()) {
00052   //   spdlog::error("Failed to open file {}", filename);
00053   //   return;
00054   // }
00055   //
00056   // std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
00057   // std::uniform_int_distribution<int> dist(numCarsMin, numCarsMax);
00058   //
00059   // for (int i = 0; i < numData; i += 1) {
00060   //   int numCars = dist(rng);
00061   //
00062   //   Manager manager(cityGraph, cityMap, false);
00063   //   auto resData = manager.createCarsCBS(numCars);
00064   //   if (!resData.first) {
00065   //     spdlog::warn("Data {}: CBS failed (numCars: {})", i + 1, numCars);
00066   //     i--;
00067   //     continue;
00068   //   }
00069   //
00070   //   data validResData = resData.second;
00071   //
00072   //   file « validResData.numCars « ";" « validResData.carDensity;
00073   //   for (auto speed : validResData.carAvgSpeed) {
00074   //     file « ";" « speed;
00075   //   }
00076   //   file « std::endl;
00077   //
00078   //   if (numData == INT_MAX) {
00079   //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, validResData.numCars,
00080   //     validResData.carDensity);
00081   //   } else {
00082   //     spdlog::info("Data {}: numCars: {}, carDensity: {:0>6.5}", i + 1, numData,
      validResData.numCars,
00083   //                  validResData.carDensity);
00084   //   }
00085   // }
00086   //
00087   // file.close();
00088 }
```

7.37 interpolator.cpp File Reference

```
#include "aStar.h"
#include "dubins.h"
#include <ompl/base/State.h>
#include <ompl/base/StateSpace.h>
```

```
#include <ompl/base/spaces/DubinsStateSpace.h>
#include <spdlog/spdlog.h>
```

7.38 interpolator.cpp

Go to the documentation of this file.

```
00001 #include "aStar.h"
00002 #include "dubins.h"
00003 #include <ompl/base/State.h>
00004 #include <ompl/base/StateSpace.h>
00005 #include <ompl/base/spaces/DubinsStateSpace.h>
00006 #include <spdlog/spdlog.h>
00007
00008 namespace ob = ompl::base;
00009
00010 void DubinsInterpolator::init(CityGraph::point start_, CityGraph::point end_, double radius_) {
00011   startPoint = start_;
00012   endPoint = end_;
00013   radius = radius_;
00014
00015   ob::DubinsStateSpace space = ob::DubinsStateSpace(radius, true);
00016
00017   ob::State *start = space.allocState();
00018   ob::State *end = space.allocState();
00019
00020   start->as<ob::DubinsStateSpace::StateType>()->setXY(startPoint.position.x, startPoint.position.y);
00021   start->as<ob::DubinsStateSpace::StateType>()->setYaw(startPoint.angle.asRadians());
00022
00023   end->as<ob::DubinsStateSpace::StateType>()->setXY(endPoint.position.x, endPoint.position.y);
00024   end->as<ob::DubinsStateSpace::StateType>()->setYaw(endPoint.angle.asRadians());
00025
00026   distance = space.distance(start, end);
00027
00028   sf::Vector2 diff = startPoint.position - endPoint.position;
00029   double absDist = std::sqrt(std::pow(diff.x, 2) + std::pow(diff.y, 2));
00030   if (distance > absDist + 2 * M_PI * radius) {
00031     spdlog::warn("Distance is way too big in DubinsInterpolator");
00032     distance = absDist;
00033   }
00034   if (distance + 0.1 < absDist) { // 0.1 is a small tolerance
00035     spdlog::warn("Distance is way too small in DubinsInterpolator");
00036     distance = absDist;
00037   }
00038
00039   double dx = DUBINS_INTERPOLATION_STEP / distance;
00040   interpolatedCurve.clear();
00041   interpolatedCurve.push_back(startPoint);
00042
00043   for (double x = dx; x < 1; x += dx) {
00044     if (x == 1)
00045       continue;
00046
00047     ob::State *state = space.allocState();
00048     space.interpolate(start, end, x, state);
00049     double x_ = state->as<ob::DubinsStateSpace::StateType>()->getX();
00050     double y_ = state->as<ob::DubinsStateSpace::StateType>()->getY();
00051     double yaw_ = state->as<ob::DubinsStateSpace::StateType>()->getYaw();
00052
00053     CityGraph::point point;
00054     point.position = {(float)x_, (float)y_};
00055     point.angle = sf::radians(yaw_);
00056
00057     interpolatedCurve.push_back(point);
00058
00059     space.freeState(state);
00060   }
00061
00062   interpolatedCurve.push_back(endPoint);
00063
00064   numInterpolatedPoints = interpolatedCurve.size();
00065
00066   space.freeState(start);
00067   space.freeState(end);
00068 }
00069
00070 CityGraph::point DubinsInterpolator::get(double time, double startSpeed, double endSpeed) {
00071   double acc = (std::pow(endSpeed, 2) - std::pow(startSpeed, 2)) / (2 * distance);
00072   auto xFun = [&](double t) { return (0.5 * acc * std::pow(t, 2) + startSpeed * t) / distance; };
00073
```

```
00074   int index = std::round((numInterpolatedPoints - 1) * xFun(time));
00075   index = std::clamp(index, 0, numInterpolatedPoints - 1);
00076
00077   return interpolatedCurve[index];
00078 }
```

7.39 fileSelector.cpp File Reference

File selector implementation.

```
#include "fileSelector.h"
#include "config.h"
#include <filesystem>
#include <spdlog/spdlog.h>
```

7.39.1 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.

7.40 fileSelector.cpp

Go to the documentation of this file.

```
00001
00007 #include "fileSelector.h"
00008 #include "config.h"
00009 #include <filesystem>
00010 #include <spdlog/spdlog.h>
00011
00012 namespace fs = std::filesystem;
00013
00014 void FileSelector::loadFiles() {
00015   files.clear();
00016   for (const auto &entry : fs::directory_iterator(folderPath)) {
00017     if (entry.is_regular_file() && entry.path().extension() == ".osm") {
00018       files.push_back(entry.path().filename().string());
00019     }
00020   }
00021   std::sort(files.begin(), files.end());
00022 }
00023
00024 char FileSelector::getKeyPress() {
00025   struct termios oldt, newt;
00026   char ch;
00027   tcgetattr(STDIN_FILENO, &oldt);
00028   newt = oldt;
00029   newt.c_lflag &= ~(ICANON | ECHO);
00030   tcsetattr(STDIN_FILENO, TCSANOW, &newt);
00031   ch = getchar();
00032   tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
00033   return ch;
00034 }
00035
00036 void FileSelector::moveCursorUp() {
00037   if (selectedIndex > 0) {
00038     std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
00039     selectedIndex--;
00040     std::cout « "\033[A\033[2K\r> " « files[selectedIndex] « std::flush;
00041   }
00042 }
00043
```

```
00044 void FileSelector::moveCursorDown() {
00045   if (selectedIndex < files.size() - 1) {
00046     std::cout « "\033[2K\r  " « files[selectedIndex] « std::flush;
00047     selectedIndex++;
00048     std::cout « "\033[B\033[2K\r> " « files[selectedIndex] « std::flush;
00049   }
00050 }
00051
00052 void FileSelector::displayFiles() {
00053   std::cout « "Use UP/DOWN arrow keys to navigate, ENTER to select:\n";
00054   for (size_t i = 0; i < files.size(); i++) {
00055     if (i == selectedIndex) {
00056       std::cout « "> " « files[i] « "\n";
00057     } else {
00058       std::cout « "  " « files[i] « "\n";
00059     }
00060   }
00061   std::cout « "\033[" « files.size() « "A";
00062 }
00063
00064 std::string FileSelector::selectFile() {
00065   std::cout « "\033[?25l";
00066   if (files.empty()) {
00067     spdlog::error("No .osm files found in the folder: {}", folderPath);
00068     return "";
00069   }
00070
00071   displayFiles();
00072
00073   while (true) {
00074     char key = getKeyPress();
00075     if (key == 27) {
00076       if (getKeyPress() == '[') {
00077         switch (getKeyPress()) {
00078         case 'A':
00079           moveCursorUp();
00080           break;
00081         case 'B':
00082           moveCursorDown();
00083           break;
00084         }
00085       }
00086     } else if (key == '\n') {
00087       std::cout « "\033[" « selectedIndex + 1 « "A\033[2K\r" « std::flush;
00088       std::cout « "\033[?25h";
00089       spdlog::info("Selected file: {}", files[selectedIndex]);
00090       return files[selectedIndex];
00091     }
00092   }
00093 }
```

7.41 main.cpp File Reference

Main file.

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

Functions

- int main (int nArgs, char **args)

7.41.1 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.

7.41.2 Function Documentation

7.41.2.1 main()

```
int main (
            int nArgs,
            char ** args )
```

Definition at line 18 of file main.cpp.

```
00018                                   {
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]",
      args[0]);
00024     return 1;
00025   }
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
00047   if (ENVIRONMENT == 0) {
00048     spdlog::set_level(spdlog::level::debug);
00049     Test test;
00050     test.runTests();
00051   } else {
00052     spdlog::set_level(spdlog::level::info);
00053   }
00054
00055   if (data) {
00056     spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
      dataNumData,
00057                  dataNumCarsMin, dataNumCarsMax);
00058
00059     DataManager dataManager(mapFile);
00060     dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
00061   } else {
00062     spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);
00063
00064     CityMap cityMap;
00065     cityMap.loadFile("assets/map/" + mapFile);
00066
00067     CityGraph cityGraph;
00068     cityGraph.createGraph(cityMap);
00069
00070     ManagerCBS manager(cityGraph, cityMap);
00071     manager.initializeAgents(runNumCars);
00072
00073     Renderer renderer;
00074     renderer.startRender(cityMap, cityGraph, manager);
00075   }
00076
00077   return 0;
00078 }
```

7.42 main.cpp

Go to the documentation of this file.

```
00001
00007 #include "cityMap.h"
00008 #include "config.h"
00009 #include "dataManager.h"
00010 #include "fileSelector.h"
00011 #include "manager.h"
00012 #include "manager_ocbs.h"
00013 #include "renderer.h"
00014 #include "spdlog/spdlog.h"
00015 #include "test.h"
00016 #include <SFML/Graphics.hpp>
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]",
      args[0]);
00024     return 1;
00025   }
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
00047   if (ENVIRONMENT == 0) {
00048     spdlog::set_level(spdlog::level::debug);
00049     Test test;
00050     test.runTests();
00051   } else {
00052     spdlog::set_level(spdlog::level::info);
00053   }
00054
00055   if (data) {
00056     spdlog::info("Creating data for map {}, numData: {}, numCarsMin: {}, numCarsMax: {}", mapFile,
      dataNumData,
00057                  dataNumCarsMin, dataNumCarsMax);
00058
00059     DataManager dataManager(mapFile);
00060     dataManager.createData(dataNumData, dataNumCarsMin, dataNumCarsMax, mapFile);
00061   } else {
00062     spdlog::info("Running simulation for map {}, numCars: {}", mapFile, runNumCars);
00063
00064     CityMap cityMap;
00065     cityMap.loadFile("assets/map/" + mapFile);
00066
00067     CityGraph cityGraph;
00068     cityGraph.createGraph(cityMap);
00069
00070     ManagerCBS manager(cityGraph, cityMap);
00071     manager.initializeAgents(runNumCars);
00072
00073     Renderer renderer;
00074     renderer.startRender(cityMap, cityGraph, manager);
00075   }
00076
00077   return 0;
00078 }
```

7.43 index.cpp File Reference

Implementation of the Manager class.

```
#include "manager.h"
```

7.43.1 Detailed Description

Implementation of the Manager class.

This file contains the implementation of the Manager class.

Definition in file index.cpp.

7.44 index.cpp

Go to the documentation of this file.

```
00001
00007 #include "manager.h"
00008
00009 void Manager::initializeAgents(int numCars) {
00010   spdlog::info("Initializing {} agent...", numCars);
00011   this->numCars = numCars;
00012
00013   for (int i = 0; i < numCars; i++) {
00014     Car car;
00015     cars.push_back(car);
00016   }
00017
00018   // Create a path for each car (random start and end points)
00019   for (int i = 0; i < numCars; i++) {
00020     bool valid = false;
00021     cars[i].chooseRandomStartEndPath(graph, map);
00022   }
00023
00024   spdlog::info("Initialized {} agents", cars.size());
00025 }
00026
00027 void Manager::updateAgents() {
00028   for (Car &car : cars) {
00029     car.move();
00030   }
00031 }
00032
00033 void Manager::renderAgents(sf::RenderWindow &window) {
00034   for (Car &car : cars) {
00035     car.render(window);
00036   }
00037 }
```

7.45 ocbs.cpp File Reference

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

7.46 ocbs.cpp

Go to the documentation of this file.

```
00001 #include "aStar.h"
00002 #include "config.h"
00003 #include "dubins.h"
00004 #include "manager_ocbs.h" #include < spdlog / spdlog.h>
00005
00006 void ManagerOCBS::userInput(sf::Event event, sf::RenderWindow &window) {
00007   // If left mouse click over a car, toggle debug for that car
00008   if (event.is<sf::Event::MouseButtonPressed>() &&
00009       event.getIf<sf::Event::MouseButtonPressed>()->button == sf::Mouse::Button::Left) {
00010     sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));
00011     for (int i = 0; i < numCars; i++) {
00012       sf::Vector2f diff = cars[i].getPosition() - mousePos;
00013       double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00014       if (len < 2 * CAR_LENGTH) {
00015         cars[i].toggleDebug();
00016         spdlog::debug("Toggling debug for car {}", i);
00017         return;
00018       }
00019     }
00020   }
00021 }
00022
00023 void ManagerOCBS::planPaths() {
00024   openSet = std::priority_queue<Node>();
00025   starts.clear();
00026   starts.resize(numCars);
00027   ends.clear();
00028   ends.resize(numCars);
00029   baseCosts.clear();
00030   baseCosts.resize(numCars);
00031
00032   Node node;
00033   node.paths.resize(numCars);
00034   node.costs.resize(numCars);
00035   node.cost = 0;
00036   node.depth = 0;
00037   node.hasResolved = false;
00038   conflicts.clear();
00039
00040   for (int i = 0; i < numCars; i++) {
00041     node.paths[i] = cars[i].getPath();
00042     node.costs[i] = cars[i].getPathTime();
00043     node.cost += node.costs[i];
00044     baseCosts[i] = node.costs[i];
00045     starts[i] = cars[i].getStart();
00046     ends[i] = cars[i].getEnd();
00047   }
00048
00049   openSet.push(node);
00050   spdlog::info("Starting to find paths using CBS");
00051   findPaths();
00052 }
00053
00054 void ManagerOCBS::initializePaths(Node *node) {
00055   for (int i = 0; i < numCars; i++) {
00056     spdlog::debug("Finding path for car {}", i);
00057     pathfinding(node, i);
00058   }
00059 }
00060
00061 bool ManagerOCBS::findConflict(int *car1, int *car2, int *time, Node *node) {
00062   int maxPathLength = -1;
00063   for (int i = 0; i < numCars; i++) {
00064     maxPathLength = std::max(maxPathLength, (int)node->paths[i].size());
00065   }
00066
00067   for (int t = 0; t < maxPathLength; t++) {
00068     for (int i = 0; i < numCars; i++) {
00069       for (int j = i + 1; j < numCars; j++) {
00070         sf::Vector2f diff = node->paths[i][t] - node->paths[j][t];
00071         double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00072         if (len < CAR_LENGTH * 1.1) {
00073           *car1 = i;
00074           *car2 = j;
00075           *time = t;
00076           return true;
00077         }
00078       }
00079     }
00080   }
00081
00082   return false;
```

```
00083 }
00084
00085 bool ManagerOCBS::findPaths() {
00086   if (openSet.empty()) {
00087     spdlog::info("No solution found");
00088     return false;
00089   }
00090
00091   Node node = openSet.top();
00092   openSet.pop();
00093
00094   spdlog::debug("Processing node with cost: {}", node.cost);
00095
00096   int car1, car2, time;
00097   if (!findConflict(&car1, &car2, &time, &node)) {
00098     spdlog::info("Found solution with cost: {}", node.cost);
00099
00100     for (int i = 0; i < numCars; i++) {
00101       cars[i].assignExistingPath(node.paths[i]);
00102     }
00103
00104     return true;
00105   }
00106
00107   spdlog::debug("Found conflict between car {} and car {} at time {}", car1, car2, time);
00108
00109   // Witch car is the most affected
00110   int car1Index = car1;
00111   int car2Index = car2;
00112
00113   double ratio1 = node.costs[car1] / baseCosts[car1];
00114   double ratio2 = node.costs[car2] / baseCosts[car2];
00115
00116   if (ratio1 > ratio2) {
00117     car1Index = car2;
00118     car2Index = car1;
00119   }
00120
00121   ConflictSituation situation1;
00122   situation1.car = car1Index;
00123   situation1.time = time * SIM_STEP_TIME;
00124   situation1.at = node.paths[car1Index][time];
00125   ConflictSituation situation2;
00126   situation2.car = car1Index;
00127   situation2.time = time * SIM_STEP_TIME;
00128   situation2.at = node.paths[car2Index][time];
00129
00130   Conflict conflict1;
00131   conflict1.car = car1Index;
00132   conflict1.withCar = car2Index;
00133   conflict1.time = time * SIM_STEP_TIME;
00134   conflict1.position = node.paths[car1Index][time];
00135
00136   Conflict conflict2;
00137   conflict2.car = car1Index;
00138   conflict2.withCar = car2Index;
00139   conflict2.time = time * SIM_STEP_TIME;
00140   conflict2.position = node.paths[car2Index][time];
00141
00142   if (conflicts.find(situation1) == conflicts.end()) {
00143     conflicts[situation1] = new std::unordered_set<Conflict>();
00144   }
00145   conflicts[situation1]->insert(conflict1);
00146
00147   if (conflicts.find(situation2) == conflicts.end()) {
00148     conflicts[situation2] = new std::unordered_set<Conflict>();
00149   }
00150   conflicts[situation2]->insert(conflict2);
00151
00152   openSet.push(node);
00153
00154   pathfinding(&node, car1Index);
00155   return findPaths();
00156 }
00157
00158 void ManagerOCBS::pathfinding(Node *node, int carIndex) {
00159   AStar::node start;
00160   start.point = starts[carIndex];
00161   start.speed = 0;
00162   AStar::node end;
00163   end.point = ends[carIndex];
00164   end.speed = 0;
00165
00166   std::unordered_map<AStar::node, AStar::node> cameFrom;
00167   std::unordered_map<AStar::node, double> gScore;
00168   std::unordered_map<AStar::node, double> fScore;
00169
```

```
00170   auto heuristic = [&](const AStar::node &a) {
00171     return 0.0;
00172     sf::Vector2f diff = end.point.position - a.point.position;
00173     double distance = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00174     return distance / CAR_MAX_SPEED_MS;
00175   };
00176   auto compare = [&](const AStar::node &a, const AStar::node &b) { return fScore[a] > fScore[b]; };
00177
00178   std::priority_queue<AStar::node, std::vector<AStar::node>, decltype(compare)> openSetAstar(compare);
00179   std::unordered_set<AStar::node> isInOpenSet;
00180
00181   openSetAstar.push(start);
00182   gScore[start] = 0;
00183   fScore[start] = heuristic(start);
00184
00185   auto neighbors = graph.getNeighbors();
00186
00187   int nbIterations = 0;
00188   while (!openSetAstar.empty() && nbIterations++ < 1e5) {
00189     AStar::node current = openSetAstar.top();
00190     openSetAstar.pop();
00191     isInOpenSet.erase(current);
00192
00193     if (current.point == end.point) {
00194       AStar::node currentCopy = current;
00195       std::vector<AStar::node> nodePaths;
00196
00197       while (!(currentCopy == start)) {
00198         nodePaths.push_back(currentCopy);
00199         currentCopy = cameFrom[currentCopy];
00200       }
00201
00202       nodePaths.push_back(currentCopy);
00203       std::reverse(nodePaths.begin(), nodePaths.end());
00204
00205       double oldCost = node->costs[carIndex];
00206       cars[carIndex].assignPath(nodePaths, graph);
00207
00208       node->paths[carIndex] = cars[carIndex].getPath();
00209       node->costs[carIndex] = cars[carIndex].getPathTime();
00210       node->cost += node->costs[carIndex] - oldCost;
00211
00212       spdlog::debug("Found path for car {} with cost: {}", carIndex, node->costs[carIndex]);
00213       return;
00214     }
00215
00216     for (const auto &neighborGraphPoint : neighbors[current.point]) {
00217       if (current.speed > neighborGraphPoint.maxSpeed)
00218         continue;
00219
00220       if (!neighborGraphPoint.isRightWay && ROAD_ENABLE_RIGHT_HAND_TRAFFIC)
00221         continue;
00222
00223       std::vector<double> newSpeeds;
00224       newSpeeds.push_back(current.speed);
00225
00226       double distance = graph.getInterpolator(current.point, neighborGraphPoint)->getDistance();
00227       double nSpeedAcc = std::sqrt(std::pow(current.speed, 2) + 2 * CAR_ACCELERATION * distance);
00228       double nSpeedDec = std::sqrt(std::pow(current.speed, 2) - 2 * CAR_DECELERATION * distance);
00229
00230       auto push = [&](double nSpeed) {
00231         int numSpeedDiv = 5;
00232         for (int i = 1; i < numSpeedDiv + 1; i++) {
00233           double s = (current.speed + (nSpeed - current.speed) * i / numSpeedDiv);
00234           if (s < SPEED_RESOLUTION)
00235             continue;
00236           newSpeeds.push_back(s);
00237         }
00238       };
00239
00240       if (nSpeedAcc > neighborGraphPoint.maxSpeed && current.speed < neighborGraphPoint.maxSpeed) {
00241         push(neighborGraphPoint.maxSpeed);
00242       } else if (nSpeedAcc < neighborGraphPoint.maxSpeed) {
00243         push(nSpeedAcc);
00244       }
00245
00246       if (nSpeedDec == nSpeedDec && std::isfinite(nSpeedDec)) { // check if nSpeedDec is finite and
    not NaN
00247         if (nSpeedDec < 0 && current.speed > 0) {
00248           push(0);
00249         } else if (nSpeedDec >= 0) {
00250           push(nSpeedDec);
00251         }
00252       }
00253
00254       AStar::node neighbor;
00255       neighbor.point = neighborGraphPoint.point;
```

```
00256       neighbor.arcFrom = {current.point, neighborGraphPoint};
00257       if (distance == 0) {
00258         neighbor.speed = current.speed;
00259         if (gScore.find(neighbor) == gScore.end() || gScore[current] < gScore[neighbor]) {
00260           cameFrom[neighbor] = current;
00261           gScore[neighbor] = gScore[current];
00262           fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00263 
00264           if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00265             openSetAstar.push(neighbor);
00266             isInOpenSet.insert(neighbor);
00267           }
00268         }
00269         continue;
00270       }
00271 
00272       for (const auto &newSpeed : newSpeeds) {
00273         if (newSpeed > CAR_MAX_SPEED_MS || newSpeed > neighborGraphPoint.maxSpeed || newSpeed < 0)
00274           continue;
00275 
00276         if (newSpeed == current.speed && newSpeed == 0)
00277           continue;
00278 
00279         neighbor.speed = newSpeed;
00280 
00281         double duration = 2 * distance / (current.speed + newSpeed);
00282         double tentativeGScore = gScore[current] + duration;
00283 
00284         double t = gScore[current];
00285         bool conflictFree = true;
00286 
00287         // Checking for conflicts
00288         DubinsInterpolator *interpolator = graph.getInterpolator(current.point, neighborGraphPoint);
00289         for (double tt = 0; tt < duration; tt = tt + SIM_STEP_TIME) {
00290           ConflictSituation confS;
00291           confS.car = carIndex;
00292           confS.at = interpolator->get(tt, current.speed, newSpeed).position;
00293           confS.time = t + tt;
00294 
00295           if (conflicts.find(confS) == conflicts.end()) {
00296             continue;
00297           }
00298 
00299           std::unordered_set<Conflict> *conflictSet = conflicts[confS];
00300 
00301           if (conflictSet->size() == 0) {
00302             continue;
00303           }
00304 
00305           conflictFree = true;
00306           break;
00307 
00308           for (const auto &conf : *conflictSet) {
00309             // Check durin all the duration if there is a conflict
00310             sf::Vector2f diff = confS.at - conf.position;
00311             double len = std::sqrt(diff.x * diff.x + diff.y * diff.y);
00312 
00313             if (len < CAR_LENGTH * 1.1) {
00314               conflictFree = false;
00315               break;
00316             }
00317           }
00318           if (!conflictFree)
00319             break;
00320         }
00321 
00322         if (!conflictFree)
00323           continue;
00324 
00325         if (gScore.find(neighbor) == gScore.end() || tentativeGScore < gScore[neighbor]) {
00326           cameFrom[neighbor] = current;
00327           gScore[neighbor] = tentativeGScore;
00328           fScore[neighbor] = gScore[neighbor] + heuristic(neighbor);
00329 
00330           if (isInOpenSet.find(neighbor) == isInOpenSet.end()) {
00331             openSetAstar.push(neighbor);
00332             isInOpenSet.insert(neighbor);
00333           }
00334         }
00335       }
00336     }
00337   }
00338 
00339   spdlog::warn("A* failed to find a path for car {}", carIndex);
00340 }
```

7.47 renderer.cpp File Reference

Implementation of the Renderer class.

```
#include "renderer.h"
#include "config.h"
#include "utils.h"
#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 <vector>
```

7.47.1 Detailed Description

Implementation of the Renderer class.

This file contains the implementation of the Renderer class.

Definition in file renderer.cpp.

7.48 renderer.cpp

Go to the documentation of this file.

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

```
00045   window.display();
00046   time = 0;
00047
00048   sf::Clock clockCars;
00049   bool speedUp = false;
00050   bool pause = true;
00051
00052   while (true) {
00053     while (const std::optional event = window.pollEvent()) {
00054       if (event->is<sf::Event::Closed>()) {
00055         window.close();
00056         return;
00057       }
00058
00059       if (event->is<sf::Event::KeyPressed>() || event->is<sf::Event::MouseButtonPressed>()) {
00060         manager.userInput(event.value(), window);
00061       }
00062
00063       if (const auto *resized = event->getIf<sf::Event::Resized>()) {
00064         resetView();
00065       }
00066
00067       if (!event->is<sf::Event::KeyPressed>())
00068         continue;
00069
00070       if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Escape) {
00071         window.close();
00072       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Up) {
00073         view.move({0, -(float)(height * MOVE_SPEED)});
00074       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Down) {
00075         view.move({0, +(float)(height * MOVE_SPEED)});
00076       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Left) {
00077         view.move({-(float)(width * MOVE_SPEED), 0});
00078       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Right) {
00079         view.move({+(float)(width * MOVE_SPEED), 0});
00080       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Equal) {
00081         view.zoom(1.0f - ZOOM_SPEED);
00082       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::Subtract) {
00083         view.zoom(1.0f + ZOOM_SPEED);
00084       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::R) {
00085         resetView();
00086         spdlog::debug("View reset");
00087       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::D) {
00088         debug = !debug;
00089         spdlog::debug("Debug mode: {}", debug);
00090       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::S) {
00091         speedUp = !speedUp;
00092       } else if (event->getIf<sf::Event::KeyPressed>()->code == sf::Keyboard::Key::P) {
00093         pause = !pause;
00094       }
00095     }
00096
00097     window.setView(view);
00098     window.clear(sf::Color(247, 246, 242));
00099     renderCityMap(cityMap);
00100     renderManager(manager);
00101     if (!pause) {
00102       if (clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME ||
00103           (speedUp && clockCars.getElapsedTime().asSeconds() > SIM_STEP_TIME / 5)) {
00104         time += SIM_STEP_TIME;
00105         manager.updateAgents();
00106         clockCars.restart();
00107       }
00108     }
00109     if (debug) {
00110       renderCityGraph(cityGraph, view);
00111     }
00112     // Remove outside the border (draw blank)
00113     sf::RectangleShape rectangle(sf::Vector2f(width, height));
00114     rectangle.setFillColor(sf::Color(247, 246, 242));
00115
00116     float w = width;
00117     float h = height;
00118
00119     std::vector<sf::Vector2f> border = {{-w, -h}, {0, -h}, {w, -h}, {w, 0}, {w, h}, {0, h}, {-w, h},
      {-w, 0}};
00120     for (auto b : border) {
00121       rectangle.setPosition(b);
00122       window.draw(rectangle);
00123     }
00124
00125     renderTime();
00126     window.display();
00127   }
00128 }
00129
00130 void Renderer::renderCityMap(const CityMap &cityMap) {
```

```
00131   // Draw buildings
00132   std::vector<sf::Color> randomBuildingColors = {
00133       sf::Color(233, 234, 232), sf::Color(238, 231, 210), sf::Color(230, 229, 226), sf::Color(236,
      234, 230),
00134       sf::Color(230, 223, 216), sf::Color(230, 234, 236), sf::Color(210, 215, 222)};
00135
00136   std::vector<sf::Color> greenAreaColor = {sf::Color(184, 230, 144), sf::Color(213, 240, 193)};
00137
00138   sf::Color waterColor(139, 214, 245);
00139
00140   auto greenAreas = cityMap.getGreenAreas();
00141   for (int i = 0; i < (int)greenAreas.size(); i++) {
00142     const auto &greenArea = greenAreas[i];
00143     auto points = greenArea.points;
00144     sf::ConvexShape convex;
00145     convex.setPointCount(points.size());
00146     for (size_t i = 0; i < points.size(); i++) {
00147       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00148     }
00149     convex.setFillColor(greenAreaColor[greenArea.type]);
00150
00151     window.draw(convex);
00152   }
00153
00154   auto waterAreas = cityMap.getWaterAreas();
00155   for (int i = 0; i < (int)waterAreas.size(); i++) {
00156     const auto &waterArea = waterAreas[i];
00157     auto points = waterArea.points;
00158     sf::ConvexShape convex;
00159     convex.setPointCount(points.size());
00160     for (size_t i = 0; i < points.size(); i++) {
00161       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00162     }
00163     convex.setFillColor(waterColor);
00164
00165     window.draw(convex);
00166   }
00167
00168   auto buildings = cityMap.getBuildings();
00169   for (int i = 0; i < (int)buildings.size(); i++) {
00170     const auto &building = buildings[i];
00171     auto points = building.points;
00172     sf::ConvexShape convex;
00173     convex.setPointCount(points.size());
00174     for (size_t i = 0; i < points.size(); i++) {
00175       convex.setPoint(i, sf::Vector2f(points[i].x, points[i].y));
00176     }
00177     convex.setFillColor(randomBuildingColors[i % randomBuildingColors.size()]);
00178
00179     window.draw(convex);
00180   }
00181
00182   // Draw roads
00183   sf::Color roadColor(194, 201, 202);
00184   for (const auto &road : cityMap.getRoads()) {
00185     for (const auto &segment : road.segments) {
00186       sf::Vector2f basedP1(segment.p1.x, segment.p1.y);
00187       sf::Vector2f basedP2(segment.p2.x, segment.p2.y);
00188
00189       sf::Angle angle = segment.angle;
00190
00191       sf::Vector2f widthVec({sin(angle.asRadians()), -cos(angle.asRadians())});
00192       widthVec *= (float)road.width / 2;
00193
00194       sf::Vector2f p1 = basedP1 + widthVec;
00195       sf::Vector2f p2 = basedP1 - widthVec;
00196       sf::Vector2f p3 = basedP2 - widthVec;
00197       sf::Vector2f p4 = basedP2 + widthVec;
00198
00199       sf::ConvexShape convex;
00200       convex.setPointCount(4);
00201       convex.setPoint(0, p1);
00202       convex.setPoint(1, p2);
00203       convex.setPoint(2, p3);
00204       convex.setPoint(3, p4);
00205
00206       convex.setFillColor(roadColor);
00207
00208       window.draw(convex);
00209
00210       // Draw a circle at the start end end of the road (for filling the gap)
00211       double radius = road.width / 2;
00212       sf::CircleShape circle(radius);
00213       circle.setFillColor(roadColor);
00214       circle.setPosition({(float)(basedP1.x - radius), (float)(basedP1.y - radius)});
00215       window.draw(circle);
00216       circle.setPosition({(float)(basedP2.x - radius), (float)(basedP2.y - radius)});
```

```
00217       window.draw(circle);
00218     }
00219   }
00220
00221   // Draw intersections
00222   if (debug) {
00223     for (const auto &intersection : cityMap.getIntersections()) {
00224       double radius = intersection.radius;
00225       sf::CircleShape circle(radius);
00226       circle.setFillColor(sf::Color(0, 255, 0, 50));
00227       circle.setPosition({(float)(intersection.center.x - radius), (float)(intersection.center.y -
      radius)});
00228       window.draw(circle);
00229     }
00230   }
00231 }
00232
00233 void Renderer::renderCityGraph(const CityGraph &cityGraph, const sf::View &view) {
00234   std::unordered_set<CityGraph::point> graphPoints = cityGraph.getGraphPoints();
00235   std::unordered_map<CityGraph::point, std::vector<CityGraph::neighbor» neighbors =
      cityGraph.getNeighbors();
00236
00237   // Draw a line between each point and its neighbors
00238   for (const auto &point : graphPoints) {
00239     for (const auto &neighbor : neighbors[point]) {
00240       if (!neighbor.isRightWay)
00241         continue;
00242
00243       double radius = turningRadius(neighbor.maxSpeed);
00244       auto space = ob::DubinsStateSpace(radius, true);
00245       ob::RealVectorBounds bounds(2);
00246       space.setBounds(bounds);
00247
00248       // Draw only if one of the points is inside the view
00249       sf::Vector2f viewCenter = view.getCenter();
00250       sf::Vector2f viewSize = view.getSize();
00251       sf::Vector2f viewMin = viewCenter - viewSize / 2.0f;
00252       sf::Vector2f viewMax = viewCenter + viewSize / 2.0f;
00253
00254       if (point.position.x < viewMin.x && neighbor.point.position.x < viewMin.x) {
00255         continue;
00256       }
00257       if (point.position.x > viewMax.x && neighbor.point.position.x > viewMax.x) {
00258         continue;
00259       }
00260
00261       ob::State *start = space.allocState();
00262       ob::State *end = space.allocState();
00263
00264       start->as<ob::DubinsStateSpace::StateType>()->setXY(point.position.x, point.position.y);
00265       start->as<ob::DubinsStateSpace::StateType>()->setYaw(point.angle.asRadians());
00266
00267       end->as<ob::DubinsStateSpace::StateType>()->setXY(neighbor.point.position.x,
      neighbor.point.position.y);
00268       end->as<ob::DubinsStateSpace::StateType>()->setYaw(neighbor.point.angle.asRadians());
00269
00270       // Draw the Dubins curve
00271       double step = CELL_SIZE / 2.0f;
00272       double distance = space.distance(start, end);
00273       int numSteps = distance / step;
00274       sf::Vector2f lastPosition;
00275       sf::Color randomColor = sf::Color(rand() % 255, rand() % 255, rand() % 255, 60);
00276
00277       for (int k = 0; k < numSteps; k++) {
00278         if (k == 0) {
00279           lastPosition = {point.position.x, point.position.y};
00280           continue;
00281         }
00282
00283         ob::State *state = space.allocState();
00284         space.interpolate(start, end, (double)k / (double)numSteps, state);
00285
00286         double x = state->as<ob::DubinsStateSpace::StateType>()->getX();
00287         double y = state->as<ob::DubinsStateSpace::StateType>()->getY();
00288
00289         double distance = std::sqrt(std::pow(x - lastPosition.x, 2) + std::pow(y - lastPosition.y,
      2));
00290         sf::Angle angle = sf::radians(atan2(y - lastPosition.y, x - lastPosition.x));
00291
00292         // Draw an arrow between the points
00293         drawArrow(window, lastPosition, angle, distance * 0.9, distance * 0.9 / 2, randomColor,
      false);
00294
00295         lastPosition = {(float)x, (float)y};
00296       }
00297
00298       continue;
```

```
00299       // Write the speed of the point
00300       sf::Font font = loadFont();
00301       sf::Text text(font);
00302       text.setString(std::to_string((int)(neighbor.maxSpeed * 3.6f)) + " km/h");
00303       text.setCharacterSize(24);
00304       text.setFillColor(sf::Color::Black);
00305       text.setOutlineColor(sf::Color::White);
00306       text.setOutlineThickness(1.0f);
00307       text.setPosition(point.position * 0.2f + neighbor.point.position * 0.8f);
00308       text.setScale({0.02f, 0.02f});
00309       text.setOrigin({text.getLocalBounds().size.x / 2.0f, text.getLocalBounds().size.y / 2.0f});
00310       window.draw(text);
00311     }
00312
00313     // Draw a dot at each points
00314     double size = 0.3;
00315     sf::CircleShape circle(size);
00316     circle.setFillColor(sf::Color(255, 0, 0, 70));
00317     circle.setPosition({(float)(point.position.x - size), (float)(point.position.y - size)});
00318     window.draw(circle);
00319   }
00320 }
00321
00322 void Renderer::renderManager(Manager &manager) { manager.renderAgents(window); }
00323
00324 void Renderer::renderTime() {
00325   // At the top right corner of the view (keep the same size even if the view is resized)
00326   sf::Font font = loadFont();
00327   sf::Text text(font);
00328   sf::Vector2f viewSize = window.getView().getSize();
00329   text.setCharacterSize(24);
00330   text.setFillColor(sf::Color::White);
00331   text.setPosition(window.getView().getCenter() + sf::Vector2f(viewSize.x / 2, -viewSize.y / 2) +
00332                    sf::Vector2f(-viewSize.x * 0.01f, viewSize.y * 0.01f));
00333   text.setString(std::to_string((int)time) + " s");
00334   text.setOutlineColor(sf::Color::Black);
00335   text.setOutlineThickness(1.0f);
00336   text.scale({viewSize.x * 0.001f, viewSize.x * 0.001f});
00337   text.setOrigin({text.getLocalBounds().size.x, 0});
00338   window.draw(text);
00339 }
```

7.49 test.cpp File Reference

A file for testing the project.

```
#include "test.h"
#include "config.h"
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Window/VideoMode.hpp>
#include <spdlog/spdlog.h>
#include <tinyxml2.h>
```

7.49.1 Detailed Description

A file for testing the project.

Definition in file test.cpp.

7.50 test.cpp

Go to the documentation of this file.

```
00001
00005 #include "test.h"
00006 #include "config.h"
00007 #include <SFML/Audio.hpp>
00008 #include <SFML/Graphics.hpp>
00009 #include <SFML/Window/VideoMode.hpp>
00010 #include <spdlog/spdlog.h>
00011 #include <tinyxml2.h>
00012
00013 void Test::runTests() {
00014   testSpdlog();
00015   testTinyXML2();
00016   testSFML();
00017 }
00018
00019 void Test::testSpdlog() {
00020   try {
00021     spdlog::debug("Testing spdlog...");
00022     spdlog::debug("spdlog is working as expected.");
00023   } catch (const std::exception &e) {
00024     throw std::runtime_error("spdlog is not working as expected.");
00025   }
00026 }
00027
00028 void Test::testTinyXML2() {
00029   try {
00030     spdlog::debug("Testing TinyXML2...");
00031     tinyxml2::XMLDocument xmlDoc;
00032     xmlDoc.Parse("<root></root>");
00033     if (xmlDoc.Error()) {
00034       spdlog::error("TinyXML2 is not working as expected.");
00035       throw std::runtime_error("TinyXML2 is not working as expected.");
00036     }
00037     spdlog::debug("TinyXML2 is working as expected.");
00038   } catch (const std::exception &e) {
00039     spdlog::error("TinyXML2 is not working as expected.");
00040     throw std::runtime_error("TinyXML2 is not working as expected.");
00041   }
00042 }
00043
00044 void Test::testSFML() {
00045   try {
00046     spdlog::debug("Testing SFML...");
00047     sf::RenderWindow window(sf::VideoMode({100, 100}), "Test Window");
00048     if (!window.isOpen()) {
00049       spdlog::error("SFML is not working as expected.");
00050       throw std::runtime_error("SFML is not working as expected.");
00051     }
00052     window.close();
00053     spdlog::debug("SFML is working as expected.");
00054   } catch (const std::exception &e) {
00055     spdlog::error("SFML is not working as expected.");
00056     throw std::runtime_error("SFML is not working as expected.");
00057   }
00058 }
```

7.51 utils.cpp File Reference

Utility functions implementation.

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

Functions

- sf::Font loadFont ()

  Load a font.
- bool carsCollided (Car car1, Car car2, int time)
- bool carConflict (sf::Vector2f carPos, sf::Angle carAngle, sf::Vector2f confPos, sf::Angle confAngle)

  Check if two cars have a conflict.

Variables

- bool fontLoaded = false
- sf::Font font

7.51.1 Detailed Description

Utility functions implementation.

Definition in file utils.cpp.

7.51.2 Function Documentation

7.51.2.1 carConflict()

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

Check if two cars have a conflict.

Parameters

carPos	The position of the car
carAngle	The angle of the car
confPos	The position of the conflicting car
confAngle	The angle of the conflicting car

Returns

If the cars have a conflict

Definition at line 47 of file utils.cpp.

```
00047                                                                                                                {
00048     sf::Vector2f diff = carPos - confPos;
00049     return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00050
00051     sf::Vector2f p11 = carPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle.asRadians()),
00052                                              CAR_LENGTH / 2.0f * sin(carAngle.asRadians()));
00053     sf::Vector2f p12 = carPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(carAngle.asRadians()),
00054                                              CAR_LENGTH / 2.0f * sin(carAngle.asRadians()));
00055     sf::Vector2f p21 = confPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00056                                               CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00057     sf::Vector2f p22 = confPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00058                                               CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00059
00060     bool colides = false;
00061     colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00062     colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00063     colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00064     colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00065
00066     return colides;
00067 }
```

7.51.2.2 carsCollided()

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

@bref Check if two cars collided

Parameters

car1	The first car
car2	The second car

Definition at line 21 of file utils.cpp.

```
00021                                                {
00022   std::vector<sf::Vector2f> path1 = car1.getPath();
00023   std::vector<sf::Vector2f> path2 = car2.getPath();
00024   sf::Vector2f diff = path1[time] - path2[time];
00025   return std::sqrt(diff.x * diff.x + diff.y * diff.y) < CAR_LENGTH * 1.1;
00026
00027   sf::Vector2f pos1 = path1[time];
00028   sf::Vector2f pos2 = path2[time];
00029
00030   double angle1 = atan2(path1[time + 1].y - path1[time].y, path1[time + 1].x - path1[time].x);
00031   double angle2 = atan2(path2[time + 1].y - path2[time].y, path2[time + 1].x - path2[time].x);
00032
00033   sf::Vector2f p11 = pos1 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
      sin(angle1));
00034   sf::Vector2f p12 = pos1 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle1), CAR_LENGTH / 2.0f *
      sin(angle1));
00035   sf::Vector2f p21 = pos2 + sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
      sin(angle2));
00036   sf::Vector2f p22 = pos2 - sf::Vector2f(CAR_LENGTH / 2.0f * cos(angle2), CAR_LENGTH / 2.0f *
      sin(angle2));
00037
00038   bool colides = false;
00039   colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00040   colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00041   colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00042   colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00043
00044   return colides;
00045 }
```

7.51.2.3 loadFont()

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

Load a font.

Returns

The font

Definition at line 11 of file utils.cpp.

```
00011                    {
00012   if (!fontLoaded) {
00013     if (!font.openFromFile("assets/fonts/arial.ttf")) {
00014       spdlog::error("Failed to load font");
00015     }
00016     fontLoaded = true;
00017   }
00018   return font;
00019 }
```

7.51.3 Variable Documentation

7.51.3.1 font

```
sf::Font font
```

Definition at line 9 of file utils.cpp.

7.51.3.2 fontLoaded

```
bool fontLoaded = false
```

Definition at line 8 of file utils.cpp.

7.52 utils.cpp

Go to the documentation of this file.

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

```
00054                                           CAR_LENGTH / 2.0f * sin(carAngle.asRadians()));
00055   sf::Vector2f p21 = confPos + sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00056                                             CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00057   sf::Vector2f p22 = confPos - sf::Vector2f(CAR_LENGTH / 2.0f * cos(confAngle.asRadians()),
00058                                             CAR_LENGTH / 2.0f * sin(confAngle.asRadians()));
00059
00060   bool colides = false;
00061   colides |= std::sqrt(std::pow(p11.x - p21.x, 2) + std::pow(p11.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00062   colides |= std::sqrt(std::pow(p11.x - p22.x, 2) + std::pow(p11.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00063   colides |= std::sqrt(std::pow(p12.x - p21.x, 2) + std::pow(p12.y - p21.y, 2)) < CAR_LENGTH * 1.1;
00064   colides |= std::sqrt(std::pow(p12.x - p22.x, 2) + std::pow(p12.y - p22.y, 2)) < CAR_LENGTH * 1.1;
00065
00066   return colides;
00067 }
```

Index